LAFF-On Programming for High Performance

Robert A. van de Geijn
Margaret E. Myers
Devangi N. Parikh

LAFF-On Programming for High Performance

ulaff.net

LAFF-On Programming for High Performance

ulaff.net

Robert van de Geijn
The University of Texas at Austin

Margaret Myers
The University of Texas at Austin

Devangi Parikh
The University of Texas at Austin

November 25, 2021

Cover: Texas Advanced Computing Center

Edition: Draft Edition 2018–2019

Website: ulaff.net

Acknowledgements

We would like to thank the people who created PreTeXt, the authoring system used to typeset these materials. We applaud you!

This work is funded in part by the National Science Foundation (Awards ACI-1550493 and CCF-1714091).

Preface

Over the years, we have noticed that many ideas that underlie programming for high performance can be illustrated through a very simple example: the computation of a matrix-matrix multiplication. It is perhaps for this reason that papers that expose the intricate details of how to optimize matrix-matrix multiplication are often used in the classroom [2] [10]. In this course, we have tried to carefully scaffold the techniques that lead to a high performance matrix-matrix multiplication implementation with the aim to make the materials of use to a broad audience.

It is our experience that some really get into this material. They dive in to tinker under the hood of the matrix-matrix multiplication implementation much like some tinker under the hood of a muscle car. Others merely become aware that they should write their applications in terms of libraries that provide high performance implementations of the functionality they need. We believe that learners who belong to both extremes, or in between, benefit from the knowledge this course shares.

Robert van de Geijn
Maggie Myers
Devangi Parikh
Austin, 2019

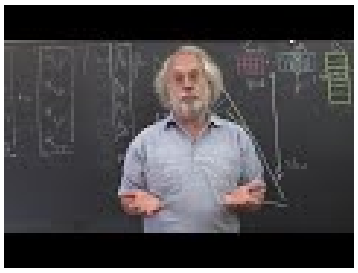
Contents

Week 0

Getting Started

0.1 Opening Remarks

0.1.1 Welcome to LAFF-On Programming for High Performance



YouTube: https://www.youtube.com/watch?v=CaaGjkQF4rU

These notes were written for a Massive Open Online Course (MOOC) that is offered on the edX platform. The first offering of this course starts on June 5, 2019 at 00:00 UTC (which is late afternoon or early evening on June 4 in the US). This course can be audited for free or you can sign up for a Verified Certificate. Register at https://www.edx.org/course/laff-on-programming-for-high-performance.

While these notes incorporate all the materials available as part of the course on edX, being enrolled there has the following benefits:

- It gives you access to videos with more convenient transcripts. They are to the side so they don't overwrite material and you can click on a sentence in the transcript to move the video to the corresponding point in the video.
- You can interact with us and the other learners on the discussion forum.
- You can more easily track your progress through the exercises.

Most importantly: you will be part of a community of learners that is also studying this subject.

0.1.2 Outline Week 0

Each week is structured so that we give the outline for the week immediately after the "launch:"

0.1.3 What you will learn

This unit in each week informs you of what you will learn in the current week. This describes the knowledge and skills that you can expect to acquire. By revisiting this unit upon completion of the week, you can self-assess if you have mastered the topics.

Upon completion of this week, you should be able to

- Recognize the structure of a typical week.
- Navigate the different components of LAFF-On Programming for High Performance.
- Set up your computer.
- Activate MATLAB Online (if you are taking this as part of an edX MOOC).
- Better understand what we expect you to know when you start and intend for you to know when you finish.

0.2 Navigating the Course

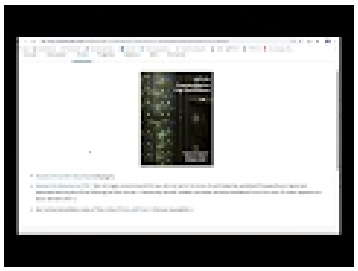
0.2.1 What should we know?

The material in this course is intended for learners who have already had an exposure to linear algebra, programming, and the Linux operating system. We try to provide enough support so that even those with a rudimentary exposure to these topics can be successful. This also ensures that we all "speak the same language" when it comes to notation and terminology. Along the way, we try to give pointers to other materials.

The language in which we will program is C. However, programming assignments start with enough of a template that even those without a previous exposure to C can be successful.

0.2.2 How to navigate the course

Remark 0.2.1 Some of the comments below regard the MOOC, offered on the edX platform, based on these materials.



YouTube: `https://www.youtube.com/watch?v=bYML2j3ECxQ`

0.2.3 When to learn

Remark 0.2.2 Some of the comments below regard the MOOC, offered on the edX platform, based on these materials.

The beauty of an online course is that you get to study when you want, where you want. Still, deadlines tend to keep people moving forward. There are no intermediate due dates but only work that is completed by the closing of the course will count towards the optional Verified Certificate. Even when the course is not officially in session, you will be able to use the notes to learn.

0.2.4 Homework

Remark 0.2.3 Some of the comments below regard the MOOC, offered on the edX platform, based on these materials.

You will notice that homework appears both in the notes and in the corresponding units on the edX platform. Auditors can use the version in the notes. Most of the time, the questions will match exactly but sometimes they will be worded slightly differently so as to facilitate automatic grading.

Realize that the edX platform is ever evolving and that at some point we had to make a decision about what features we would embrace and what features did not fit our format so well. As a result, homework problems have frequently been (re)phrased in a way that fits both the platform and our course.

Some things you will notice:

- "Open" questions in the text are sometimes rephrased as multiple choice questions in the course on edX.
- Sometimes we simply have the video with the answer right after a homework because edX does not support embedding videos in answers.

Please be patient with some of these decisions. Our course and the edX platform are both evolving, and sometimes we had to improvise.

0.2.5 Grading

Remark 0.2.4 Some of the comments below regard the MOOC, offered on the edX platform, based on these materials.

How to grade the course was another decision that required compromise. Our fundamental assumption is that you are taking this course because you want to learn the material, and that the homework and other activities are mostly there to help you learn and self-assess. For this reason, for the homework, we

- Give you an infinite number of chances to get an answer right;
- Provide you with detailed answers;
- Allow you to view the answer any time you believe it will help you master the material efficiently;

- Include some homework that is ungraded to give those who want extra practice an opportunity to do so while allowing others to move on.

In other words, you get to use the homework in whatever way helps you learn best.

Remark 0.2.5 Don't forget to click on "Check" or you don't get credit for the exercise!

To view your progress while the course is in session, click on "Progress" in the edX navigation bar. If you find out that you missed a homework, scroll down the page and you will be able to identify where to go to fix it. Don't be shy about correcting a missed or incorrect answer. The primary goal is to learn.

Some of you will be disappointed that the course is not more rigorously graded, thereby (to you) diminishing the value of a certificate. The fact is that MOOCs are still evolving. People are experimenting with how to make them serve different audiences. In our case, we decided to focus on quality material first, with the assumption that for most participants the primary goal for taking the course is to learn.

Remark 0.2.6 Let's focus on what works, and please be patient with what doesn't!

0.3 Setting Up

0.3.1 Hardware requirements

While you may learn quite a bit by simply watching the videos and reading these materials, real comprehension comes from getting your hands dirty with the programming assignments.

We assume that you have access to a computer with a specific instruction set: AVX2. In our discussions, we will assume you have access to a computer with an Intel Haswell processor or a more recent processor. More precisely, you will need access to one of the following "x86_64" processors:

- Intel architectures: Haswell, Broadwell, Skylake, Kaby Lake, Coffee Lake.
- AMD architectures: Ryzen/Epyc

How do can you tell what processor is in your computer?

- On the processor.

  Instructions on how to interpret markings on an Intel processor can be found at https://www.intel.com/content/www/us/en/support/articles/000006059/processors.html.

- From a label.

  Instructions on how to interpret some labels you may find on your computer can be also be found at https://www.intel.com/content/www/us/en/support/articles/000006059/processors.html.

- Windows.

  Instructions for computers running a Windows operating system can, once again, be found at https://www.intel.com/content/www/us/en/support/articles/000006059/processors.html.

- Apple Mac.

  - In the terminal session type

    ```
    sysctl -n machdep.cpu.brand_string
    ```

  - Cut and paste whatever is reported into a Google search.

- Linux.

  In a terminal window execute

  ```
  more /proc/cpuinfo
  ```

Through one of these means, you will determine the model name of your processor. In our case, we determined the model name

```
Intel(R) Core(TM) i7-8850H CPU @ 2.60GHz
```

After a bit of digging, we discovered that i7-8850H belongs to the "Coffee Lake" family of processors: https://ark.intel.com/content/www/us/en/ark/products/134899/intel-core-i7-8850h-processor-9m-cache-up-to-4-30-ghz.html. With a bit more digging, we find out that it supports AVX2 instructions.

Remark 0.3.1 As pointed out by a learner, the following commands give a list of the features of your CPU if you are running Linux:

```
sysctl -a | grep machdep.cpu.features
sysctl -a | grep machdep.cpu.leaf7_features
```

If one of these yields a list that includes AVX2, you're set. On my mac, the second command works did the trick.

0.3.2 Operating system

We assume you have access to some variant of the Linux operating system on your computer.

0.3.2.1 Linux

If you are on a computer that already uses Linux as its operating system, you are all set. (Although... It has been reported that some linux installations don't automatically come with the gcc compiler, which you will definitely need.)

0.3.2.2 Windows

Sorry, we don't do Windows... But there are options.

- In theory, there is some way of installing a native version of Linux if you are using Windows 10. Instructions can be found at https://docs.microsoft.com/en-us/windows/wsl/install-win10.
- Alternatively, you can install an open-source version of Linux instead of, or in addition to, your Windows OS. Popular versions include Ubuntu, for which instructions can be found at https://www.ubuntu.com/download/desktop.
- Finally, you may want to use a Virtual Machine (see below).

0.3.2.3 Mac OS-X

Mac OS-X comes with Darwin, an open-source Unix-like operating system (https://en.wikipedia.org/wiki/Darwin_(operating_system)).

A complication on the Max OS-X operating system is that the gcc compiler defaults to their clang compiler, which does not by default include the ability to use OpenMP directives, which we will need for the material on using multiple threads and cores in Week 4.

There are a few tools that you will want to install to have the complete toolchain that we will use. We will walk you through how to set this up in Unit 0.3.3

Again, alternatively you may want to use a Virtual Machine (see below).

0.3.2.4 Virtual box alternative

Remark 0.3.2 Note: if you run into problems, you will need to ask someone with more expertise with virtual machines that we have (for example, on the edX discussion board).

Non-Linux users can also install a Linux virtual machine onto both Windows and macOS using VirtualBox from Oracle, which is free to download.

Essentially, you can run a Linux OS from within Windows / macOS without having to actually install it on the hard drive alongside Windows / macOS, and (for Windows users) without having to go through Microsoft Store.

The basic process looks like this:

- Install VirtualBox.
- Download an .iso of an Ubuntu distribution. UbuntuMATE is a safe choice. Make sure to download the 64-bit version.
- In VirtualBox.
  - Create a New Machine by clicking on the "New" button.
  - Give your virtual machine (VM) a name. For the Type, select "Linux" and for the Version, select "Ubuntu (64-bit)"
  - On the next page, set the memory to 2048 MB.

- Continue through the prompts, opting for the default selections.
- Once your VM is created, click on Settings -> System -> Processor. Under the "Processor(s)" menu select upto 4 processors.
- Load the .iso you downloaded above.

  Under Settings->Storage->Controller IDE Select the "Empty" disk on the left panel. On the right panel under "Optical Drive", click on the icon that looks like a CD. Select "Choose Virtual Optical Disk File", and select the .iso you downloaded from the web.
- Start the VM.
- Go through the prompts of installing Ubuntu on the VM, opting for the default selections.
- Once Ubuntu is set up (this may take a few minutes), start a terminal and install gcc, make, and git.

  ```
  sudo apt-get install gcc
  sudo apt-get install make
  sudo apt-get install git
  ```

Now you are set up to continue with the installation of BLIS.

Note: The VM does not automatically detect the hardware of the host machine, hence when configuring BLIS, you will have to explicitly configure using the correct configuration for your architectures (e.g., Haswell). That can be done with the following configure options (in this example for the Haswell architecture)::

```
./configure -t openmp -p ~/blis haswell
```

Then the user can access the virtual machine as a self-contained contained Linux OS.

0.3.3 Installing BLIS

For our programming assignments, we will need to install the BLAS-like Library Instantiation Software (BLIS). You will learn more about this library in the enrichment in Unit 1.5.2.

The following steps will install BLIS if you are using the Linux OS (on a Mac, there may be a few more steps, which are discussed later in this unit.)

- Visit the BLIS Github repository.
- Click on

  and copy https://github.com/flame/blis.git.
- In a terminal session, in your home directory, enter

  ```
  git clone https://github.com/flame/blis.git
  ```

(to make sure you get the address right, you will want to paste the address you copied in the last step.)

- Change directory to blis:

  ```
  cd blis
  ```

- Indicate a specific version of BLIS so that we all are using the same release:

  ```
  git checkout pfhp
  ```

- Configure, build, and install with OpenMP turned on.

  ```
  ./configure -t openmp -p ~/blis auto
  make -j8
  make check -j8
  make install
  ```

  The -p ~/blis installs the library in the subdirectory ~/blis of your home directory, which is where the various exercises in the course expect it to reside.

- If you run into a problem while installing BLIS, you may want to consult https://github.com/flame/blis/blob/master/docs/BuildSystem.md.

On Mac OS-X

- You may need to install Homebrew, a program that helps you install various software on you mac. Warning: you may need "root" privileges to do so.

  ```
  $ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master
  ```

  Keep an eye on the output to see if the “Command Line Tools” get installed. This may not be installed if you already have Xcode Command line tools installed. If this happens, post in the "Discussion" for this unit, and see if someone can help you out.

- Use Homebrew to install the gcc compiler:

  ```
  $ brew install gcc
  ```

  Check if gcc installation overrides clang:

  ```
  $ which gcc
  ```

  The output should be /usr/local/bin. If it isn’t, you may want to add /usr/local/bin to "the path." I did so by inserting

  ```
  export PATH="/usr/local/bin:$PATH"
  ```

  into the file .bash_profile in my home directory. (Notice the "period" before "bash_profile"

- Now you can go back to the beginning of this unit, and follow the instructions to install BLIS.

0.3.4 Cloning the LAFF-On-PfHP repository

We have placed all materials on GitHub. GitHub is a development environment for software projects. In our case, we use it to disseminate the various activities associated with this course.

On the computer on which you have chosen to work, "clone" the GitHub repository for this course:

- Visit https://github.com/ULAFF/LAFF-On-PfHP
- Click on

  and copy https://github.com/ULAFF/LAFF-On-PfHP.git.
- On the computer where you intend to work, in a terminal session on the command line in the directory where you would like to place the materials, execute

  ```
  git clone https://github.com/ULAFF/LAFF-On-PfHP.git
  ```

  This will create a local copy (clone) of the materials.
- Sometimes we will update some of the files from the repository. When this happens you will want to execute, in the cloned directory,

  ```
  git stash save
  ```

  which saves any local changes you have made, followed by

  ```
  git pull
  ```

  which updates your local copy of the repository, followed by

  ```
  git stash pop
  ```

  which restores local changes you made. This last step may require you to "merge" files that were changed in the repository that conflict with local changes.

Upon completion of the cloning, you will have a directory structure similar to that given in Figure 0.3.3.

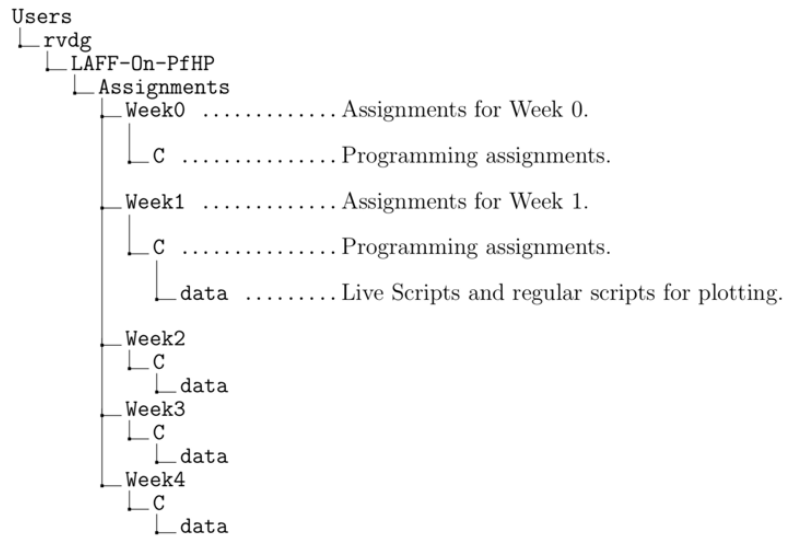


Figure 0.3.3: Directory structure for your basic LAFF-On-PfHP materials. In this example, we cloned the repository in Robert's home directory, rvdg.

Homework 0.3.4.1 We start by checking whether your environment is set up correctly for the programming exercises. In a terminal window, change to the directory Assignments/Week0/C/. In it, at the prompt, type
`make HelloWorld`

This will compile, link, and execute the driver in file main.c.

Solution. If all went well, you will see output that looks like

```
gcc -O3 -I/Users/rvdg/blis/include/blis -m64 -mavx2 -std=c99 -march=native -fopenmp -D_POSIX_C_
gcc -O3 -I/Users/rvdg/blis/include/blis -m64 -mavx2 -std=c99 -march=native -fopenmp -D_POSIX_C_
gcc -O3 -I/Users/rvdg/blis/include/blis -m64 -mavx2 -std=c99 -march=native -fopenmp -D_POSIX_C_
gcc -O3 -I/Users/rvdg/blis/include/blis -m64 -mavx2 -std=c99 -march=native -fopenmp -D_POSIX_C_
gcc driver.o FLA_Clock.o MaxAbsDiff.o RandomMatrix.o /Users/rvdg/blis/lib/libblis.a -o driver.x
echo

./driver.x
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

(The number of times "Hello World" is printed may vary depending on how many cores your processor has.)

Remark 0.3.4

- If make HelloWorld indicates libblis.a is not found, then you likely did not install that library correction in Unit 0.3.3. In particular, you likely skipped make install.
- If you get the error message

  ```
  clang: error: unsupported option '-fopenmp'
  ```

  Then you need to set the path to the gcc compiler. See Unit 0.3.3.

0.3.5 Get a copy of the book

Remark 0.3.5 We are still working on putting the mechanism in place by which you can obtain your own copy of the book. So, this section is merely a preview of what will become available in the future.

For now, you can access a PDF of these notes at http://www.cs.utexas.edu/users/flame/laff/pfhp/LAFF-On-PfHP.pdf. Expect this PDF to be frequently updated.

By now, you will have experienced the electronic book we have authored with PreTeXt for this course. We host a copy of that book at `http://www.cs.utexas.edu/users/flame/laff/pfhp/LAFF-On-PfHP.html` so that you can use it, free of charge, either as part of the course on edX or independent of that.

We make the book available at `http://ulaff.net` to those who also want a copy local on their computer, in a number of formats:

- As the online document, except now downloaded to your computer.
- As a PDF that incorporates the solutions to all the homeworks embedded in the text.
- As a more concise PDF that has all the homeworks, but not the solutions.
- As a PDF that includes only the solutions.

IN THE FUTURE: By following the directions at `http://ulaff.net`, you can download a "ZIP" file. First clone the LAFF-On-PfHP repository as described earlier. Then unzip the "ZIP" file in the directory LAFF-On-PfHP it created, then you end up with the materials illustrated in Figure 0.3.6.

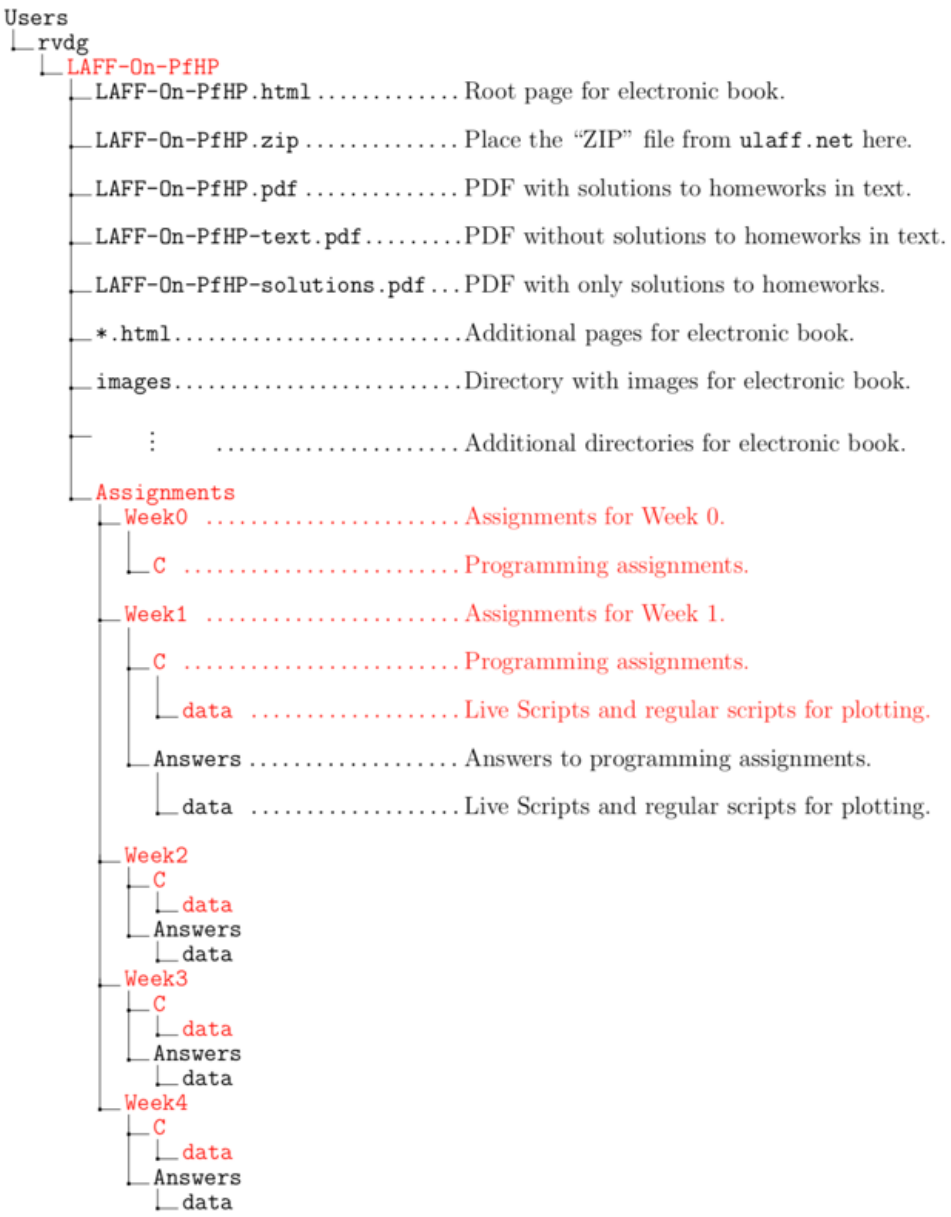


Figure 0.3.6: Directory structure for your basic LAFF-On-PfHP materials after unzipping LAFF-On-PfHP.zip from http://ulaff.net in the directory created by cloning the LAFF-On-PfHP repository. The materials you obtained by cloning are highlighted in red.

You will now have the material self-contained on your computer.

Remark 0.3.7 For the online book to work properly, you will still have to be connected to the internet.

0.3.6 Matlab

We will use Matlab to create performance graphs for your various implementations of matrix-matrix multiplication.

There are a number of ways in which you can use Matlab:

- Via MATLAB that is installed on the same computer as you will execute your performance experiments. This is usually called a "desktop installation of Matlab."

- Via MATLAB Online. You will have to transfer files from the computer where you are performing your experiments to MATLAB Online. You could try to set up MATLAB Drive, which allows you to share files easily between computers and with MATLAB Online. Be warned that there may be a delay in when files show up, and as a result you may be using old data to plot if you aren't careful!

If you are using these materials as part of an offering of the Massive Open Online Course (MOOC) titled "LAFF-On Programming for High Performance" on the edX platform, you will be given a temporary license to Matlab, courtesy of MathWorks.

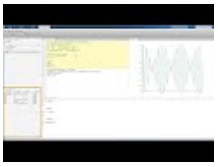
You need relatively little familiarity with MATLAB in order to learn what we want you to learn in this course. So, you could just skip these tutorials altogether, and come back to them if you find you want to know more about MATLAB and its programming language (M-script).

Below you find a few short videos that introduce you to MATLAB. For a more comprehensive tutorial, you may want to visit MATLAB Tutorials at MathWorks and click "Launch Tutorial".

What is MATLAB?

https://www.youtube.com/watch?v=2sB-NMD9Qhk

Getting Started with MATLAB Online



https://www.youtube.com/watch?v=4shp284pGc8

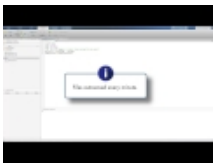
MATLAB Variables

https://www.youtube.com/watch?v=gPIsIzHJA9I

MATLAB as a Calculator

https://www.youtube.com/watch?v=K9xy5kQHDBo

Managing Files with MATLAB Online



https://www.youtube.com/watch?v=mqYwMnM-x5Q

0.3.7 Setting up MATLAB Online

Once you have watched "Getting started with MATLAB Online" and set up your account, you will want to set up the directory in which you will place the materials for the course. Follow these steps:

- While logged into MATLAB online, create a folder "LAFF-On-PfHP" (or with a name of your choosing).
- Click on the folder name to change directory to that folder.
- On your computer, download `http://www.cs.utexas.edu/users/flame/laff/pfhp/Assignments.zip`. It doesn't really matter where you store it.
- In the MATLAB Online window, click on the "HOME" tab.
- Double click on "Upload" (right between "New" and "Find Files") and upload Assignments.zip from wherever you stored it.
- In the MATLAB Online window, find the "COMMAND WINDOW" (this window should be in the lower-right corner of the MATLAB Online window).
- In the "COMMAND WINDOW" execute

  ```
  ls
  ```

  Thus should print the contents of the current directory. If Assignments.zip is not listed as a file in the directory, you will need to fix this by changing to the appropriate directory.
- Execute

  ```
  unzip Assignments
  ```

  in the command window. This may take a while
- In the "CURRENT FOLDER" window, you should now see a new folder "Assignments." If you double click on that folder name, you should see five folders: Week0, Week1, Week2, Week3, and Week4.

You are now set up for the course.

Remark 0.3.8 If you also intend to use a desktop copy of MATLAB, then you may want to coordinate the local content with the content on MATLAB Online with MATLAB Drive: https://www.mathworks.com/products/matlab-drive.html. However, be warned that there may be a delay in files being synchronized. So, if you perform a performance experiment on your computer and move the resulting file into the MATLAB Drive directory, the data file may not immediately appear in your MATLAB Online session. This may lead to confusion if you are visualizing performance and are looking at graphs created from stale data.

Remark 0.3.9 We only use MATLAB Online (or your desktop MATLAB) to create performance graphs. For this, we recommend you use the Live Scripts that we provide. To work with Live Scripts, you will want to make sure the "LIVE EDITOR" tab is where you are working in MATLAB Online. You can ensure this is the case by clicking on the "LIVE EDITOR" tab.

0.4 Experimental Setup

0.4.1 Programming exercises

For each week, we will explore high-performance programming through a number of programming exercises. To keep the course accessible to a wide audience, we limit the actual amount of coding so as to allow us to focus on the key issues. Typically, we will only program a few lines of code. In the remainder of this section, we discuss the setup that supports this.

Most of the material in the section is here for reference. It may be best to quickly scan it and then revisit it after you have completed the first programming homework in Unit 1.1.1.

0.4.2 MyGemm

The exercises revolve around the optimization of implementations of matrix-matrix multiplication. We will write and rewrite the routine

```
MyGemm( m, n, k, A, ldA, B, ldB, C, ldC )
```

which computes $C := AB + C$, where C is $m \times n$, A is $m \times k$, and B is $k \times n$.

"Gemm" is a commonly used acronym that stands for "Ge"neral "m"atrix "m"ultiplication. More on this in the enrichment in Unit 1.5.1.

0.4.3 The driver

The term "driver" is typically used for a main program that exercises (tests and/or times) functionality implemented in a routine. In our case, the driver tests a range of problem sizes by creating random matrices C, A, and B. You will want to look through the file "driver.c" in Week 1. Hopefully the comments are self-explanatory. If not, be sure to ask questions on the discussion board.

0.4.4 How we use makefiles

YouTube: https://www.youtube.com/watch?v=K0Ymbpvk59o

Example 0.4.1 When one wants to execute an implementation of matrix-matrix multiplication in, for example, file Gemm_IJP.c that will be discussed in Unit 1.1.1, a number of steps need to come together.

- We have already discussed the driver routine in `Assignments/Week1/C/driver.c`. It is compiled into an object file (an intermediate step towards creating an executable file) with

  ```
  gcc <options> -c -o driver.o driver.c
  ```

  where <options> indicates a bunch of options passed to the gcc compiler.

- The implementation of matrix-matrix multiplication is in file Gemm_IJP.c which is compiled into an object file with

  ```
  gcc <options> -c -o Gemm_IJP.o Gemm_IJP.c
  ```

- A routine for timing the executions (borrowed from the libflame library is in file FLA_Clock.c and is compiled into an object file with

  ```
  gcc <options> -c -o FLA_Clock.o FLA_Clock.c
  ```

- A routine that compares two matices so that correctness can be checked is in file MaxAbsDiff.c and is compiled into an object file with

  ```
  gcc <options> -c -o MaxAbsDiff.o MaxAbsDiff.c
  ```

- A routine that creates a random matrix is in file RandomMatrix.c and is compiled into an object file with

  ```
  gcc <options> -c -o RandomMatrix.o RandomMatrix.c
  ```

- Once these object files have been created, they are linked with

  ```
  gcc driver.o Gemm_IJP.o FLA_Clock.o MaxAbsDiff.o  RandomMatrix.o
         -o driver_IJP.x <libraries>
  ```

  where <libraries> is a list of libraries to be linked.

Together, these then create the executable driver_IJP.x that can be executed with

```
./driver_IJP.x
```

This should be interpreted as "executed the file driver.x in the current directory." The driver is set up to take input:

```
% number of repeats:3
% 3
% enter first, last, inc:100 500 100
% 100 500 100
```

where "3" and "100 500 100" are inputs. □

Many decades ago, someone came up with the idea of creating a file in which the rules of how to, in this situation, create an executable from parts could be spelled out. Usually, these rules are entered in a Makefile. Below is a video that gives a quick peek into the Makefile for the exercises in Week 1.

YouTube: https://www.youtube.com/watch?v=K0Ymbpvk59o

The bottom line is: by using that Makefile, to make the executable for the implementation of matrix-matrix multiplication in the file Gemm_IJP.c all one has to do is type

```
make IJP
```

Better yet, this also then "pipes" input to be used by the driver, and redirects the output into a file for later use when we plot the results.

You can read up on Makefiles on, where else, Wikipedia: https://en.wikipedia.org/wiki/Makefile.

Remark 0.4.2 Many of us have been productive software developers without ever reading a manual on how to create a Makefile. Instead, we just copy one that works, and modify it for our purposes. So, don't be too intimidated!

0.5 Enrichments

> In each week, we include "enrichments" that allow the participant to go beyond.

0.5.1 High-performance computers are for high-performance computing

Some of the fastest computers in the world are used for scientific computing. At UT-Austin, the Texas Advanced Computing Center (TACC) has some of the fastest academic machines.

You may enjoy the following video on Stampede, until recently the workhorse machine at TACC.

YouTube: https://www.youtube.com/watch?v=HMEL1RVa1E8

This is actually not the most recent machine. It was succeeded by Stampede2.

Now, what if these machines, which cost tens of millions of dollars, only achieved a fraction of their peak performance? As a taxpayer, you would be outraged. In this course, you find out how, for some computations, near-peak performance can be attained.

0.6 Wrap Up

0.6.1 Additional Homework

For a typical week, additional assignments may be given in this unit.

0.6.2 Summary

In a typical week, we provide a quick summary of the highlights in this unit.

Week 1

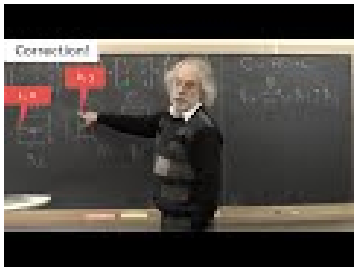
Loops and More Loops

1.1 Opening Remarks

1.1.1 Launch

Homework 1.1.1.1 Compute $\begin{pmatrix} 1 & -2 & 2 \\ 1 & 1 & 3 \\ -2 & 2 & 1 \end{pmatrix} \begin{pmatrix} -2 & 1 \\ 1 & 3 \\ -1 & 2 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ -1 & 2 \\ -2 & 1 \end{pmatrix} =$

Answer. $\begin{pmatrix} 1 & -2 & 2 \\ 1 & 1 & 3 \\ -2 & 2 & 1 \end{pmatrix} \begin{pmatrix} -2 & 1 \\ 1 & 3 \\ -1 & 2 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ -1 & 2 \\ -2 & 1 \end{pmatrix} = \begin{pmatrix} -5 & -1 \\ -5 & 12 \\ 3 & 7 \end{pmatrix}$



YouTube: https://www.youtube.com/watch?v=knTLy1j-rco

Let us explicitly define some notation that we will use to discuss matrix-matrix multiplication more abstractly.

Let A, B, and C be $m \times k$, $k \times n$, and $m \times n$ matrices, respectively. We can expose their individual entries as

$$A = \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,k-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,k-1} \\ \vdots & \vdots & & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,k-1} \end{pmatrix}, B = \begin{pmatrix} \beta_{0,0} & \beta_{0,1} & \cdots & \beta_{0,n-1} \\ \beta_{1,0} & \beta_{1,1} & \cdots & \beta_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \beta_{k-1,0} & \beta_{k-1,1} & \cdots & \beta_{k-1,n-1} \end{pmatrix},$$

and

$$C = \begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} & \cdots & \gamma_{0,n-1} \\ \gamma_{1,0} & \gamma_{1,1} & \cdots & \gamma_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \gamma_{m-1,0} & \gamma_{m-1,1} & \cdots & \gamma_{m-1,n-1} \end{pmatrix}.$$

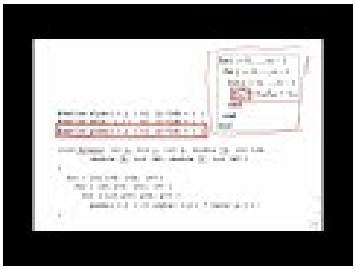
The computation $C := AB + C$, which adds the result of matrix-matrix multiplication AB to a matrix C, is defined as

$$\gamma_{i,j} := \sum_{p=0}^{k-1} \alpha_{i,p}\beta_{p,j} + \gamma_{i,j} \tag{1.1.1}$$

for all $0 \leq i < m$ and $0 \leq j < n$. Why we add to C will become clear later. This leads to the following pseudo-code for computing $C := AB + C$:

for $i := 0, \ldots, m-1$
  for $j := 0, \ldots, n-1$
    for $p := 0, \ldots, k-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end
  end
end

The outer two loops visit each element of C, and the inner loop updates $\gamma_{i,j}$ with (1.1.1).



YouTube: https://www.youtube.com/watch?v=yq6-qN8QXz8

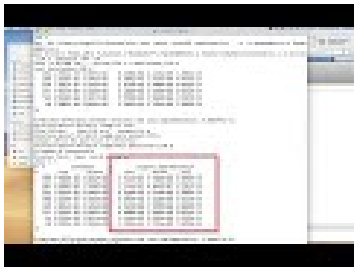
```
#define alpha( i,j ) A[ (j)*ldA + i ]   // map alpha( i,j ) to array A
#define beta( i,j )  B[ (j)*ldB + i ]   // map beta( i,j )  to array B
#define gamma( i,j ) C[ (j)*ldC + i ]   // map gamma( i,j ) to array C

void MyGemm( int m, int n, int k, double *A, int ldA,
             double *B, int ldB, double *C, int ldC )
{
  for ( int i=0; i<m; i++ )
    for ( int j=0; j<n; j++ )
      for ( int p=0; p<k; p++ )
        gamma( i,j ) += alpha( i,p ) * beta( p,j );
}
```

Figure 1.1.1: Implementation, in the C programming language, of the IJP ordering for computing matrix-matrix multiplication.

Homework 1.1.1.2 In the file Assignments/Week1/C/Gemm_IJP.c you will find the simple implementation given in Figure 1.1.1 that computes $C := AB + C$. In the directory Assignments/Week1/C execute

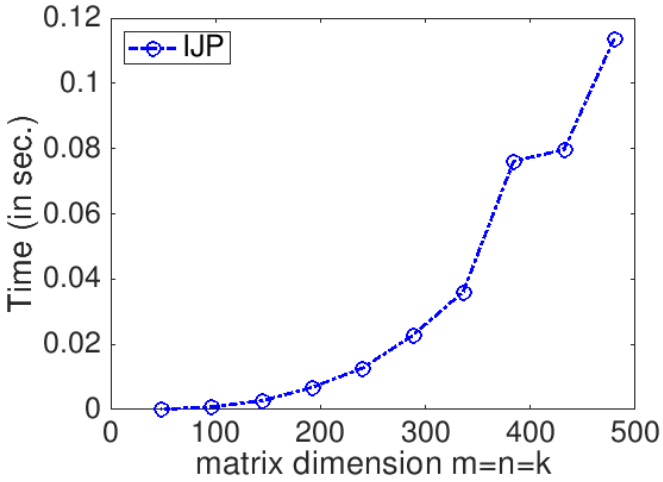
```
make IJP
```

to compile, link, and execute it. You can view the performance attained on your computer with the Matlab Live Script in Assignments/Week1/C/data/Plot_IJP.mlx (Alternatively, read and execute Assignments/Week1/C/data/Plot_IJP_m.m.)

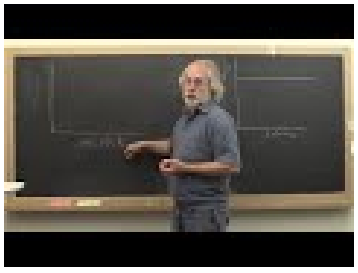


YouTube: https://www.youtube.com/watch?v=3BPCfpqWWCk

On Robert's laptop, Homework 1.1.1.2 yields the graph



as the curve labeled with IJP. The time, in seconds, required to compute matrix-matrix multiplication as a function of the matrix size is plotted, where $m = n = k$ (each matrix is square). The "dips" in the time required to complete can be attributed to a number of factors, including that other processes that are executing on the same processor may be disrupting the computation. One should not be too concerned about those.



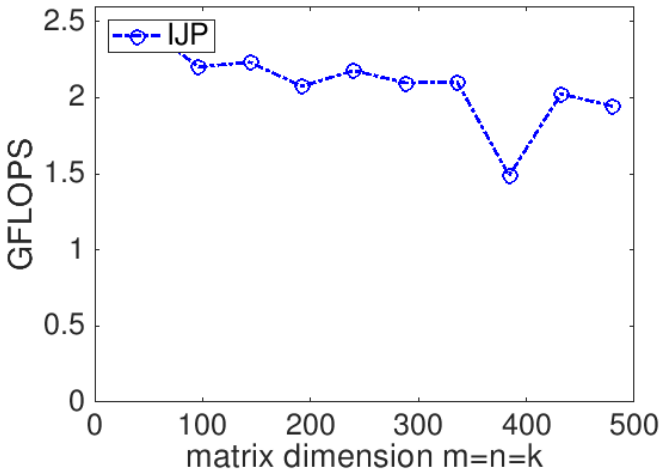
YouTube: https://www.youtube.com/watch?v=cRyVMrNNRCk

The performance of a matrix-matrix multiplication implementation is measured in billions of floating point operations (flops) per second (GFLOPS). The idea is that we know that it takes $2mnk$ flops to compute $C := AB + C$ where C is $m \times n$, A is $m \times k$, and B is $k \times n$. If we measure the time it takes to complete the computation, $T(m, n, k)$, then the rate at which we compute is

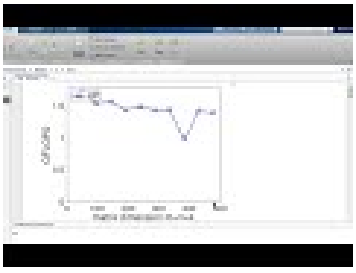
given by

$$\frac{2mnk}{T(m,n,k)} \times 10^{-9} \text{ GFLOPS}.$$

For our implementation and the reference implementation, this yields



Again, don't worry too much about the dips in the curves. If we controlled the environment in which we performed the experiments, (for example, by making sure few other programs are running at the time of the experiments) these would largely disappear.



YouTube: `https://www.youtube.com/watch?v=lNEIVq5_DW4`

Remark 1.1.2 The Gemm in the name of the routine stands for General Matrix-Matrix multiplication. Gemm is an acronym that is widely used in scientific computing, with roots in the Basic Linear Algebra Subprograms (BLAS) discussed in the enrichment in Unit 1.5.1.

1.1.2 Outline Week 1

1.1.3 What you will learn

In this week, we not only review matrix-matrix multiplication, but we also start thinking about this operation in different ways.

Upon completion of this week, we will be able to

- Map matrices to memory.
- Apply conventions to describe how to index into arrays that store matrices.
- Observe the effects of loop order on performance.
- Recognize that simple implementations may not provide the performance that can be achieved.

- Realize that compilers don't automagically do all the optimization.
- Think about matrix-matrix multiplication in terms of matrix-vector multiplications and rank-1 updates.
- Understand how matrix-matrix multiplication can be layered in terms of simpler operations with matrices and/or vectors.
- Relate execution time of matrix-matrix multiplication to the rate of computation achieved by the implementation.
- Plot performance data with Matlab.

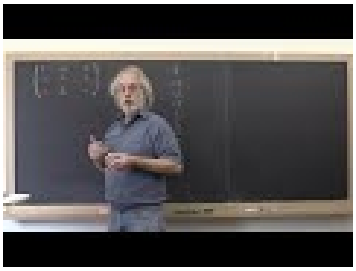
The enrichments introduce us to

- The Basic Linear Algebra Subprograms (BLAS), an interface widely used in the computational science.
- The BLAS-like Library Instantiation Software (BLIS) framework for implementing the BLAS.
- How to count floating point operations.

1.2 Loop Orderings

1.2.1 Mapping matrices to memory

Matrices are usually visualized as two-dimensional arrays of numbers while computer memory is inherently one-dimensional in the way it is addressed. So, we need to agree on how we are going to store matrices in memory.



YouTube: https://www.youtube.com/watch?v=7dUEghwlwt0

Consider the matrix

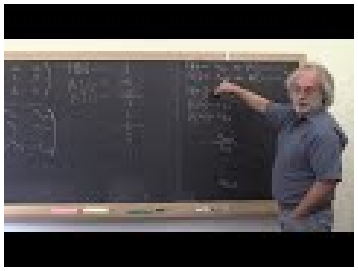
$$\begin{pmatrix} 1 & -2 & 2 \\ -1 & 1 & 3 \\ -2 & 2 & -1 \end{pmatrix}$$

from the opener of this week. In memory, this may be stored in a one-

dimentional array A by columns:

A[0]	$\longrightarrow$	1
A[1]	$\longrightarrow$	-1
A[2]	$\longrightarrow$	-2
A[3]	$\longrightarrow$	-2
A[4]	$\longrightarrow$	1
A[5]	$\longrightarrow$	2
A[6]	$\longrightarrow$	2
A[7]	$\longrightarrow$	3
A[8]	$\longrightarrow$	-1

which is known as column-major ordering.



YouTube: https://www.youtube.com/watch?v=lAxUYWCEeDM

More generally, consider the matrix

$$\begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,n-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,n-1} \end{pmatrix}.$$

Column-major ordering would store this in array A as illustrated by Figure 1.2.1.

.

A[0]	A[0 * m + 0]	$\longrightarrow$	$\alpha_{0,0}$
A[1]	A[0 * m + 1]	$\longrightarrow$	$\alpha_{1,0}$
		$\vdots$	
A[m − 1]	A[0 * m + (m − 1)]	$\longrightarrow$	$\alpha_{m-1,0}$
A[m]	A[1 * m + 0]	$\longrightarrow$	$\alpha_{0,1}$
A[m + 1]	A[1 * m + 1]	$\longrightarrow$	$\alpha_{1,1}$
		$\vdots$	
	A[1 * m + (m − 1)]	$\longrightarrow$	$\alpha_{m-1,1}$
		$\vdots$	
	A[j * m + i]	$\longrightarrow$	$\alpha_{i,j}$
		$\vdots$	
	A[(n − 1) * m + 0]	$\longrightarrow$	$\alpha_{0,n-1}$
	A[(n − 1) * m + 1]	$\longrightarrow$	$\alpha_{1,n-1}$
		$\vdots$	
	A[(n − 1) * m + (m − 1)]	$\longrightarrow$	$\alpha_{m-1,n-1}$

Figure 1.2.1: Mapping of $m \times n$ matrix A to memory with column-major order.

Obviously, one could use the alternative known as row-major ordering.

Homework 1.2.1.1 Let the following picture represent data stored in memory starting at address A:

$$\texttt{A[0]} \longrightarrow \begin{array}{r} 3 \\ 1 \\ -1 \\ -2 \\ -2 \\ 1 \\ 2 \\ 2 \end{array}$$

and let A be the 2×3 matrix stored there in column-major order. Then
$A =$

Answer.

$$A = \begin{pmatrix} 1 & -2 & 1 \\ -1 & -2 & 2 \end{pmatrix}$$

Homework 1.2.1.2 Let the following picture represent data stored in memory

starting at address A.

$$\texttt{A[0]} \longrightarrow \begin{array}{r} 3 \\ 1 \\ -1 \\ -2 \\ -2 \\ 1 \\ 2 \\ 2 \end{array}$$

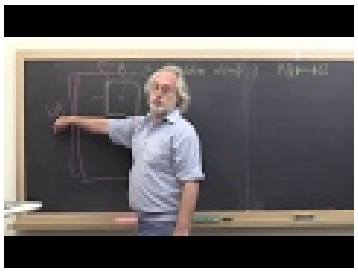
and let A be the 2×3 matrix stored there in row-major order. Then

$$A =$$

Answer.

$$A = \begin{pmatrix} 1 & -1 & -2 \\ -2 & 1 & 2 \end{pmatrix}$$

1.2.2 The leading dimension



YouTube: https://www.youtube.com/watch?v=PhjildK5oO8

Very frequently, we will work with a matrix that is a submatrix of a larger matrix. Consider Figure 1.2.2.

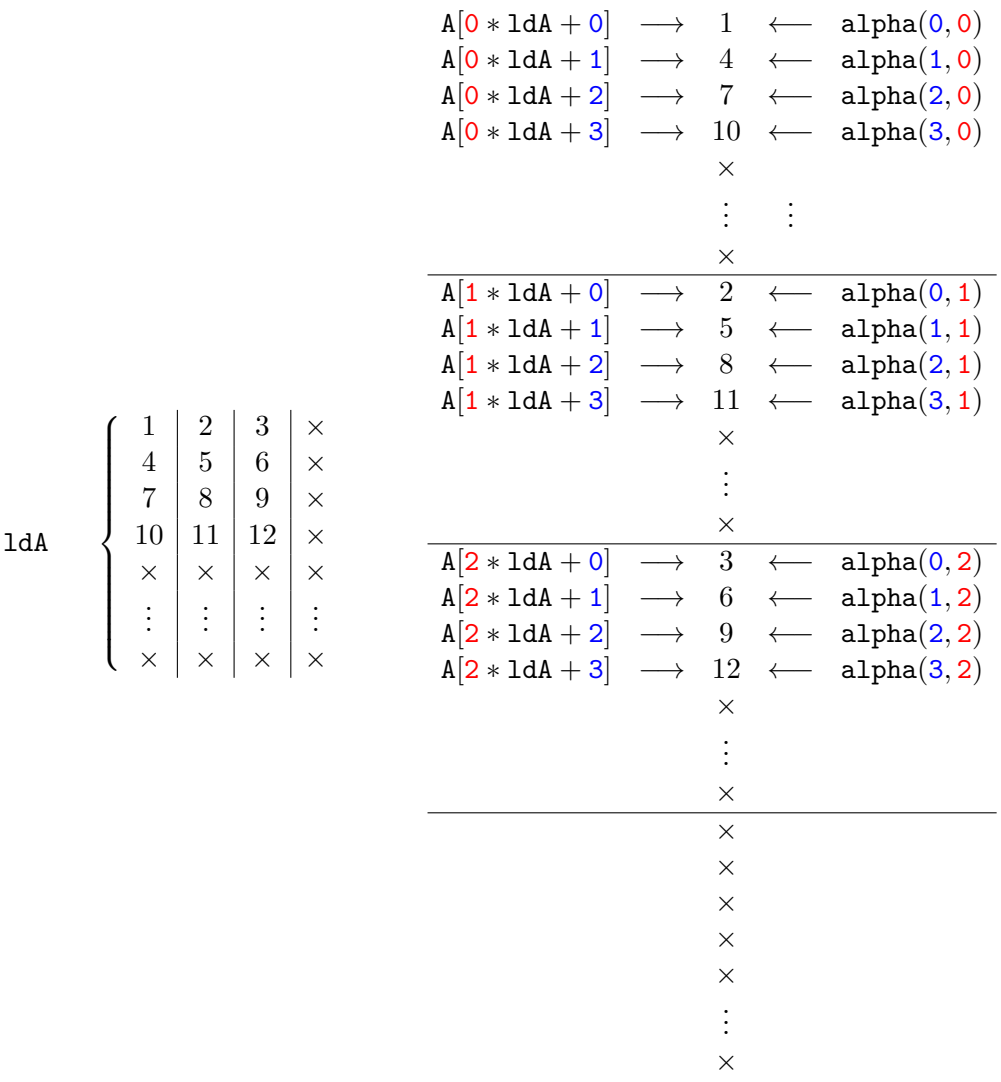


Figure 1.2.2: Addressing a matrix embedded in an array with `ldA` rows. At the left we illustrate a 4×3 submatrix of a `ldA` $\times 4$ matrix. In the middle, we illustrate how this is mapped into an linear array `a`. In the right, we show how defining the C macro #define alpha(i,j) A[(j)*ldA + (i)] allows us to address the matrix in a more natural way.

What we depict there is a matrix that is embedded in a larger matrix. The larger matrix consists of ldA (the leading dimension) rows and some number of columns. If column-major order is used to store the larger matrix, then addressing the elements in the submatrix requires knowledge of the leading dimension of the larger matrix. In the C programming language, if the top-left element of the submatrix is stored at address A, then one can address the (i, j) element as A[j*ldA + i]. In our implementations, we define a macro that makes addressing the elements more natural:

```
#define alpha(i,j) A[ (j)*ldA + (i) ]
```

where we assume that the variable or constant ldA holds the leading dimension parameter.

Remark 1.2.3 Learn more about macro definitions in C at https://gcc.gnu.org/onlinedocs/cpp/Macro-Arguments.html. (We'd be interested in a better tutorial).

Homework 1.2.2.1 Consider the matrix

$$\begin{pmatrix} 0.0 & 0.1 & 0.2 & 0.3 \\ 1.0 & \mathbf{1.1} & \mathbf{1.2} & \mathbf{1.3} \\ 2.0 & \mathbf{2.1} & \mathbf{2.2} & \mathbf{2.3} \\ 3.0 & 3.1 & 3.2 & 3.3 \\ 4.0 & 4.1 & 4.2 & 4.3 \end{pmatrix}$$

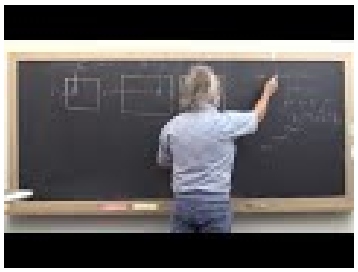
If this matrix is stored in column-major order in a linear array A,

1. The highlighted submatrix starts at A[...].
2. The number of rows (height) of the highlighted submatrix equals ....
3. The number of columns (width) of the highlighted submatrix equals ....
4. The leading dimension of the highlighted submatrix is ....

Solution.

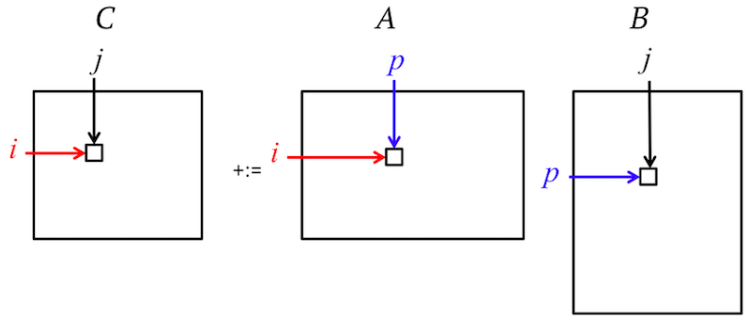
1. The highlighted submatrix starts at A[6].
2. The number of rows of the boxed submatrix is 2.
3. The number of columns of the boxed submatrix is 3.
4. The leading dimension of the boxed submatrix is 5.

1.2.3 A convention regarding the letters used for the loop index



YouTube: https://www.youtube.com/watch?v=l8XFfkoTrHA

When we talk about loops for matrix-matrix multiplication, it helps to keep in mind the picture



which illustrates which loop index (variable name) is used for what row or column of the matrices.

- Index i is used to index the row of C and corresponding row of A.
- Index j is used to index the column of C and corresponding column of B.
- Index p is used to index the column of A and corresponding row of B.

We try to be consistent in this use, as should you.

Remark 1.2.4 In the literature, people often use i, j, and k for indexing the three loops and talk about the ijk loop ordering when we talk about the IJP ordering (later in Week 1). The problem with that convention is that k is also used for the "inner size" of the matrices (the column size of A and the row size of B). It is for this reason that we use p instead.

Our convention comes with its own problems, since p is often used to indicate the number of processors involved in a computation. No convention is perfect!

1.2.4 Ordering the loops

Consider again a simple algorithm for computing $C := AB + C$:

```
for i := 0, ..., m − 1
  for j := 0, ..., n − 1
    for p := 0, ..., k − 1
      γ_{i,j} := α_{i,p} β_{p,j} + γ_{i,j}
    end
  end
end
```

Given that we now embrace the convention that i indexes rows of C and A, j indexes columns of C and B, and p indexes "the other dimension," we can call this the IJP ordering of the loops around the assignment statement.

Different orders in which the elements of C are updated, and the order in which terms of

$$\alpha_{i,0}\beta_{0,j} + \cdots + \alpha_{i,k-1}\beta_{k-1,j}$$

are added to $\gamma_{i,j}$, are mathematically equivalent as long as each

$$\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$$

is performed exactly once. That is, in exact arithmetic, the result does not change. (Computation is typically performed in floating point arithmetic, in which case roundoff error may accumulate slightly differently, depending on the order of the loops. One tends to ignore this when implementing matrix-matrix multiplication.) A consequence is that the three loops can be reordered without changing the result.

Homework 1.2.4.1 The IJP ordering is one possible ordering of the loops. How many distinct reorderings of those loops are there?

Answer.

$$3! = 6.$$

Solution.

- There are three choices for the outer-most loop: i, j, or p.
- Once a choice is made for the outer-most loop, there are two choices left for the second loop.
- Once that choice is made, there is only one choice left for the inner-most loop.

Thus, there are $3! = 3 \times 2 \times 1 = 6$ loop orderings.

Homework 1.2.4.2 In directory Assignments/Week1/C make copies of Assignments/Week1/C/Gemm_IJP.c into files with names that reflect the different loop orderings (Gemm_IPJ.c, etc.). Next, make the necessary changes to the loops in each file to reflect the ordering encoded in its name. Test the implementions by executing

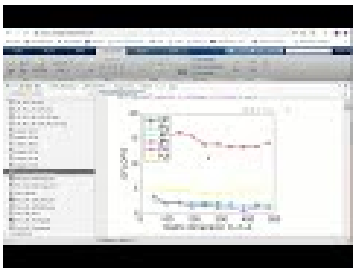
```
make IPJ
make JIP
   ...
```

for each of the implementations and view the resulting performance by making the indicated changes to the Live Script in Assignments/Week1/C/data/Plot_All_Orderings.mlx (Alternatively, use the script in Assignments/Week1/C/data/Plot_All_Orderings_m.m). If you have implemented them all, you can test them all by executing

```
make All_Orderings
```

Solution.

- Assignments/Week1/Answers/Gemm_IPJ.c
- Assignments/Week1/Answers/Gemm_JIP.c
- Assignments/Week1/Answers/Gemm_JPI.c
- Assignments/Week1/Answers/Gemm_PIJ.c
- Assignments/Week1/Answers/Gemm_PJI.c



YouTube: https://www.youtube.com/watch?v=13gSeXQwnrE

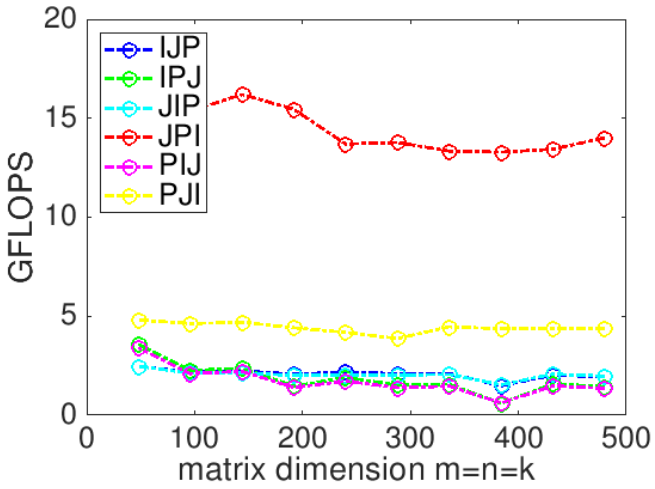
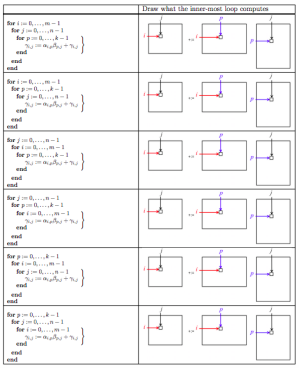


Figure 1.2.5: Performance comparison of all different orderings of the loops, on Robert's laptop.

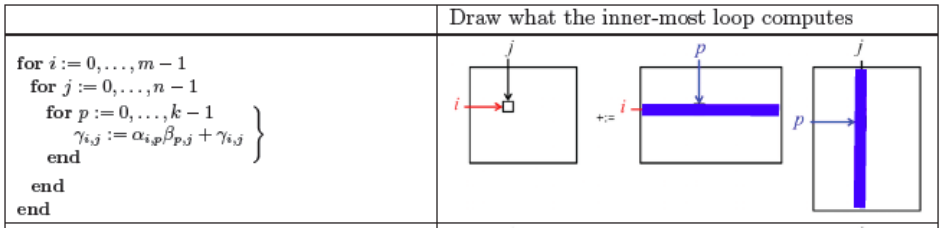
Homework 1.2.4.3 In Figure 1.2.5, the results of Homework 1.2.4.2 on Robert's laptop are reported. What do the two loop orderings that result in the best performances have in common? You may want to use the following worksheet to answer this question:



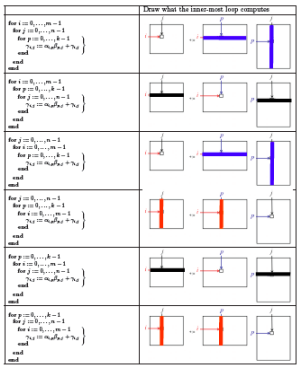
Click here to enlarge.

Figure 1.2.6: You may want to print this figure, and mark how the algorithms access memory, as you work through the homework.

Hint. Here is how you may want to mark the answer for the first set of loops:



Solution.



Click here to enlarge.

Homework 1.2.4.4 In Homework 1.2.4.3, why do they get better performance?

Hint. Think about how matrices are mapped to memory.

Solution. Matrices are stored with column major order, accessing contiguous data usually yields better performance, and data in columns are stored contiguously.

Homework 1.2.4.5 In Homework 1.2.4.3, why does the implementation that gets the best performance outperform the one that gets the next to best performance?

Hint. Think about how and what data get reused for each of the two implementations.

Solution. For both the JPI and PJI loop orderings, the inner loop accesses columns of C and A. However,

- Each execution of the inner loop of the JPI ordering updates the same column of C.
- Each execution of the inner loop of the PJI ordering updates a different column of C.

In Week 2 you will learn about cache memory. The JPI ordering can keep the column of C in cache memory, reducing the number of times elements of C need to be read and written.

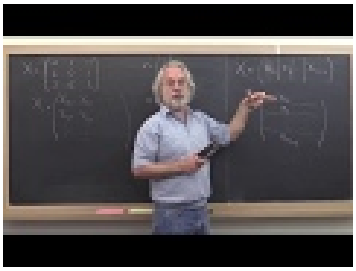
If this isn't obvious to you, it should become clearer in the next unit and

the next week.

Remark 1.2.7 One thing that is obvious from all the exercises so far is that the gcc compiler doesn't do a very good job of automatically reordering loops to improve performance, at least not for the way we are writing the code.

1.3 Layering Matrix-Matrix Multiplication

1.3.1 Notation



YouTube: `https://www.youtube.com/watch?v=mt-kOxpinDc`

In our discussions, we use capital letters for matrices $(A, B, C, \ldots)$, lower case letters for vectors $(a, b, c, \ldots)$, and lower case Greek letters for scalars $(\alpha, \beta, \gamma, \ldots)$. Exceptions are integer scalars, for which we will use k, m, n, i, j, and p.

Vectors in our universe are column vectors or, equivalently, $n \times 1$ matrices if the vector has n components (size n). A row vector we view as a column vector that has been transposed. So, x is a column vector and x^T is a row vector.

In the subsequent discussion, we will want to expose the rows or columns of a matrix. If X is an $m \times n$ matrix, then we expose its columns as

$$X = \left(\begin{array}{c|c|c|c} x_0 & x_1 & \cdots & x_{n-1} \end{array} \right)$$

so that x_j equals the column with index j. We expose its rows as

$$X = \begin{pmatrix} \widetilde{x}_0^T \\ \widetilde{x}_1^T \\ \vdots \\ \widetilde{x}_{m-1}^T \end{pmatrix}$$

so that $\widetilde{x}_i^T$ equals the row with index i. Here the ^{T} indicates it is a row (a column vector that has been transposed). The tilde is added for clarity since x_i^T would in this setting equal the column indexed with i that has been transposed, rather than the row indexed with i. When there isn't a cause for confusion, we will sometimes leave the tilde off. We use the lower case letter that corresponds to the upper case letter used to denote the matrix, as an added visual clue that x_j is a column of X and $\widetilde{x}_i^T$ is a row of X.

We have already seen that the scalars that constitute the elements of a matrix or vector are denoted with the lower Greek letter that corresponds to

the letter used for the matrix of vector:

$$X = \begin{pmatrix} \chi_{0,0} & \chi_{0,1} & \cdots & \chi_{0,n-1} \\ \chi_{1,0} & \chi_{1,1} & \cdots & \chi_{1,n-1} \\ \vdots & \vdots & & \vdots \\ \chi_{m-1,0} & \chi_{m-1,1} & \cdots & \chi_{m-1,n-1} \end{pmatrix} \quad \text{and} \quad x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{m-1} \end{pmatrix}.$$

If you look carefully, you will notice the difference between x and χ. The latter is the lower case Greek letter "chi."

Remark 1.3.1 Since this course will discuss the computation $C := AB + C$, you will only need to remember the Greek letters α (alpha), β (beta), and γ (gamma).

Homework 1.3.1.1 Consider

$$A = \begin{pmatrix} -2 & -3 & -1 \\ 2 & 0 & 1 \\ 3 & -2 & 2 \end{pmatrix}.$$

Identify

- $\alpha_{1,2} =$
- $a_0 =$
- $\widetilde{a}_2^T =$

Solution.

- $\alpha_{1,2} = 1$.
- $a_0 = \begin{pmatrix} -2 \\ 2 \\ 3 \end{pmatrix}$.
- $\widetilde{a}_2^T = \begin{pmatrix} 3 & -2 & 2 \end{pmatrix}$.

1.3.2 The dot product (inner product)

YouTube: https://www.youtube.com/watch?v=62GO5uCCaGU

Given two vectors x and y of size n

$$x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{n-1} \end{pmatrix} \quad \text{and} \quad y = \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{n-1} \end{pmatrix},$$

their dot product is given by

$$x^T y = \chi_0\psi_0 + \chi_1\psi_1 + \cdots + \chi_{n-1}\psi_{n-1} = \sum_{i=0}^{n-1} \chi_i\psi_i.$$

The notation $x^T y$ comes from the fact that the dot product also equals the result of multiplying $1 \times n$ matrix x^T times $n \times 1$ matrix y.

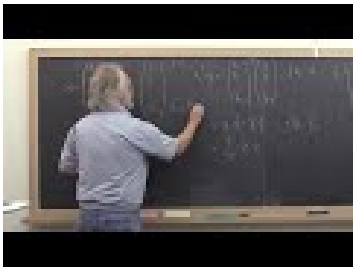
Example 1.3.2

$$\begin{pmatrix} -1 \\ 2 \\ 3 \end{pmatrix}^T \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}$$
$=$ $<$ View first vector as a 3×1 matrix and transpose it $>$
$$\begin{pmatrix} -1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}$$
$=$ $<$ multiply 1×3 matrix times 3×1 matrix $>$
$(-1) \times (1) + (2) \times (0) + \times(3) \times (-1)$
$=$
-4.

□

Pseudo-code for computing $\gamma := x^T y + \gamma$ is given by

for $i := 0, \ldots, n-1$
  $\gamma := \chi_i\psi_i + \gamma$
end



YouTube: https://www.youtube.com/watch?v=f1_shSplMiY

Homework 1.3.2.1 Consider

$$C = \underbrace{\begin{pmatrix} 1 & -2 & 2 \\ -1 & 1 & 3 \\ -2 & 2 & -1 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} -2 & 1 \\ 1 & 3 \\ -1 & 2 \end{pmatrix}}_{B}.$$

Compute $\gamma_{2,1}$.

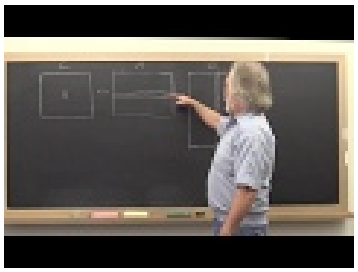
Solution.

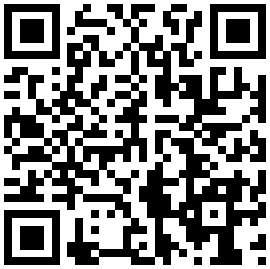
$$\gamma_{2,1} = \begin{pmatrix} -2 & 2 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix} = (-2) \times (1) + (2) \times (3) + (-1) \times (2) = 2.$$

The point we are trying to make here is that $\gamma_{2,1}$ is computed as the dot product of the third row of A (the row indexed with 2) with the last column of B (the column indexed with 1):

$$\gamma_{2,1} = \begin{pmatrix} -2 \\ 2 \\ -1 \end{pmatrix}^T \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix} = (-2) \times (1) + (2) \times (3) + (-1) \times (2) = 2.$$

Importantly, if you think back about how matrices are stored in column-major order, marching through memory from one element to the next element in the last row of A, as you access $\tilde{a}_2^T$, requires "striding" through memory.





YouTube: https://www.youtube.com/watch?v=QCjJA5jqnr0

A routine, coded in C, that computes $x^T y + \gamma$ where x and y are stored at location x with stride incx and location y with stride incy, respectively, and γ is stored at location gamma, is given by

```
#define chi( i ) x[ (i)*incx ]   // map chi( i ) to array x
#define psi( i ) y[ (i)*incy ]   // map psi( i ) to array y

void Dots( int n, double *x, int incx, double *y, int incy, double *gamma )
{
  for ( int i=0; i<n; i++ )
    *gamma += chi( i ) * psi( i );
}
```

in Assignments/Week1/C/Dots.c. Here stride refers to the number of items in memory between the stored components of the vector. In particular, the stride

when accessing a row of a matrix is ldA when the matrix is stored in column-major order with leading dimension ldA, as illustrated in Figure 1.2.2.

Homework 1.3.2.2 Consider the following double precision values stored in sequential memory starting at address A

$$\begin{array}{rcr} & & 3 \\ \texttt{A[0]} & \longrightarrow & -1 \\ & & 0 \\ & & 2 \\ & & 3 \\ & & -1 \\ & & 1 \\ & & 4 \\ & & -1 \\ & & 3 \\ & & 5 \end{array}$$

and assume gamma initially contains the value 1. After executing
`Dots( 3, &A[1], 4, &A[3], 1, &gamma )`

What are the contents of variable gamma?

Answer. 5

Solution. The first 3 indicates that the vectors are of size 3. The &A[1], 4 identifies the vector starting at address &A[1] with stride 4:

$$\begin{array}{rcr} & & -1 \\ \texttt{A[1]} & \longrightarrow & 0 \\ & & 2 \\ & & 3 \\ & & -1 \\ \texttt{A[1+4]} & \longrightarrow & 1 \\ & & 4 \\ & & -1 \\ & & 3 \\ \texttt{A[1+8]} & \longrightarrow & 5 \end{array}$$

and the &A[3], 1 identifies the vector starting at address &A[3] with stride 1:

$$\begin{array}{rcr} & & -1 \\ & & 0 \\ & & 2 \\ \texttt{A[3]} & \longrightarrow & 3 \\ \texttt{A[3+1]} & \longrightarrow & -1 \\ \texttt{A[3+2]} & \longrightarrow & 1 \\ & & 4 \\ & & -1 \\ & & 3 \\ & & 5 \end{array}$$

Hence, upon completion, gamma contains

$$1 + \begin{pmatrix} 0 \\ 1 \\ 5 \end{pmatrix}^T \begin{pmatrix} 3 \\ -1 \\ 1 \end{pmatrix} = 1 + (0) \times (3) + (1) \times (-1) + (5) \times (1) = 5.$$

1.3.3 Matrix-vector multiplication via dot products

Homework 1.3.3.1 Compute $\begin{pmatrix} 2 & 2 & -1 & 2 \\ 2 & 1 & 0 & -2 \\ -2 & -2 & 2 & 2 \end{pmatrix} \begin{pmatrix} 2 \\ -1 \\ 0 \\ -1 \end{pmatrix} =$

Solution.

$$\left(\begin{array}{cccc} 2 & 2 & -1 & 2 \\ \hline 2 & 1 & 0 & -2 \\ \hline -2 & -2 & 2 & 2 \end{array}\right) \begin{pmatrix} 2 \\ -1 \\ 0 \\ -1 \end{pmatrix} = \left(\begin{array}{c} \begin{pmatrix} 2 & 2 & -1 & 2 \end{pmatrix} \begin{pmatrix} 2 \\ -1 \\ 0 \\ -1 \end{pmatrix} \\ \hline \begin{pmatrix} 2 & 1 & 0 & -2 \end{pmatrix} \begin{pmatrix} 2 \\ -1 \\ 0 \\ -1 \end{pmatrix} \\ \hline \begin{pmatrix} -2 & -2 & 2 & 2 \end{pmatrix} \begin{pmatrix} 2 \\ -1 \\ 0 \\ -1 \end{pmatrix} \end{array}\right) = \begin{pmatrix} 0 \\ 5 \\ -4 \end{pmatrix}$$

(Here we really care more about exposing the dot products of rows with the vector than the final answer.)

YouTube: https://www.youtube.com/watch?v=lknmeBwMp4o

Consider the matrix-vector multiplication

$$y := Ax + y.$$

The way one is usually taught to compute this operation is that each element of y, ψ_i, is updated with the dot product of the corresponding row of A, $\widetilde{a}_i^T$,

with vector x. With our notation, we can describe this as

$$\left(\begin{array}{c}\psi_0\\ \hline \psi_1\\ \hline \vdots\\ \hline \psi_{m-1}\end{array}\right) := \left(\begin{array}{c}\widetilde{a}_0^T\\ \hline \widetilde{a}_1^T\\ \hline \vdots\\ \hline \widetilde{a}_{m-1}^T\end{array}\right) x + \left(\begin{array}{c}\psi_0\\ \hline \psi_1\\ \hline \vdots\\ \hline \psi_{m-1}\end{array}\right)$$
$$= \left(\begin{array}{c}\widetilde{a}_0^T x\\ \hline \widetilde{a}_1^T x\\ \hline \vdots\\ \hline \widetilde{a}_{m-1}^T x\end{array}\right) + \left(\begin{array}{c}\psi_0\\ \hline \psi_1\\ \hline \vdots\\ \hline \psi_{m-1}\end{array}\right) = \left(\begin{array}{c}\widetilde{a}_0^T x + \psi_0\\ \hline \widetilde{a}_1^T x + \psi_1\\ \hline \vdots\\ \hline \widetilde{a}_{m-1}^T x + \psi_{m-1}\end{array}\right).$$

Remark 1.3.3 The way we remember how to do the partitioned matrix-vector multiplication

$$\left(\begin{array}{c}\widetilde{a}_0^T\\ \hline \widetilde{a}_1^T\\ \hline \vdots\\ \hline \widetilde{a}_{m-1}^T\end{array}\right) x = \left(\begin{array}{c}\widetilde{a}_0^T x\\ \hline \widetilde{a}_1^T x\\ \hline \vdots\\ \hline \widetilde{a}_{m-1}^T x\end{array}\right)$$

is to replace the m with a specific integer, 3, and each symbol with a number:

$$\left(\begin{array}{c}1\\ \hline -2\\ \hline 3\end{array}\right)(-1).$$

If you view the first as a 3×1 matrix and the second as a 1×1 matrix, the result is

$$\left(\begin{array}{c}1\\ \hline -2\\ \hline 3\end{array}\right)(-1) = \left(\begin{array}{c}(1) \times (-1)\\ \hline (-2) \times (-1)\\ \hline (3) \times (-1)\end{array}\right). \tag{1.3.1}$$

You should then notice that multiplication with the partitioned matrix and vector works the same way:

$$\left(\begin{array}{c}\widetilde{a}_0^T\\ \hline \widetilde{a}_1^T\\ \hline \vdots\\ \hline \widetilde{a}_{m-1}^T\end{array}\right) x = \left(\begin{array}{c}(\widetilde{a}_0^T) \times (x)\\ \hline (\widetilde{a}_1^T) \times (x)\\ \hline \vdots\\ \hline (\widetilde{a}_{m-1}^T) \times (x)\end{array}\right) \tag{1.3.2}$$

with the change that in (1.3.1) the multiplication with numbers commutes while in (1.3.1) the multiplication does not (in general) commute:

$$(-2) \times (-1) = (-1) \times (-2)$$

but, in general,

$$\widetilde{a}_i^T x \neq x \widetilde{a}_i^T.$$

Indeed, $\widetilde{a}_i^T x$ is a dot product (inner product) while we will later be reminded that $x\widetilde{a}_i^T$ is an outerproduct of vectors $\widetilde{a}_i$ and x.

If we then expose the individual elements of A and y we get

$$\begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{pmatrix} := \begin{pmatrix} \alpha_{0,0} \chi_0 + \alpha_{0,1} \chi_1 + \cdots \alpha_{0,n-1} \chi_{n-1} + \psi_0 \\ \alpha_{1,0} \chi_0 + \alpha_{1,1} \chi_1 + \cdots \alpha_{1,n-1} \chi_{n-1} + \psi_1 \\ \vdots \\ \alpha_{m-1,0} \chi_0 + \alpha_{m-1,1} \chi_1 + \cdots \alpha_{m-1,n-1} \chi_{n-1} + \psi_{m-1} \end{pmatrix}$$

This discussion explains the IJ loop for computing $y := Ax + y$:

for $i := 0, \ldots, m-1$
  for $j := 0, \ldots, n-1$
    $\psi_i := \alpha_{i,j} \chi_j + \psi_i$
  end
end

$\psi_i := \widetilde{a}_i^T x + \psi_i$

where we notice that again the i index ranges over the rows of the matrix and j index ranges over the columns of the matrix.

Homework 1.3.3.2 In directory Assignments/Week1/C complete the implementation of matrix-vector multiplication in terms of dot operations

```
#define alpha( i,j ) A[ (j)*ldA + i ]   // map alpha( i,j ) to array A
#define chi( i )  x[ (i)*incx ]            // map chi( i )  to array x
#define psi( i )  y[ (i)*incy ]            // map psi( i )  to array y

void Dots( int, const double *, int, const double *, int, double * );

void MyGemv( int m, int n, double *A, int ldA,
           double *x, int incx, double *y, int incy )
{
  for ( int i=0; i<m; i++ )
    Dots(   ,      ,     ,     ,    ,      );
}
```

in file Assignments/Week1/C/Gemv_I_Dots.c. You can test it by executing

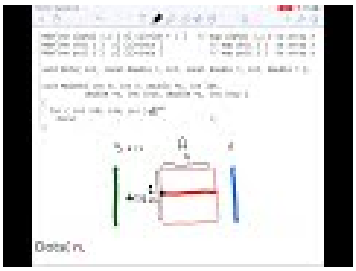
```
make I_Dots
```

Upon completion, this should print

```
It appears all is well
```

Admittedly, this is a very incomplete test of the routine. However, it doesn't play much of a role in the course, so we choose to be sloppy.

Solution. Assignments/Week1/Answers/Gemv_I_Dots.c.



YouTube: https://www.youtube.com/watch?v=fGS9P_fgo6Q

Remark 1.3.4 The file name Gemv_I_Dots.c can be decoded as follows:

- The Gemv stands for "GEneral Matrix-Vector multiplication."
- The I indicates a loop indexed by i (a loop over rows of the matrix).
- The Dots indicates that the operation performed in the body of the loop is a dot product (hence the Dot) with the result added to a scalar (hence the s).

1.3.4 The axpy operation

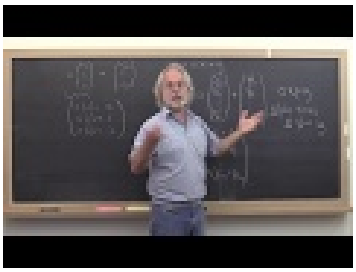
Homework 1.3.4.1 Compute

$$(-2)\begin{pmatrix} 2 \\ -1 \\ 3 \end{pmatrix} + \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix}$$

Solution.

$$(-2)\begin{pmatrix} 2 \\ -1 \\ 3 \end{pmatrix} + \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} (-2)\times(2) \\ (-2)\times(-1) \\ (-2)\times(3) \end{pmatrix} + \begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix}$$

$$= \begin{pmatrix} (-2)\times(2)+2 \\ (-2)\times(-1)+1 \\ (-2)\times(3)+0 \end{pmatrix} = \begin{pmatrix} -2 \\ 3 \\ -6 \end{pmatrix}$$



YouTube: https://www.youtube.com/watch?v=O-D5l3OVstE

Given a scalar, α, and two vectors, x and y, of size n with elements

$$x = \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{n-1} \end{pmatrix} \quad \text{and} \quad y = \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{n-1} \end{pmatrix},$$

the scaled vector addition (axpy) operation is given by

$$y := \alpha x + y$$

which in terms of the elements of the vectors equals

$$\begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{n-1} \end{pmatrix} := \alpha \begin{pmatrix} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{n-1} \end{pmatrix} + \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{n-1} \end{pmatrix}$$
$$= \begin{pmatrix} \alpha\chi_0 \\ \alpha\chi_1 \\ \vdots \\ \alpha\chi_{n-1} \end{pmatrix} + \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{n-1} \end{pmatrix} = \begin{pmatrix} \alpha\chi_0 + \psi_0 \\ \alpha\chi_1 + \psi_1 \\ \vdots \\ \alpha\chi_{n-1} + \psi_{n-1} \end{pmatrix}.$$

The name axpy comes from the fact that in Fortran 77 only six characters and numbers could be used to designate the names of variables and functions. The operation $\alpha x + y$ can be read out loud as "scalar alpha times x plus y" which yields the acronym axpy.

Homework 1.3.4.2 An outline for a routine that implements the axpy operation is given by

```
#define chi( i ) x[ (i)*incx ]   // map chi( i ) to array x
#define psi( i ) y[ (i)*incy ]   // map psi( i ) to array y

void Axpy( int n, double alpha, double *x, int incx, double *y, int incy )
{
  for ( int i=0; i<n; i++ )
    psi(i)  +=
}
```

in file Assignments/Week1/C/Axpy.c.

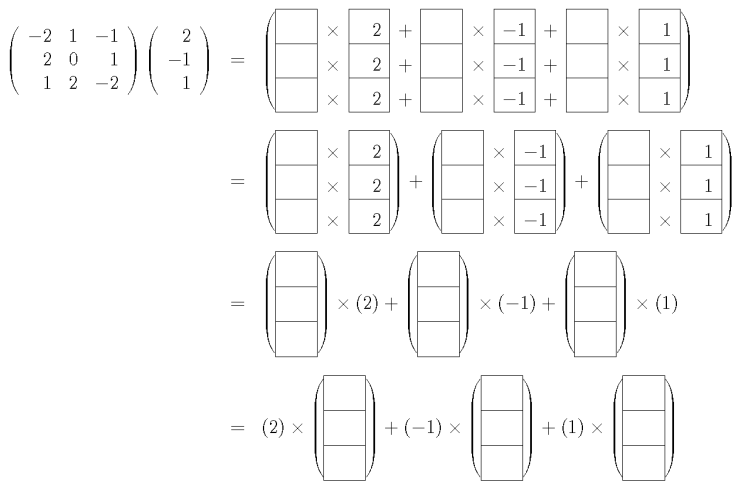
Complete the routine and test the implementation by using it in the next unit.

Solution. Assignments/Week1/Answers/Axpy.c

1.3.5 Matrix-vector multiplication via axpy operations

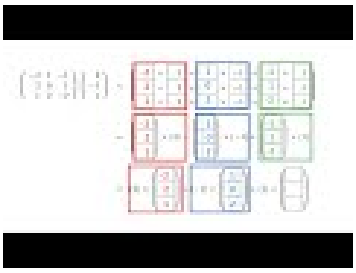
Homework 1.3.5.1 Complete

Complete the following:



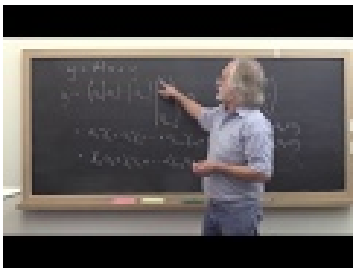
Click here to enlarge.

Solution.



YouTube: https://www.youtube.com/watch?v=_yx9c5OyITI

We now discuss how matrix-vector multiplication can be cast in terms of axpy operations.



YouTube: https://www.youtube.com/watch?v=1uryOws5320

We capture the insights from this last exercise: Partition $m \times n$ matrix A by columns and x by individual elements.

$$
\begin{aligned}
y &:= \left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{n-1} \end{array} \right) \left(\begin{array}{c} \chi_0 \\ \hline \chi_1 \\ \hline \vdots \\ \hline \chi_{n-1} \end{array} \right) + y \\
&= \chi_0 a_0 + \chi_1 a_1 + \cdots + \chi_{n-1} a_{n-1} + y.
\end{aligned}
$$

If we then expose the individual elements of A and y we get

$$
\begin{array}{l}
\begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{pmatrix} \\
\quad := \chi_0 \begin{pmatrix} \alpha_{0,0} \\ \alpha_{1,0} \\ \vdots \\ \alpha_{m-1,0} \end{pmatrix} + \chi_1 \begin{pmatrix} \alpha_{0,1} \\ \alpha_{1,1} \\ \vdots \\ \alpha_{m-1,1} \end{pmatrix} + \cdots + \chi_{n-1} \begin{pmatrix} \alpha_{0,n-1} \\ \alpha_{1,n-1} \\ \vdots \\ \alpha_{m-1,n-1} \end{pmatrix} + \begin{pmatrix} \psi_0 \\ \psi_1 \\ \vdots \\ \psi_{m-1} \end{pmatrix} \\
\quad = \begin{pmatrix} \chi_0 \alpha_{0,0} + \chi_1 \alpha_{0,1} + \cdots \chi_{n-1} \alpha_{0,n-1} + \psi_0 \\ \chi_0 \alpha_{1,0} + \chi_1 \alpha_{1,1} + \cdots \chi_{n-1} \alpha_{1,n-1} + \psi_1 \\ \vdots \\ \chi_0 \alpha_{m-1,0} + \chi_1 \alpha_{m-1,1} + \cdots \chi_{n-1} \alpha_{m-1,n-1} + \psi_{m-1} \end{pmatrix} \\
\quad = \begin{pmatrix} \alpha_{0,0} \chi_0 + \alpha_{0,1} \chi_1 + \cdots \alpha_{0,n-1} \chi_{n-1} + \psi_0 \\ \alpha_{1,0} \chi_0 + \alpha_{1,1} \chi_1 + \cdots \alpha_{1,n-1} \chi_{n-1} + \psi_1 \\ \vdots \\ \alpha_{m-1,0} \chi_0 + \alpha_{m-1,1} \chi_1 + \cdots \alpha_{m-1,n-1} \chi_{n-1} + \psi_{m-1} \end{pmatrix}.
\end{array}
$$

This discussion explains the JI loop for computing $y := Ax + y$:

for $j := 0, \ldots, n-1$
  for $i := 0, \ldots, m-1$
    $\psi_i := \alpha_{i,j} \chi_j + \psi_i$
  end
end

$\Big\}$ $y := \chi_j a_j + y$

What it also demonstrates is how matrix-vector multiplication can be implemented as a sequence of axpy operations.

Homework 1.3.5.2 In directory Assignments/Week1/C complete the implementation of matrix-vector multiplication in terms of axpy operations

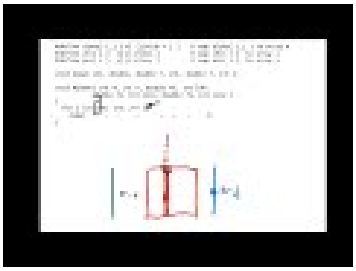
```
#define alpha( i,j ) A[ (j)*ldA + i ]   // map alpha( i,j ) to array A
#define chi( i )  x[ (i)*incx ]         // map chi( i )  to array x
#define psi( i )  y[ (i)*incy ]         // map psi( i )  to array y

void Axpy( int, double, double *, int, double *, int );

void MyGemv( int m, int n, double *A, int ldA,
             double *x, int incx, double *y, int incy )
{
  for ( int j=0; j<n; j++ )
    Axpy(      ,        ,      ,   ,   ,     );
}
```

in file Assignments/Week1/C/Gemv_J_Axpy.c. You can test your implementation by executing

```
make J_Axpy
```

Remember: you implemented the Axpy routine in the last unit, and we did not test it... So, if you get a wrong answer, the problem may be in that implementation.

Hint.



YouTube: https://www.youtube.com/watch?v=EYHkioY639c

Solution. Assignments/Week1/Answers/Gemv_J_Axpy.c.

Remark 1.3.5 The file name Gemv_J_Axpy.c can be decoded as follows:

- The Gemv stands for "GEneral Matrix-Vector multiplication."
- The J indicates a loop indexed by j (a loop over columns of the matrix).
- The Axpy indicates that the operation performed in the body of the loop is an axpy operation.

1.3.6 Matrix-matrix multiplication via matrix-vector multiplications

Homework 1.3.6.1 Fill in the blanks:

Complete the following:

$$\begin{pmatrix} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{pmatrix} \left(\begin{array}{cc|c|cc} & & -2 & & 0 \\ 1 & 2 & -1 & 3 & \end{array} \right) + \left(\begin{array}{cc|c|cc} & & 3 & & 0 \\ & & -2 & & 1 \\ & & 1 & & -1 \end{array} \right) \left| \begin{array}{c} -4 \\ -3 \\ -2 \end{array} \right.$$

$$= \left(\begin{array}{cc|c|cc} \square \times 1 + \square \times 2 + 3 & & \square \times -2 + \square \times -1 + 0 & & \square \times 0 + \square \times 3 - 4 \\ \square \times 1 + \square \times 2 - 2 & & \square \times -2 + \square \times -1 + 1 & & \square \times 0 + \square \times 3 - 3 \\ \square \times 1 + \square \times 2 + 1 & & \square \times -2 + \square \times -1 - 1 & & \square \times 0 + \square \times 3 - 2 \end{array} \right)$$

$$= \left(\left(\begin{pmatrix} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{pmatrix} \begin{pmatrix} \square \\ \square \end{pmatrix} + \begin{pmatrix} 3 \\ -2 \\ 1 \end{pmatrix} \right| \left(\begin{pmatrix} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{pmatrix} \begin{pmatrix} \square \\ \square \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix} \right| \left(\begin{pmatrix} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{pmatrix} \begin{pmatrix} \square \\ \square \end{pmatrix} + \begin{pmatrix} -4 \\ -3 \\ -2 \end{pmatrix} \right) \right)$$

Click here to enlarge.

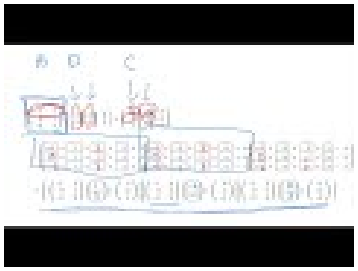
Solution.

$$\left(\begin{array}{cc} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{array}\right)\left(\begin{array}{c|c|c} 1 & -2 & 0 \\ 2 & -1 & 3 \end{array}\right)+\left(\begin{array}{c|c|c} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{array}\right)$$

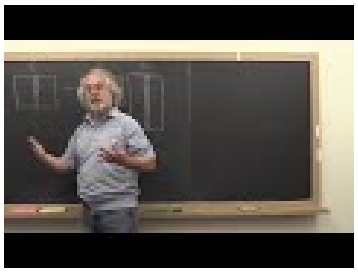
$$=\left(\begin{array}{c|c|c} \boxed{-1}\times\boxed{1}+\boxed{2}\times\boxed{2}+3 & \boxed{-1}\times\boxed{-2}+\boxed{2}\times\boxed{-1}+0 & \boxed{-1}\times\boxed{0}+\boxed{2}\times\boxed{3}-4 \\ \boxed{0}\times\boxed{1}+\boxed{1}\times\boxed{2}-2 & \boxed{0}\times\boxed{-2}+\boxed{1}\times\boxed{-1}+1 & \boxed{0}\times\boxed{0}+\boxed{1}\times\boxed{3}-3 \\ \boxed{-2}\times\boxed{1}+\boxed{3}\times\boxed{2}+1 & \boxed{-2}\times\boxed{-2}+\boxed{3}\times\boxed{-1}-1 & \boxed{-2}\times\boxed{0}+\boxed{3}\times\boxed{3}-2 \end{array}\right)$$

$$=\left(\begin{array}{c|c|c} \left(\begin{array}{cc} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{array}\right)\boxed{\begin{array}{c}1\\2\end{array}}+\left(\begin{array}{c} 3 \\ -2 \\ 1 \end{array}\right) & \left(\begin{array}{cc} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{array}\right)\boxed{\begin{array}{c}-2\\-1\end{array}}+\left(\begin{array}{c} 0 \\ 1 \\ -1 \end{array}\right) & \left(\begin{array}{cc} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{array}\right)\boxed{\begin{array}{c}0\\3\end{array}}+\left(\begin{array}{c} -4 \\ -3 \\ -2 \end{array}\right) \end{array}\right)$$

Click here to enlarge.



YouTube: https://www.youtube.com/watch?v=sXQ-QDne9uw

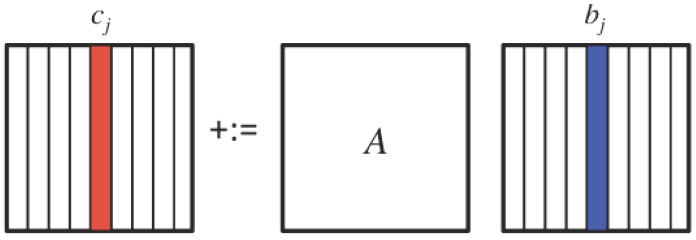


YouTube: https://www.youtube.com/watch?v=6-OReBSHuGc

Now that we are getting comfortable with partitioning matrices and vectors, we can view the six algorithms for $C := AB + C$ in a more layered fashion. If we partition C and B by columns, we find that

$$\begin{array}{rcl} \left(\begin{array}{c|c|c|c} c_0 & c_1 & \cdots & c_{n-1} \end{array}\right) & := & A\left(\begin{array}{c|c|c|c} b_0 & b_1 & \cdots & b_{n-1} \end{array}\right)+\left(\begin{array}{c|c|c|c} c_0 & c_1 & \cdots & c_{n-1} \end{array}\right) \\ & = & \left(\begin{array}{c|c|c|c} Ab_0+c_0 & Ab_1+c_1 & \cdots & Ab_{n-1}+c_{n-1} \end{array}\right) \end{array}$$

A picture that captures this is given by



This illustrates how the JIP and JPI algorithms can be layered as a loop around matrix-vector multiplications, which itself can be layered as a loop

around dot products or axpy operations:

for $j := 0, \ldots, n-1$

  for $p := 0, \ldots, k-1$

    for $i := 0, \ldots, m-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end

    $\underbrace{c_j := \beta_{p,j}a_p + c_j}_{\text{axpy}}$

  end

  $\underbrace{c_j := Ab_j + c_j}_{\text{mv mult}}$

end

and

for $j := 0, \ldots, n-1$

  for $i := 0, \ldots, m-1$

    for $p := 0, \ldots, k-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end

    $\gamma_{i,j} := \underbrace{\widetilde{a}_i^T b_j + \gamma_{i,j}}_{\text{dots}}$

  end

  $\underbrace{c_j := Ab_j + c_j}_{\text{mv mult}}$

end

Homework 1.3.6.2 Complete the code in Assignments/Week1/C/Gemm_J_Gemv.c. Test two versions:
`make J_Gemv_I_Dots`
`make J_Gemv_J_Axpy`

View the resulting performance by making the necessary changes to the Live Script in `Assignments/Week1/C/Plot_Outer_J.mlx`. (Alternatively, use the script in `Assignments/Week1/C/data/Plot_Outer_J_m.m`.)

Solution. `Assignments/Week1/Answers/Gemm_J_Gemv.c`.

Remark 1.3.6 The file name Gemm_J_Gemv.c can be decoded as follows:

- The Gemm stands for "GEneral Matrix-Matrix multiplication."
- The J indicates a loop indexed by j (a loop over columns of matrix C).
- The Gemv indicates that the operation performed in the body of the loop is matrix-vector multiplication.

Similarly, J_Gemv_I_Dots means that the implementation of matrix-matrix multiplication being compiled and executed consists of a loop indexed by j (a loop over the columns of matrix C) which itself calls a matrix-vector multiplication that is implemented as a loop indexed by i (over the rows of the matrix with which the matrix-vector multiplication is performed), with a dot product in the body of the loop.

Remark 1.3.7 Hopefully you are catching on to our naming convention. It is a bit adhoc and we are probably not completely consistent. They give some

hint as to what the implementation looks like.

Going forward, we refrain from further explaining the convention.

1.4 Layering Matrix-Matrix Multiplication: Alternatives

1.4.1 Rank-1 update (rank-1)

An operation that will become very important in future discussion and optimization of matrix-matrix multiplication is the rank-1 update:

$$A := xy^T + A.$$

Homework 1.4.1.1 Fill in the blanks:

$$\begin{pmatrix} 2 \\ -1 \\ 0 \end{pmatrix} \begin{pmatrix} -2 & 0 & 1 \end{pmatrix} + \left(\begin{array}{c|c|c} 2 & 2 & -1 \\ 2 & 1 & 0 \\ -2 & -2 & 2 \end{array}\right)$$

$$= \left(\begin{array}{c|c|c} \square \times \boxed{-2} + 2 & \square \times \boxed{0} + 2 & \square \times \boxed{1} - 1 \\ \square \times \boxed{-2} + 2 & \square \times \boxed{0} + 1 & \square \times \boxed{1} + 0 \\ \square \times \boxed{-2} - 2 & \square \times \boxed{0} - 2 & \square \times \boxed{1} + 2 \end{array}\right)$$

$$= \left(\begin{array}{c|c|c} \left(\begin{array}{c}\square\\\hline\square\\\hline\square\end{array}\right)(-2) + \begin{pmatrix} 2 \\ 2 \\ -2 \end{pmatrix} & \left(\begin{array}{c}\square\\\hline\square\\\hline\square\end{array}\right)(0) + \begin{pmatrix} 2 \\ 1 \\ -2 \end{pmatrix} & \left(\begin{array}{c}\square\\\hline\square\\\hline\square\end{array}\right)(1) + \begin{pmatrix} -1 \\ 0 \\ 2 \end{pmatrix} \end{array}\right)$$

$$= \left(\begin{array}{c|c|c} (-2)\left(\begin{array}{c}\square\\\hline\square\\\hline\square\end{array}\right) + \begin{pmatrix} 2 \\ 2 \\ -2 \end{pmatrix} & (0)\left(\begin{array}{c}\square\\\hline\square\\\hline\square\end{array}\right) + \begin{pmatrix} 2 \\ 1 \\ -2 \end{pmatrix} & (1)\left(\begin{array}{c}\square\\\hline\square\\\hline\square\end{array}\right) + \begin{pmatrix} -1 \\ 0 \\ 2 \end{pmatrix} \end{array}\right)$$

Click here to enlarge.

Solution.

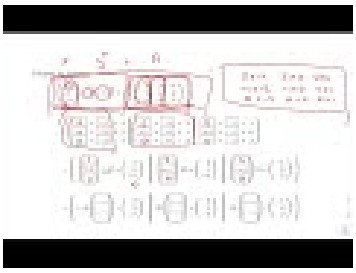
$$\begin{pmatrix} 2 \\ -1 \\ 0 \end{pmatrix} \left(\begin{array}{c|c|c} -2 & 0 & 1 \end{array} \right) + \left(\begin{array}{c|c|c} 2 & 2 & -1 \\ 2 & 1 & 0 \\ -2 & -2 & 2 \end{array} \right)$$

$$= \left(\begin{array}{c|c|c} \begin{array}{|c|}\hline 2 \\ \hline -1 \\ \hline 0 \\ \hline \end{array} \times \begin{array}{|c|}\hline -2 \\ \hline -2 \\ \hline -2 \\ \hline \end{array} + 2 & \begin{array}{|c|}\hline 2 \\ \hline -1 \\ \hline 0 \\ \hline \end{array} \times \begin{array}{|c|}\hline 0 \\ \hline 0 \\ \hline 0 \\ \hline \end{array} + 2 & \begin{array}{|c|}\hline 2 \\ \hline -1 \\ \hline 0 \\ \hline \end{array} \times \begin{array}{|c|}\hline 1 \\ \hline 1 \\ \hline 1 \\ \hline \end{array} - 1 \\ \begin{array}{|c|}\hline 2 \\ \hline -1 \\ \hline 0 \\ \hline \end{array} \times \begin{array}{|c|}\hline -2 \\ \hline -2 \\ \hline -2 \\ \hline \end{array} + 2 & \begin{array}{|c|}\hline 2 \\ \hline -1 \\ \hline 0 \\ \hline \end{array} \times \begin{array}{|c|}\hline 0 \\ \hline 0 \\ \hline 0 \\ \hline \end{array} + 1 & \begin{array}{|c|}\hline 2 \\ \hline -1 \\ \hline 0 \\ \hline \end{array} \times \begin{array}{|c|}\hline 1 \\ \hline 1 \\ \hline 1 \\ \hline \end{array} + 0 \\ \begin{array}{|c|}\hline 2 \\ \hline -1 \\ \hline 0 \\ \hline \end{array} \times \begin{array}{|c|}\hline -2 \\ \hline -2 \\ \hline -2 \\ \hline \end{array} - 2 & \begin{array}{|c|}\hline 2 \\ \hline -1 \\ \hline 0 \\ \hline \end{array} \times \begin{array}{|c|}\hline 0 \\ \hline 0 \\ \hline 0 \\ \hline \end{array} - 2 & \begin{array}{|c|}\hline 2 \\ \hline -1 \\ \hline 0 \\ \hline \end{array} \times \begin{array}{|c|}\hline 1 \\ \hline 1 \\ \hline 1 \\ \hline \end{array} + 2 \end{array} \right)$$

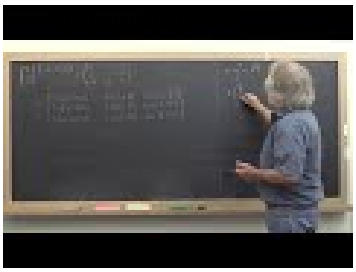
$$= \left(\begin{array}{c|c|c} \left(\begin{array}{|c|}\hline 2 \\ \hline -1 \\ \hline 0 \\ \hline \end{array} \right) (-2) + \begin{pmatrix} 2 \\ 2 \\ -2 \end{pmatrix} & \left(\begin{array}{|c|}\hline 2 \\ \hline -1 \\ \hline 0 \\ \hline \end{array} \right) (0) + \begin{pmatrix} 2 \\ 1 \\ -2 \end{pmatrix} & \left(\begin{array}{|c|}\hline 2 \\ \hline -1 \\ \hline 0 \\ \hline \end{array} \right) (1) + \begin{pmatrix} -1 \\ 0 \\ 2 \end{pmatrix} \end{array} \right)$$

$$= \left(\begin{array}{c|c|c} (-2) \left(\begin{array}{|c|}\hline 2 \\ \hline -1 \\ \hline 0 \\ \hline \end{array} \right) + \begin{pmatrix} 2 \\ 2 \\ -2 \end{pmatrix} & (0) \left(\begin{array}{|c|}\hline 2 \\ \hline -1 \\ \hline 0 \\ \hline \end{array} \right) + \begin{pmatrix} 2 \\ 1 \\ -2 \end{pmatrix} & (1) \left(\begin{array}{|c|}\hline 2 \\ \hline -1 \\ \hline 0 \\ \hline \end{array} \right) + \begin{pmatrix} -1 \\ 0 \\ 2 \end{pmatrix} \end{array} \right)$$

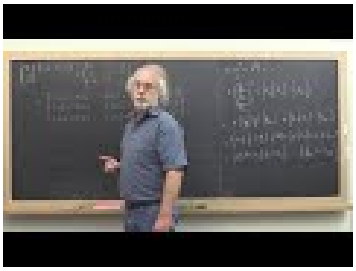
Click here to enlarge.



YouTube: https://www.youtube.com/watch?v=5YxKCoUEdyM



YouTube: https://www.youtube.com/watch?v=0Emae-CWkNo



YouTube: https://www.youtube.com/watch?v=FsdMId76ejE

More generally,

$$A := xy^T + A$$

is computed as

$$\left(\begin{array}{c|c|c|c} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,k-1} \\ \hline \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,k-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,k-1} \end{array}\right) :=$$

$$\left(\begin{array}{c} \chi_0 \\ \hline \chi_1 \\ \hline \vdots \\ \hline \chi_{m-1} \end{array}\right) \left(\begin{array}{c|c|c|c} \psi_0 & \psi_1 & \cdots & \psi_{n-1} \end{array}\right) + \left(\begin{array}{c|c|c|c} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,n-1} \\ \hline \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,n-1} \end{array}\right)$$

$$= \left(\begin{array}{c|c|c|c} \chi_0\psi_0 + \alpha_{0,0} & \chi_0\psi_1 + \alpha_{0,1} & \cdots & \chi_0\psi_{n-1} + \alpha_{0,n-1} \\ \hline \chi_1\psi_0 + \alpha_{1,0} & \chi_1\psi_1 + \alpha_{1,1} & \cdots & \chi_1\psi_{n-1} + \alpha_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \chi_{m-1}\psi_0 + \alpha_{m-1,0} & \chi_{m-1}\psi_1 + \alpha_{m-1,1} & \cdots & \chi_{m-1}\psi_{n-1} + \alpha_{m-1,n-1} \end{array}\right).$$

so that each entry $\alpha_{i,j}$ is updated by

$$\alpha_{i,j} := \chi_i\psi_j + \alpha_{i,j}.$$

If we now focus on the columns in this last matrix, we find that

$$\left(\begin{array}{c|c|c|c} \chi_0\psi_0 + \alpha_{0,0} & \chi_0\psi_1 + \alpha_{0,1} & \cdots & \chi_0\psi_{n-1} + \alpha_{0,n-1} \\ \hline \chi_1\psi_0 + \alpha_{1,0} & \chi_1\psi_1 + \alpha_{1,1} & \cdots & \chi_1\psi_{n-1} + \alpha_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline \chi_{m-1}\psi_0 + \alpha_{m-1,0} & \chi_{m-1}\psi_1 + \alpha_{m-1,1} & \cdots & \chi_{m-1}\psi_{n-1} + \alpha_{m-1,n-1} \end{array}\right)$$

$$= \left(\begin{array}{c|c|c|c} \left(\begin{array}{c} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{m-1} \end{array}\right)\psi_0 + \left(\begin{array}{c} \alpha_{0,0} \\ \alpha_{1,0} \\ \vdots \\ \alpha_{m-1,0} \end{array}\right) & \cdots & \left(\begin{array}{c} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{m-1} \end{array}\right)\psi_{n-1} + \left(\begin{array}{c} \alpha_{0,n-1} \\ \alpha_{1,n-1} \\ \vdots \\ \alpha_{m-1,n-1} \end{array}\right) \end{array}\right)$$

$$= \left(\begin{array}{c|c|c|c} \psi_0\left(\begin{array}{c} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{m-1} \end{array}\right) + \left(\begin{array}{c} \alpha_{0,0} \\ \alpha_{1,0} \\ \vdots \\ \alpha_{m-1,0} \end{array}\right) & \cdots & \psi_{n-1}\left(\begin{array}{c} \chi_0 \\ \chi_1 \\ \vdots \\ \chi_{m-1} \end{array}\right) + \left(\begin{array}{c} \alpha_{0,n-1} \\ \alpha_{1,n-1} \\ \vdots \\ \alpha_{m-1,n-1} \end{array}\right) \end{array}\right).$$

What this illustrates is that we could have partitioned A by columns and y by elements to find that

$$\begin{array}{l} \left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{n-1} \end{array}\right) \\ \quad := x\left(\begin{array}{c|c|c|c} \psi_0 & \psi_1 & \cdots & \psi_{n-1} \end{array}\right) + \left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{n-1} \end{array}\right) \\ \quad = \left(\begin{array}{c|c|c|c} x\psi_0 + a_0 & x\psi_1 + a_1 & \cdots & x\psi_{n-1} + a_{n-1} \end{array}\right) \\ \quad = \left(\begin{array}{c|c|c|c} \psi_0x + a_0 & \psi_1x + a_1 & \cdots & \psi_{n-1}x + a_{n-1} \end{array}\right). \end{array}$$

This discussion explains the JI loop ordering for computing $A := xy^T + A$:

for $j := 0, \ldots, n-1$
  for $i := 0, \ldots, m-1$
    $\alpha_{i,j} := \chi_i\psi_j + \alpha_{i,j}$
  end
  $a_j := \psi_j x + a_j$
end

It demonstrates is how the rank-1 operation can be implemented as a sequence of axpy operations.

Homework 1.4.1.2 In Assignments/Week1/C/Ger_J_Axpy.c complete the implementation of rank-1 in terms of axpy operations. You can test it by executing
`make Ger_J_Axpy`

Solution. Assignments/Week1/Answers/Ger_J_Axpy.c

Homework 1.4.1.3 Fill in the blanks:

$$\begin{pmatrix} 2 \\ -1 \\ 0 \end{pmatrix} \begin{pmatrix} -2 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 2 & 2 & -1 \\ 2 & 1 & 0 \\ -2 & -2 & 2 \end{pmatrix}$$

$$= \begin{pmatrix} \boxed{2} \times \boxed{} + 2 & \boxed{2} \times \boxed{} + 2 & \boxed{2} \times \boxed{} - 1 \\ \boxed{-1} \times \boxed{} + 2 & \boxed{-1} \times \boxed{} + 1 & \boxed{-1} \times \boxed{} + 0 \\ \boxed{0} \times \boxed{} - 2 & \boxed{0} \times \boxed{} - 2 & \boxed{0} \times \boxed{} + 2 \end{pmatrix}$$

$$= \begin{pmatrix} (2) \begin{pmatrix} \boxed{} & \boxed{} & \boxed{} \end{pmatrix} + (2 & 2 & -1) \\ (-1) \begin{pmatrix} \boxed{} & \boxed{} & \boxed{} \end{pmatrix} + (2 & 1 & 0) \\ (0) \begin{pmatrix} \boxed{} & \boxed{} & \boxed{} \end{pmatrix} + (-2 & -2 & 2) \end{pmatrix}$$

Click here to enlarge.

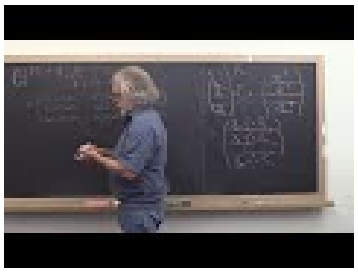
Solution.

$$\begin{pmatrix} 2 \\ -1 \\ 0 \end{pmatrix} \begin{pmatrix} -2 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 2 & 2 & -1 \\ 2 & 1 & 0 \\ -2 & -2 & 2 \end{pmatrix}$$

$$= \begin{pmatrix} \boxed{2} \times \boxed{-2} + 2 & \boxed{2} \times \boxed{0} + 2 & \boxed{2} \times \boxed{1} - 1 \\ \boxed{-1} \times \boxed{-2} + 2 & \boxed{-1} \times \boxed{0} + 1 & \boxed{-1} \times \boxed{1} + 0 \\ \boxed{0} \times \boxed{-2} - 2 & \boxed{0} \times \boxed{0} - 2 & \boxed{0} \times \boxed{1} + 2 \end{pmatrix}$$

$$= \begin{pmatrix} (2) \begin{pmatrix} \boxed{-2} & \boxed{0} & \boxed{1} \end{pmatrix} + (2 & 2 & -1) \\ (-1) \begin{pmatrix} \boxed{-2} & \boxed{0} & \boxed{1} \end{pmatrix} + (2 & 1 & 0) \\ (0) \begin{pmatrix} \boxed{-2} & \boxed{0} & \boxed{1} \end{pmatrix} + (-2 & -2 & 2) \end{pmatrix}$$

Click here to enlarge.



YouTube: https://www.youtube.com/watch?v=r_l93XMfJ1Q

The last homework suggests that there is also an IJ loop ordering that can be explained by partitioning A by rows and x by elements:

$$\begin{pmatrix} \widetilde{a}_0^T \\ \hline \widetilde{a}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T \end{pmatrix} := \begin{pmatrix} \chi_0 \\ \hline \chi_1 \\ \hline \vdots \\ \hline \chi_{m-1} \end{pmatrix} y^T + \begin{pmatrix} \widetilde{a}_0^T \\ \hline \widetilde{a}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T \end{pmatrix} = \begin{pmatrix} \chi_0 y^T + \widetilde{a}_0^T \\ \hline \chi_1 y^T + \widetilde{a}_1^T \\ \hline \vdots \\ \hline \chi_{m-1} y^T + \widetilde{a}_{m-1}^T \end{pmatrix}$$

leading to the algorithm

for $i := 0, \ldots, n-1$

  for $j := 0, \ldots, m-1$

    $\alpha_{i,j} := \chi_i \psi_j + \alpha_{i,j}$

  end

$\widetilde{a}_i^T := \chi_i y^T + \widetilde{a}_i^T$

end

and corresponding implementation.

Homework 1.4.1.4 In Assignments/Week1/C/Ger_I_Axpy.c complete the implementation of rank-1 in terms of axpy operations (by rows). You can test it by executing

```
make Ger_I_Axpy
```

Solution. Assignments/Week1/Answers/Ger_I_Axpy.c

1.4.2 Matrix-matrix multiplication via rank-1 updates

Homework 1.4.2.1 Fill in the blanks:

$$\left(\begin{array}{c|c} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{array}\right)\left(\begin{array}{ccc} 1 & -2 & 0 \\ \hline 2 & -1 & 3 \end{array}\right)+\begin{pmatrix} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{pmatrix}$$

$$=\begin{pmatrix} \square\times\boxed{1}+\square\times\boxed{2} & \square\times\boxed{-2}+\square\times\boxed{-1} & \square\times\boxed{0}+\square\times\boxed{3} \\ \square\times\boxed{1}+\square\times\boxed{2} & \square\times\boxed{-2}+\square\times\boxed{-1} & \square\times\boxed{0}+\square\times\boxed{3} \\ \square\times\boxed{1}+\square\times\boxed{2} & \square\times\boxed{-2}+\square\times\boxed{-1} & \square\times\boxed{0}+\square\times\boxed{3} \end{pmatrix}$$

$$+\begin{pmatrix} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{pmatrix}$$

$$=\begin{pmatrix} \square\times\boxed{1} & \square\times\boxed{-2} & \square\times\boxed{0} \\ \square\times\boxed{1} & \square\times\boxed{-2} & \square\times\boxed{0} \\ \square\times\boxed{1} & \square\times\boxed{-2} & \square\times\boxed{0} \end{pmatrix}+\begin{pmatrix} \square\times\boxed{2} & \square\times\boxed{-1} & \square\times\boxed{3} \\ \square\times\boxed{2} & \square\times\boxed{-1} & \square\times\boxed{3} \\ \square\times\boxed{2} & \square\times\boxed{-1} & \square\times\boxed{3} \end{pmatrix}$$

$$+\begin{pmatrix} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{pmatrix}$$

$$=\begin{pmatrix} \square \\ \square \\ \square \end{pmatrix}\begin{pmatrix} 1 & -2 & 0 \end{pmatrix}+\begin{pmatrix} \square \\ \square \\ \square \end{pmatrix}\begin{pmatrix} 2 & -1 & 3 \end{pmatrix}+\begin{pmatrix} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{pmatrix}$$

Click here to enlarge.

Solution.

$$\left(\begin{array}{c|c} -1 & 2 \\ 0 & 1 \\ -2 & 3 \end{array}\right)\left(\begin{array}{ccc} 1 & -2 & 0 \\ \hline 2 & -1 & 3 \end{array}\right)+\begin{pmatrix} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{pmatrix}$$

$$=\begin{pmatrix} \boxed{-1}\times\boxed{1}+\boxed{2}\times\boxed{2} & \boxed{-1}\times\boxed{-2}+\boxed{2}\times\boxed{-1} & \boxed{-1}\times\boxed{0}+\boxed{2}\times\boxed{3} \\ \boxed{0}\times\boxed{1}+\boxed{1}\times\boxed{2} & \boxed{0}\times\boxed{-2}+\boxed{1}\times\boxed{-1} & \boxed{0}\times\boxed{0}+\boxed{1}\times\boxed{3} \\ \boxed{-2}\times\boxed{1}+\boxed{3}\times\boxed{2} & \boxed{-2}\times\boxed{-2}+\boxed{3}\times\boxed{-1} & \boxed{-2}\times\boxed{0}+\boxed{3}\times\boxed{3} \end{pmatrix}$$

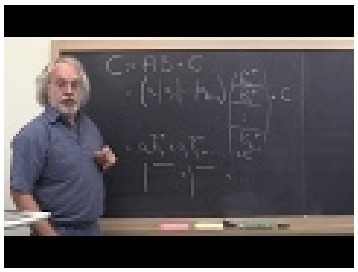
$$+\begin{pmatrix} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{pmatrix}$$

$$=\begin{pmatrix} \boxed{-1}\times\boxed{1} & \boxed{-1}\times\boxed{-2} & \boxed{-1}\times\boxed{0} \\ \boxed{0}\times\boxed{1} & \boxed{0}\times\boxed{-2} & \boxed{0}\times\boxed{0} \\ \boxed{-2}\times\boxed{1} & \boxed{-2}\times\boxed{-2} & \boxed{-2}\times\boxed{0} \end{pmatrix}+\begin{pmatrix} \boxed{2}\times\boxed{2} & \boxed{2}\times\boxed{-1} & \boxed{2}\times\boxed{3} \\ \boxed{1}\times\boxed{2} & \boxed{1}\times\boxed{-1} & \boxed{1}\times\boxed{3} \\ \boxed{3}\times\boxed{2} & \boxed{3}\times\boxed{-1} & \boxed{3}\times\boxed{3} \end{pmatrix}$$

$$+\begin{pmatrix} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{pmatrix}$$

$$=\begin{pmatrix} \boxed{-1} \\ \boxed{0} \\ \boxed{-2} \end{pmatrix}\begin{pmatrix} 1 & -2 & 0 \end{pmatrix}+\begin{pmatrix} \boxed{2} \\ \boxed{1} \\ \boxed{3} \end{pmatrix}\begin{pmatrix} 2 & -1 & 3 \end{pmatrix}+\begin{pmatrix} 3 & 0 & -4 \\ -2 & 1 & -3 \\ 1 & -1 & -2 \end{pmatrix}$$

Click here to enlarge.

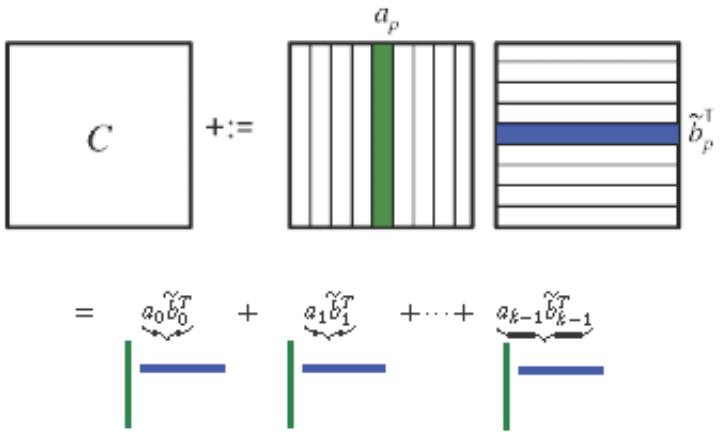


YouTube: https://www.youtube.com/watch?v=YxAiEueEfNk

Let us partition A by columns and B by rows, so that

$$\begin{aligned} C &:= \left(a_0 \mid a_1 \mid \cdots \mid a_{k-1} \right) \left(\begin{array}{c} \widetilde{b}_0^T \\ \hline \widetilde{b}_1^T \\ \hline \vdots \\ \hline \widetilde{b}_{k-1}^T \end{array} \right) + C \\ &= a_0 \widetilde{b}_0^T + a_1 \widetilde{b}_1^T + \cdots + a_{k-1} \widetilde{b}_{k-1}^T + C \end{aligned}$$

A picture that captures this is given by



This illustrates how the PJI and PIJ algorithms can be viewed as a loop around rank-1 updates:

for $p := 0, \ldots, k-1$

  for $j := 0, \ldots, n-1$

    for $i := 0, \ldots, m-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end

  $\Big\}$ $\underbrace{c_j := \beta_{p,j} a_p + c_j}_{\text{axpy}}$ $\Big\}$ $\underbrace{C := a_p \widetilde{b}_p^T + C}_{\text{rank-1 update}}$

end

end

and

for $p := 0, \ldots, k-1$

  for $i := 0, \ldots, m-1$

    for $j := 0, \ldots, n-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end

  $\left.\begin{array}{l}\\ \\ \\ \end{array}\right\}$ $\underbrace{\widetilde{c}_i^T := \alpha_{i,p}\widetilde{b}_p^T + \widetilde{c}_i^T}_{\text{axpy}}$ $\left.\begin{array}{l}\\ \\ \\ \end{array}\right\}$ $\underbrace{C := a_p\widetilde{b}_p^T + C}_{\text{rank-1 update}}$

  end

end

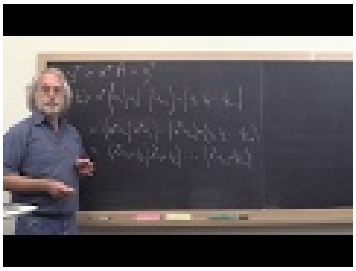
Homework 1.4.2.2 Complete the code in Assignments/Week1/C/Gemm_P_Ger.c. Test two versions:
`make P_Ger_J_Axpy`
`make P_Ger_I_Axpy`

View the resulting performance by making the necessary changes to the Live Script in `Assignments/Week1/C/Plot_Outer_P.mlx`. (Alternatively, use the script in `Assignments/Week1/C/data/Plot_Outer_P_m.m`.)

Solution.

- `Assignments/Week1/Answers/Gemm_P_Ger.c`.

1.4.3 Row-times-matrix multiplication



YouTube: `https://www.youtube.com/watch?v=eonBpl9XfAc`

An operation that is closely related to matrix-vector multiplication is the multipication of a row times a matrix, which in the setting of this course updates a row vector:

$$y^T := x^T A + y^T.$$

If we partition A by columns and y^T by elements, we get

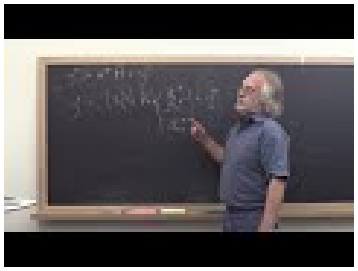
$$\begin{array}{l}\left(\begin{array}{c|c|c|c} \psi_0 & \psi_1 & \cdots & \psi_{n-1} \end{array}\right) := x^T \left(\begin{array}{c|c|c|c} a_0 & a_1 & \cdots & a_{n-1} \end{array}\right) + \left(\begin{array}{c|c|c|c} \psi_0 & \psi_1 & \cdots & \psi_{n-1} \end{array}\right) \\ \quad = \left(\begin{array}{c|c|c|c} x^T a_0 + \psi_0 & x^T a_1 + \psi_1 & \cdots & x^T a_{n-1} + \psi_{n-1} \end{array}\right).\end{array}$$

This can be implemented as a loop:

for $j := 0, \ldots, n-1$

$$\left.\begin{array}{l} \textbf{for } i := 0, \ldots, m-1 \\ \quad \psi_j := \chi_i \alpha_{i,j} + \psi_j \\ \textbf{end} \end{array}\right\} \quad \underbrace{\psi_j := x^T a_j + \psi_j}_{\text{dot}}$$

end



YouTube: https://www.youtube.com/watch?v=2VWSXkg1kGk

Alternatively, if we partition A by rows and x^T by elements, we get

$$\begin{aligned} y^T &:= \left(\begin{array}{c|c|c|c} \chi_0 & \chi_1 & \cdots & \chi_{m-1} \end{array} \right) \left(\begin{array}{c} \widetilde{a}_0^T \\ \hline \widetilde{a}_1^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T \end{array} \right) + y^T \\ &= \chi_0 \widetilde{a}_0^T + \chi_1 \widetilde{a}_1^T + \cdots + \chi_{m-1} \widetilde{a}_{m-1}^T + y^T. \end{aligned}$$

This can be implemented as a loop:

for $i := 0, \ldots, m-1$

$$\left.\begin{array}{l} \textbf{for } j := 0, \ldots, n-1 \\ \quad \psi_j := \chi_i \alpha_{i,j} + \psi_j \\ \textbf{end} \end{array}\right\} \quad \underbrace{y^T := \chi_i \widetilde{a}_i^T + y^T}_{\text{axpy}}$$

end

There is an alternative way of looking at this operation:

$$y^T := x^T A + y^T$$

is equivalent to

$$(y^T)^T := (x^T A)^T + (y^T)^T$$

and hence

$$y := A^T x + y.$$

Thus, this operation is equivalent to matrix-vector multiplication with the transpose of the matrix.

Remark 1.4.1 Since this operation does not play a role in our further discussions, we do not include exercises related to it.

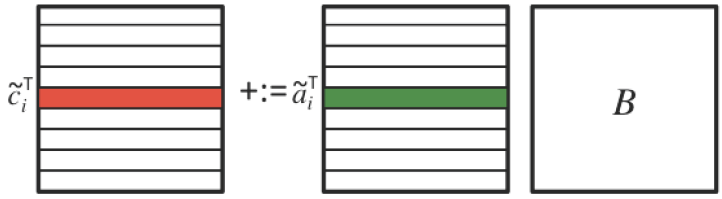
1.4.4 Matrix-matrix multiplication via row-times-matrix multiplications

YouTube: https://www.youtube.com/watch?v=4YhQ42fxevY

Finally, let us partition C and A by rows so that

$$\begin{pmatrix} \widetilde{c}_0^T \\ \widetilde{c}_1^T \\ \vdots \\ \widetilde{c}_{m-1}^T \end{pmatrix} := \begin{pmatrix} \widetilde{a}_0^T \\ \widetilde{a}_1^T \\ \vdots \\ \widetilde{a}_{m-1}^T \end{pmatrix} B + \begin{pmatrix} \widetilde{c}_0^T \\ \widetilde{c}_1^T \\ \vdots \\ \widetilde{c}_{m-1}^T \end{pmatrix} = \begin{pmatrix} \widetilde{a}_0^T B + \widetilde{c}_0^T \\ \widetilde{a}_1^T B + \widetilde{c}_1^T \\ \vdots \\ \widetilde{a}_{m-1}^T B + \widetilde{c}_{m-1}^T \end{pmatrix}$$

A picture that captures this is given by



This illustrates how the IJP and IPJ algorithms can be viewed as a loop around the updating of a row of C with the product of the corresponding row of A times matrix B:

for $i := 0, \ldots, m-1$

  for $j := 0, \ldots, n-1$

    for $p := 0, \ldots, k-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end

  $\underbrace{\widetilde{\gamma}_{i,j} := \widetilde{a}_i^T b_j + \widetilde{\gamma}_{i,j}}_{\text{dot}}$

  end

$\underbrace{\widetilde{c}_i^T := \widetilde{a}_i^T B + \widetilde{c}_i^T}_{\text{row-matrix mult}}$

end

and

for $i := 0, \ldots, m-1$

for $p := 0, \ldots, k-1$

for $j := 0, \ldots, n-1$
$\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
end

$\underbrace{\widetilde{c}_i^T := \alpha_{i,p}\widetilde{b}_p^T + \widetilde{c}_i^T}_{\text{axpy}}$

$\underbrace{\widetilde{c}_i^T := \widetilde{a}_i^T B + \widetilde{c}_i^T}_{\text{row-matrix mult}}$

end

end

The problem with implementing the above algorithms is that Gemv_I_Dots and Gemv_J_Axpy implement $y := Ax + y$ rather than $y^T := x^T A + y^T$. Obviously, you could create new routines for this new operation. We will get back to this in the "Additional Exercises" section of this chapter.

1.5 Enrichments

1.5.1 The Basic Linear Algebra Subprograms

Linear algebra operations are fundamental to computational science. In the 1970s, when vector supercomputers reigned supreme, it was recognized that if applications and software libraries are written in terms of a standardized interface to routines that implement operations with vectors, and vendors of computers provide high-performance instantiations for that interface, then applications would attain portable high performance across different computer platforms. This observation yielded the original Basic Linear Algebra Subprograms (BLAS) interface [17] for Fortran 77, which are now referred to as the level-1 BLAS. The interface was expanded in the 1980s to encompass matrix-vector operations (level-2 BLAS) [6] and matrix-matrix operations (level-3 BLAS) [5].

You should become familiar with the BLAS. Here are some resources:

An overview of the BLAS and how they are used to achieve portable high performance is given in the article [28]:

- Robert van de Geijn and Kazushige Goto, BLAS (Basic Linear Algebra Subprograms), Encyclopedia of Parallel Computing, Part 2, pp. 157-164, 2011. If you don't have access, you may want to read an advanced draft.

If you use the BLAS, you should cite one or more of the original papers (as well as the implementation that you use):

- C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, Basic Linear Algebra Subprograms for Fortran Usage, ACM Transactions on Mathematical Software, Vol. 5, No. 3, pp. 308-323, Sept. 1979.

- Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson, An Extended Set of {FORTRAN} Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software, Vol. 14, No. 1, pp. 1-17, March 1988.

- Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff, A Set of Level 3 Basic Linear Algebra Subprograms, ACM Transactions on Mathematical Software, Vol. 16, No. 1, pp. 1-17, March 1990.

A handy reference guide to the BLAS:

- Basic Linear Algebra Subprograms: A Quick Reference Guide. http://www.netlib.org/blas/blasqr.pdf.

There are a number of implementations of the BLAS available for various architectures:

- A reference implementation in Fortran is available from http://www.netlib.org/blas/.

  This is an unoptimized implementation: It provides the functionality without high performance.

- The current recommended high-performance open-source implementation of the BLAS is provided by the BLAS-like Library Instantiation Software (BLIS) discussed in Unit 1.5.2. The techniques you learn in this course underlie the implementation in BLIS.

- Different vendors provide their own high-performance implementations:

  - Intel provides optimized BLAS as part of their Math Kernel Library (MKL): https://software.intel.com/en-us/mkl.
  - AMD's open-source BLAS for their CPUs can be found at https://developer.amd.com/amd-cpu-libraries/blas-library/. Their implementation is based on BLIS.

    AMD also has a BLAS library for its GPUs: https://github.com/ROCmSoftwarePlatform/rocBLAS.
  - Arm provides optimized BLAS as part of their Arm Performance Library https://developer.arm.com/products/software-development-tools/hpc/arm-performance-libraries.
  - IBM provides optimized BLAS as part of their Engineering and Scientific Subroutine Library (ESSL): https://www.ibm.com/support/knowledgecenter/en/SSFHY8/essl_welcome.html.
  - Cray provides optimized BLAS as part of their Cray Scientific Libraries (LibSci) https://www.cray.com/sites/default/files/SB-Cray-Programming-Environm pdf.
  - For their GPU accelerators, NVIDIA provides the cuBLAS https://developer.nvidia.com/cublas.

1.5.2 The BLAS-like Library Instantiation Software (BLIS)

BLIS provides the currently recommended open-source implementation of the BLAS for traditional CPUs. It is the version used in this course. Its extensive documentation (and source) can be found at https://github.com/flame/blis/.

The techniques you learn in this course underlie the implementation of matrix-matrix multiplication in BLIS.

In addition to supporting the traditional BLAS interface, BLIS also exposes two more interfaces:

- The BLIS Typed API, which is BLAS-like, but provides more functionalility and what we believe to be a more flexible interface for the C and C++ programming languages.
- The BLIS Object-Based API, which elegantly hides many of the details that are exposed in the BLAS and Typed API interfaces.

BLIS is described in the paper

- Field G. Van Zee and Robert A. van de Geijn, BLIS: A Framework for Rapidly Instantiating BLAS Functionality, ACM Journal on Mathematical Software, Vol. 41, No. 3, June 2015. You can access this article for free by visiting the Science of High-Performance Computing group publication webpage (http://shpc.ices.utexas.edu/publications.html) and clicking on the title of Journal Article 39.

Many additional related papers can be found by visiting the BLIS GitHub repository (https://github.com/flame/blis) or the Science of High-Performance Computing group publication webpage (http://shpc.ices.utexas.edu/publications.html).

1.5.3 Counting flops

Floating point multiplies or adds are examples of floating point operation (flops). What we have noticed is that for all of our computations (dots, axpy, gemv, ger, and gemm) every floating point multiply is paired with a floating point add into a fused multiply-add (FMA).

Determining how many floating point operations are required for the different operations is relatively straight forward: If x and y are of size n, then

- $\gamma := x^T y + \gamma = \chi_0\psi_0 + \chi_1\psi_1 + \cdots + \chi_{n-1}\psi_{n-1} + \gamma$ requires n FMAs and hence $2n$ flops.
- $y := \alpha x + y$ requires n FMAs (one per pair of elements, one from x and one from y) and hence $2n$ flops.

Similarly, it is pretty easy to establish that if A is $m \times n$, then

- $y := Ax + y$ requires mn FMAs and hence $2mn$ flops.

  n axpy operations each of size m for $n \times 2m$ flops or m \dots\ operations each of size n for $m \times 2n$ flops.
- $A := xy^T + A$ required mn FMAs and hence $2mn$ flops.

  n axpy operations each of size m for $n \times 2m$ flops or m axpy operations each of size n for $m \times 2n$ flops.

Finally, if C is $m \times n$, A is $m \times k$, and B is $k \times n$, then $C := AB + C$ requires $2mnk$ flops. We can estiblish this by recognizing that if C is updated one column at a time, this takes n matrix-vector multiplication operations each with a matrix of size $m \times k$, for a total of $n \times 2mk$. Alternatively, if C is updated with rank-1 updates, then we get $k \times 2mn$.

When we run experiments, we tend to execute with matrices that are $n \times n$, where n ranges from small to large. Thus, the total operations required equal

$$\sum_{n=\mathtt{n_{first}}}^{\mathtt{n_{last}}} 2n^3 \text{ flops},$$

where $n_{\mathtt{first}}$ is the first problem size, $n_{\mathtt{last}}$ the last, and the summation is in increments of $n_{\mathtt{inc}}$.

If $\mathtt{n_{last}} = \mathtt{n_{inc}} \times N$ and $n_{\mathtt{inc}} = n_{\mathtt{first}}$, then we get

$$\sum_{n=n_{\mathtt{first}}}^{n_{\mathtt{last}}} 2n^3 = \sum_{i=1}^{N} 2(i \times n_{\mathtt{inc}})^3 = 2n_{\mathtt{inc}}^3 \sum_{i=1}^{N} i^3.$$

A standard trick is to recognize that

$$\sum_{i=1}^{N} i^3 \approx \int_0^N x^3 dx = \frac{1}{4}N^4.$$

So,

$$\sum_{n=n_{\mathtt{first}}}^{n_{\mathtt{last}}} 2n^3 \approx \frac{1}{2}n_{\mathtt{inc}}^3 N^4 = \frac{1}{2}\frac{n_{\mathtt{inc}}^4 N^4}{n_{\mathtt{inc}}} = \frac{1}{2n_{\mathtt{inc}}} n_{\mathtt{last}}^4.$$

The important thing is that every time we double $n_{\mathtt{last}}$, we have to wait, approximately, sixteen times as long for our experiments to finish...

An interesting question is why we count flops rather than FMAs. By now, you have noticed we like to report the rate of computation in billions of floating point operations per second, GFLOPS. Now, if we counted FMAs rather than flops, the number that represents the rate of computation would be cut in half. For marketing purposes, bigger is better and hence flops are reported! Seriously!

1.6 Wrap Up

1.6.1 Additional exercises

We start with a few exercises that let you experience the BLIS typed API and the traditional BLAS interface.

Homework 1.6.1.1 (Optional). In Unit 1.3.3 and Unit 1.3.5, you wrote implementations of the routine

```
MyGemv( int m, int n, double *A, int ldA, double *x, int incx, double *y, int incy )
```

The BLIS typed API (Unit 1.5.2) includes the routine

```
bli_dgemv( trans_t transa, conj_t conjx, dim_t  m,  dim_t  n,
           double* alpha, double* a, inc_t rsa, inc_t csa,
```

```
                          double* x, inc_t incx,
          double* beta,   double* y, inc_t incy );
```

which supports the operations

$$\begin{array}{rcl} y & := & \alpha Ax + \beta y \\ y & := & \alpha A^T x + \beta y \end{array}$$

and other variations on matrix-vector multiplication.

To get experience with the BLIS interface, copy Assignments/Week1/C/Gemm_J_Gemv.c into Gemm_J_bli_dgemv.c and replace the call to MyGemv with a call to bli_dgemv instead. You will also want to include the header file blis.h,

```
#include "blis.h"
```

replace the call to MyGemv with a call to bli_dgemv and, if you wish, comment out or delete the now unused prototype for MyGemv. Test your implementation with

```
make J_bli_dgemv
```

You can then view the performance with Assignments/Week1/C/data/Plot_Outer_J.mlx.

Hint. Replace

```
MyGemv( m, k, A, ldA, &beta( 0, j ), 1, &gamma( 0,j ), 1 );
```

with

```
double d_one=1.0;
bli_dgemv( BLIS_NO_TRANSPOSE, BLIS_NO_CONJUGATE, m, k,
&d_one, A, 1, ldA, &beta( 0,j ), 1, &d_one, &gamma( 0,j ), 1 );
```

Note: the BLIS_NO_TRANSPOSE indicates we want to compute $y := \alpha Ax + \beta y$ rather than $y := \alpha A^T x + \beta y$. The BLIS_NO_CONJUGATE parameter indicates that the elements of x are not conjugated (something one may wish to do with complex valued vectors).

The ldA parameter in MyGemv now becomes two paramters: 1, ldA. The first indicates the stride in memory between elements in the same column by consecutive rows (which we refer to as the row stride) and the second refers to the stride in memory between elements in the same row and consecutive columns (which we refer to as the column stride).

Solution. Assignments/Week1/Answers/Gemm_J_bli_dgemv.c

Homework 1.6.1.2 (Optional). The traditional BLAS interface (Unit 1.5.1) includes the Fortran call

```
DGEMV( TRANSA, M, N, ALPHA, A, LDA, X, INCX, BETA, Y, INCY )
```

which also supports the operations

$$\begin{array}{rcl} y & := & \alpha Ax + \beta y \\ y & := & \alpha A^T x + \beta y \end{array}$$

and other variations on matrix-vector multiplication.

To get experience with the BLAS interface, copy Assignments/Week1/C/Gemm_J_Gemv.c (or Gemm_J_bli_dgemv) into Gemm_J_dgemv and replace the call to MyGemv with a call to dgemv. Some hints:

- In calling a Fortran routine from C, you will want to call the routine using lower case letters, and adding an underscore:

  ```
  dgemv_
  ```

- You will need to place a "prototype" for dgemv_ near the beginning of Gemm_J_dgemv:

  ```
  void dgemv_( char *, int *, int *, double *, double *, int *, double *, int *, do
  ```

- Fortran passes parameters by address. So,

  ```
  MyGemv( m, k, A, ldA, &beta( 0, j ), 1, &gamma( 0,j ), 1 );
  ```

  becomes

  ```
  {
    int i_one=1;
    double d_one=1.0;
    dgemv_( "No transpose", &m, &k, &d_one, A, &ldA, &beta( 0, j ), &i_one, &d_one,
  }
  ```

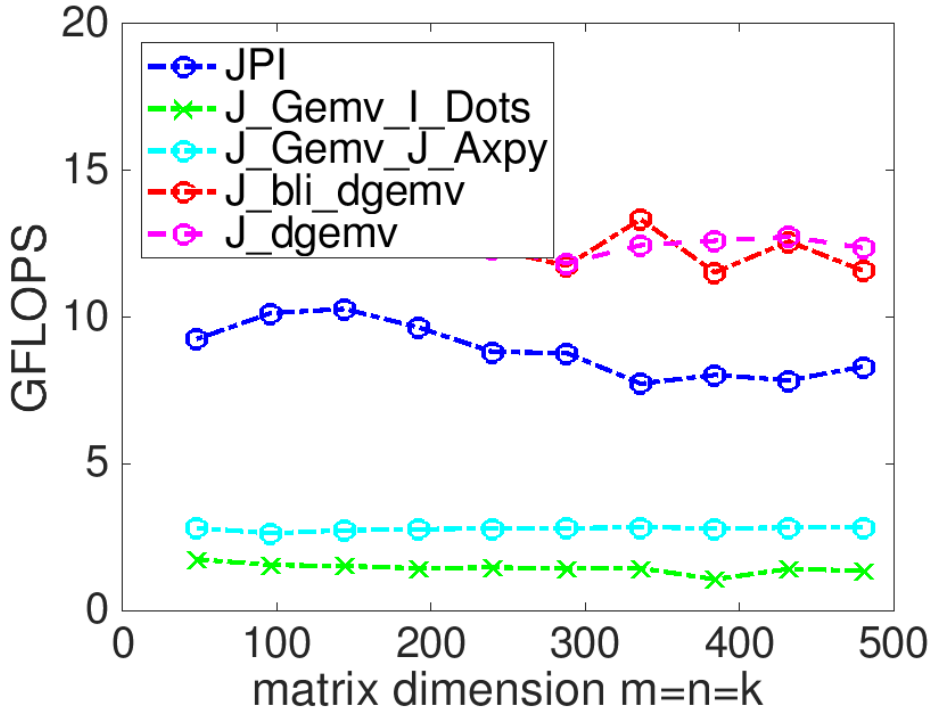
Test your implementation with

```
make J_dgemv
```

You can then view the performance with Assignments/Week1/C/data/Plot_Outer_J.mlx.

Solution. Assignments/Week1/Answers/Gemm_J_dgemv.c

On Robert's laptop, the last two exercises yield



Notice

- Linking to an optmized implementation (provided by BLIS) helps.
- The calls to bli_dgemv and dgemv are wrappers to the same implementations of matrix-vector multiplication within BLIS. The difference in performance that is observed should be considered "noise."
- It pays to link to a library that someone optimized.

Homework 1.6.1.3 (Optional). In Unit 1.4.1, You wrote the routine
`MyGer( int m, int n, double *x, int incx, double *y, int incy, double *A, int ldA )`

that compues a rank-1 update. The BLIS typed API (Unit 1.5.2) includes the routine

```
bli_dger( conj_t conjx, conj_t conjy, dim_t  m,  dim_t  n,
          double* alpha, double* x, inc_t incx,  double* y, inc_t incy,
          double* A, inc_t rsa,  inc_t csa );
```

which supports the operation

$$A := \alpha x y^T + A$$

and other variations on rank-1 update.

To get experience with the BLIS interface, copy Assignments/Week1/C/Gemm_P_Ger.c into Gemm_P_bli_dger and replace the call to MyGer with a call to bli_dger. Test your implementation with

```
make P_bli_dger
```

You can then view the performance with Assignments/Week1/C/data/Plot_Outer_P.mlx.

Hint. Replace

```
MyGer( m, n, &alpha( 0, p ), 1, &beta( p,0 ), ldB, C, ldC );
```

with

```
double d_one=1.0;
bli_dger( BLIS_NO_CONJUGATE, BLIS_NO_CONJUGATE, m, n,
&d_one, &alpha( 0, p ), 1, &beta( p,0 ), ldB, C, 1, ldC );
```

The BLIS_NO_CONJUGATE parameters indicate that the elements of x and y are not conjugated (something one may wish to do with complex valued vectors).

The ldC parameter in MyGer now becomes two paramters: 1, ldC. The first indicates the stride in memory between elements in the same column by consecutive rows (which we refer to as the row stride) and the second refers to the stride in memory between elements in the same row and consecutive columns (which we refer to as the column stride).

Solution. Assignments/Week1/Answers/Gemm_P_bli_dger.c

Homework 1.6.1.4 (Optional). The traditional BLAS interface (Unit 1.5.1) includes the Fortran call

```
DGER( M, N, ALPHA, X, INCX, Y, INCY, A, LDA );
```

which also supports the operation

$$A := \alpha x y^T + A.$$

To get experience with the BLAS interface, copy Assignments/Week1/C/Gemm_P_Ger.c (or Gemm_P_bli_dger) into Gemm_P_dger and replace the call to MyGer with a call to dger. Test your implementation with

```
make P_dger
```

You can then view the performance with Assignments/Week1/C/data/???.

Hint. Replace

```
Ger( m, n, &alpha( 0,p ), 1, &beta( p,0 ), ldB, C, ldC );
```

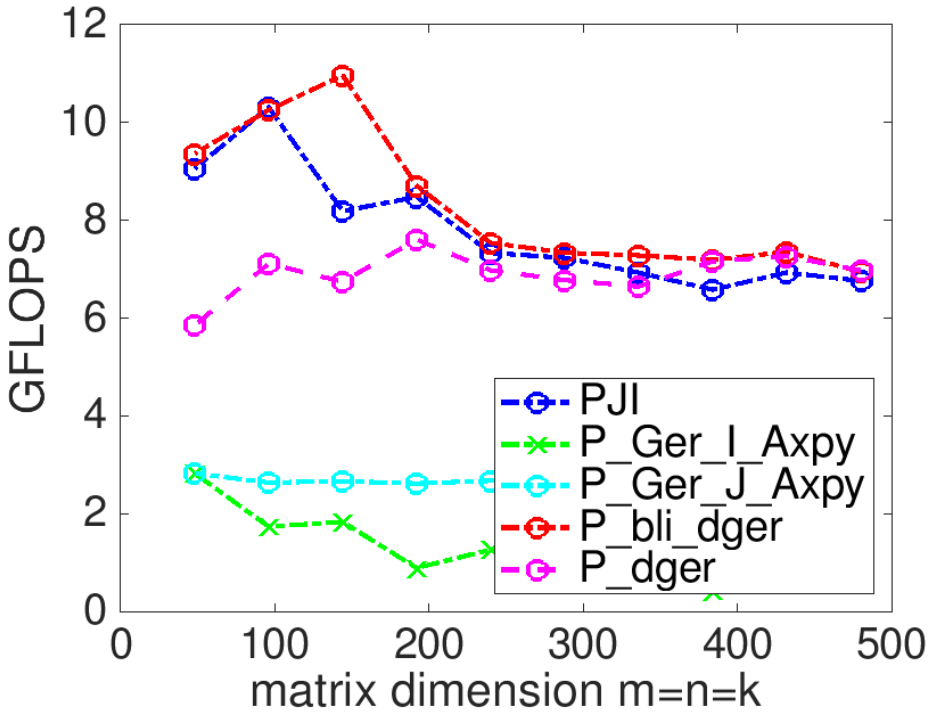
with

```
double d_one=1.0;
int i_one=1;
dger_( m, n,  &d_one, &alpha( 0,p ), &i_one, &beta( p,0 ), \amp;ldB, C, \amp;ldC );
```

The _ (underscore) allows C to call the Fortran routine. (This is how it works for most compilers, but not all.) The "No transpose" indicates we want to compute $y := \alpha Ax + \beta y$ rather than $y := \alpha A^T x + \beta y$. Fortran passes parameters by address, which is why d_one and i_one are passed as in this call.

Solution. Assignments/Week1/Answers/Gemm_P_dger.c

On Robert's laptop, the last two exercises yield



Notice

- Linking to an optmized implementation (provided by BLIS) does not seem to help (relative to PJI).
- The calls to bli_dger and dger are wrappers to the same implementations of matrix-vector multiplication within BLIS. The difference in performance that is observed should be considered "noise."
- It doesn't always pay to link to a library that someone optimized.

Next, we turn to computing $C := AB + C$ via a series of row-times-matrix multiplications.

Recall that the BLAS include the routine dgemv that computes

$$y := \alpha Ax + \beta y \quad \text{or} \quad y := \alpha A^T x + \beta y.$$

What we want is a routine that computes

$$y^T := x^T A + y^T.$$

What we remember from linear algebra is that if $A = B$ then $A^T = B^T$, that $(A + B)^T = A^T + B^T$, and that $(AB)^T = B^T A^T$. Thus, starting with the the equality

$$y^T = x^T A + y^T,$$

and transposing both sides, we get that

$$(y^T)^T = (x^T A + y^T)^T$$

which is equivalent to

$$y = (x^T A)^T + (y^T)^T$$

and finally

$$y = A^T x + y.$$

So, updating

$$y^T := x^T A + y^T$$

is equivalent to updating

$$y := A^T x + y.$$

It this all seems unfamiliar, you may want to look at Linear Algebra: Foundations to Frontiers

Now, if

- $m \times n$ matrix A is stored in array A with its leading dimension stored in variable ldA,
- m is stored in variable m and n is stored in variable n,
- vector x is stored in array x with its stride stored in variable incx,
- vector y is stored in array y with its stride stored in variable incy, and
- α and β are stored in variables alpha and beta, respectively,

then $y := \alpha A^T x + \beta y$ translates, from C, into the call

```
dgemv_( "Transpose", &m, &n, &alpha, A, &ldA, x, &incx, &beta, y, &incy );
```

Homework 1.6.1.5 In directory Assignments/Week1/C complete the code in file Gemm_I_dgemv.c that casts matrix-matrix multiplication in terms of the dgemv BLAS routine, but compute the result by rows. Compile and execute it with

```
make I_dgemv
```

View the resulting performance by making the necessary changes to the Live Script in `Assignments/Week1/C/Plot_All_Outer.mlx`. (Alternatively, use the script in `Assignments/Week1/C/data/Plot_All_Outer_m.mlx`.)

Solution. `Assignments/Week1/Answers/Gemm_I_dgemv.c`.

The BLIS native call that is similar to the BLAS dgemv routine in this setting translates to

```
bli_dgemv( BLIS_TRANSPOSE, BLIS_NO_CONJUGATE, m, n,
&alpha, A, ldA, 1, x, incx, &beta, y, incy );
```

Because of the two parameters after A that capture the stride between elements in a column (the row stride) and elements in rows (the column stride), one can alternatively swap these parameters:

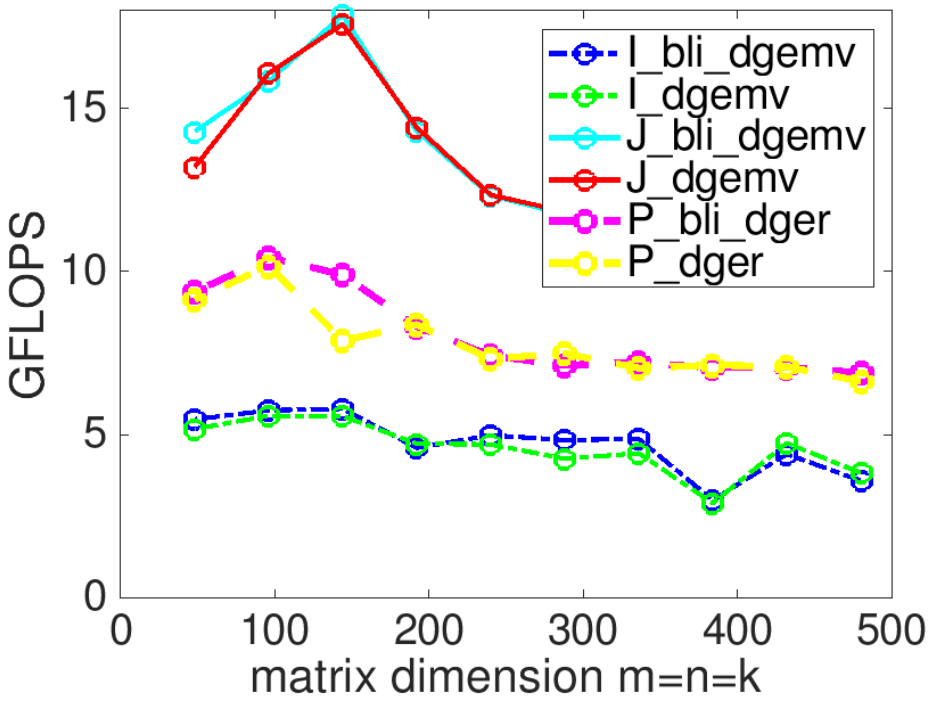
```
bli_dgemv( BLIS_NO_TRANSPOSE, BLIS_NO_CONJUGATE, m, n,
&alpha, A, ldA, 1, x, incx, &beta, y, incy );
```

Homework 1.6.1.6 In directory Assignments/Week1/C complete the code in file Gemm_I_bli_dgemv.c that casts matrix-matrix mulitplication in terms of the bli_dgemv BLIS routine. Compile and execute it with
`make I_bli_dgemv`

View the resulting performance by making the necessary changes to the Live Script in `Assignments/Week1/C/Plot_All_Outer.mlx`. (Alternatively, use the script in `Assignments/Week1/C/data/Plot_All_Outer_m.mlx`.)

Solution. `Assignments/Week1/Answers/Gemm_I_bli_dgemv.c`.

On Robert's laptop, the last two exercises yield



Notice

- Casting matrix-matrix multiplication in terms of matrix-vector multiplications attains the best performance. The reason is that as columns of C are computed, they can stay in cache, reducing the number of times elements of C have to be read and written from main memory.
- Accessing C and A by rows really gets in the way of performance.

1.6.2 Summary

The building blocks for matrix-matrix multiplication (and many matrix algorithms) are

- Vector-vector operations:
  - Dot: $x^T y$.

- Axpy: $y := \alpha x + y$.

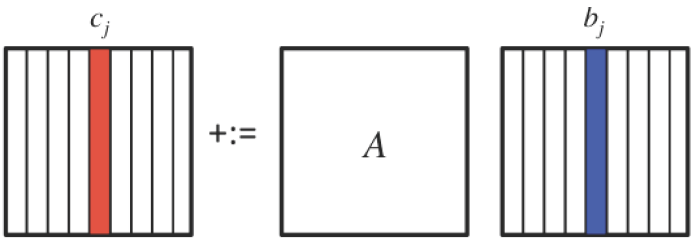
- Matrix-vector operations:
  - Matrix-vector multiplication: $y := Ax + y$ and $y := A^T x + y$.
  - Rank-1 update: $A := xy^T + A$.

Partitioning the matrices by rows and colums, with these operations, the matrix-matrix multiplication $C := AB+C$ can be described as one loop around

- multiple matrix-vector multiplications:

$$\left(\begin{array}{c|c|c} c_0 & \cdots & c_{n-1} \end{array} \right) := \left(\begin{array}{c|c|c} Ab_0 + c_0 & \cdots & Ab_{n-1} + c_{n-1} \end{array} \right)$$

illustrated by



.

This hides the two inner loops of the triple nested loop in the matrix-vector multiplication:

for $j := 0, \ldots, n-1$

  for $p := 0, \ldots, k-1$

    for $i := 0, \ldots, m-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end

  end
end

$\left.\begin{array}{l} \\ \\ \end{array}\right\}$ $\underbrace{c_j := \beta_{p,j} a_p + c_j}_{\text{axpy}}$ $\left.\begin{array}{l} \\ \\ \end{array}\right\}$ $\underbrace{c_j := Ab_j + c_j}_{\text{mv mult}}$

and

for $j := 0, \ldots, n-1$

  for $i := 0, \ldots, m-1$

    for $p := 0, \ldots, k-1$
      $\gamma_{i,j} := \underbrace{\alpha_{i,p}\beta_{p,j} + \gamma_{i,j}}_{\text{axpy}}$
    end

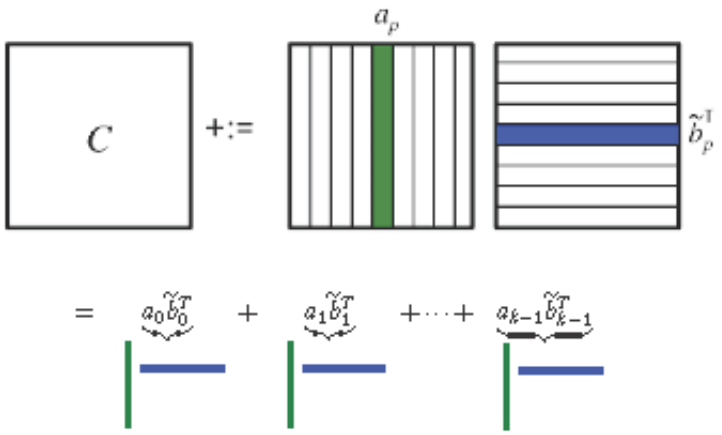
  end
end

$\left.\begin{array}{l} \\ \\ \end{array}\right\}$ $\gamma_{i,j} := \widetilde{a}_i^T b_j + \gamma_{i,j}$ $\left.\begin{array}{l} \\ \\ \end{array}\right\}$ $\underbrace{c_j := Ab_j + c_j}_{\text{mv mult}}$

- multiple rank-1 updates:

$$C := a_0\tilde{b}_0^T + a_1\tilde{b}_1^T + \cdots + a_{k-1}\tilde{b}_{k-1}^T + C$$

illustrated by



This hides the two inner loops of the triple nested loop in the rank-1 update:

for $p := 0, \ldots, k-1$

  for $j := 0, \ldots, n-1$

    for $i := 0, \ldots, m-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end

    $\underbrace{c_j := \beta_{p,j}a_p + c_j}_{\text{axpy}}$

  end

  $\underbrace{C := a_p\tilde{b}_p^T + C}_{\text{rank-1 update}}$

end

and

for $p := 0, \ldots, k-1$

  for $i := 0, \ldots, m-1$

    for $j := 0, \ldots, n-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end

    $\underbrace{\tilde{c}_i^T := \alpha_{i,p}\tilde{b}_p^T + \tilde{c}_i^T}_{\text{axpy}}$

  end

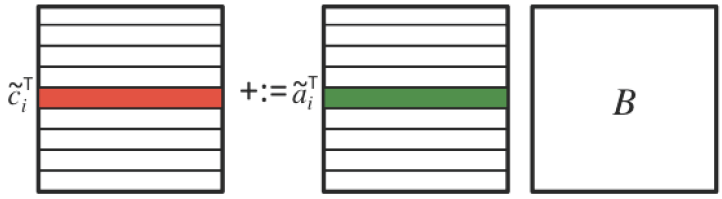
  $\underbrace{C := a_p\tilde{b}_p^T + C}_{\text{rank-1 update}}$

end

- multiple row times matrix multiplications:

$$\begin{pmatrix} \tilde{c}_0^T \\ \hline \tilde{c}_1^T \\ \hline \vdots \\ \hline \tilde{c}_{m-1}^T \end{pmatrix} := \begin{pmatrix} \tilde{a}_0^T B + \tilde{c}_0^T \\ \hline \tilde{a}_1^T B + \tilde{c}_1^T \\ \hline \vdots \\ \hline \tilde{a}_{m-1}^T B + \tilde{c}_{m-1}^T \end{pmatrix}$$

illustrated by



This hides the two inner loops of the triple nested loop in multiplications of rows with matrix B:

for $i := 0, \ldots, m-1$

  for $j := 0, \ldots, n-1$

    for $p := 0, \ldots, k-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end

  end
end

$\underbrace{\widetilde{\gamma}_{i,j} := \widetilde{a}_i^T b_j + \widetilde{\gamma}_{i,j}}_{\text{dot}}$

$\underbrace{\widetilde{c}_i^T := \widetilde{a}_i^T B + \widetilde{c}_i^T}_{\text{row-matrix mult}}$

and

for $i := 0, \ldots, m-1$

  for $p := 0, \ldots, k-1$

    for $j := 0, \ldots, n-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end

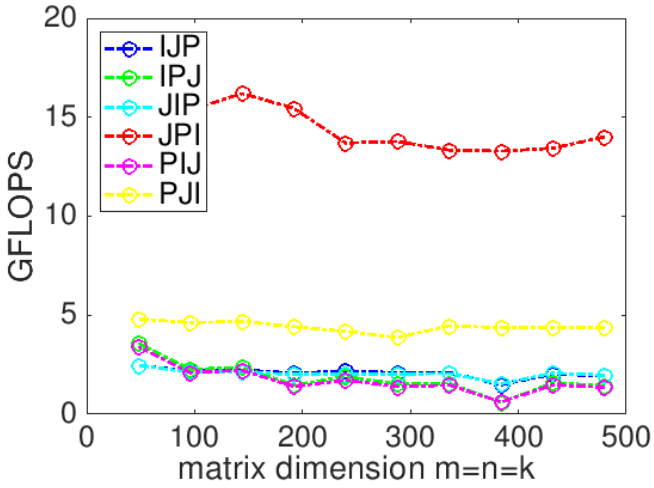
  end
end

$\underbrace{\widetilde{c}_i^T := \alpha_{i,p}\widetilde{b}_p^T + \widetilde{c}_i^T}_{\text{axpy}}$

$\underbrace{\widetilde{c}_i^T := \widetilde{a}_i^T B + \widetilde{c}_i^T}_{\text{row-matrix mult}}$

This summarizes all six loop orderings for matrix-matrix multiplication.

While it would be tempting to hope that a compiler would translate any of the six loop orderings into high-performance code, this is an optimistic dream. Our experiments shows that the order in which the loops are ordered has a nontrivial effect on performance:



We have observed that accessing matrices by columns, so that the most frequently loaded data is contiguous in memory, yields better performance.

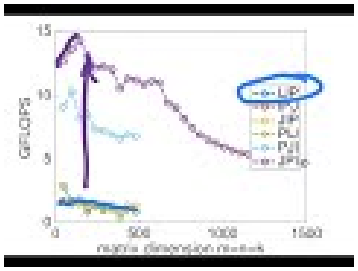
Those who did the additional exercises in Unit 1.6.1 found out that casting compution in terms of matrix-vector or rank-1 update operations in the BLIS typed API or the BLAS interface, when linked to an optimized implementation of those interfaces, yields better performance.

Week 2

Start Your Engines

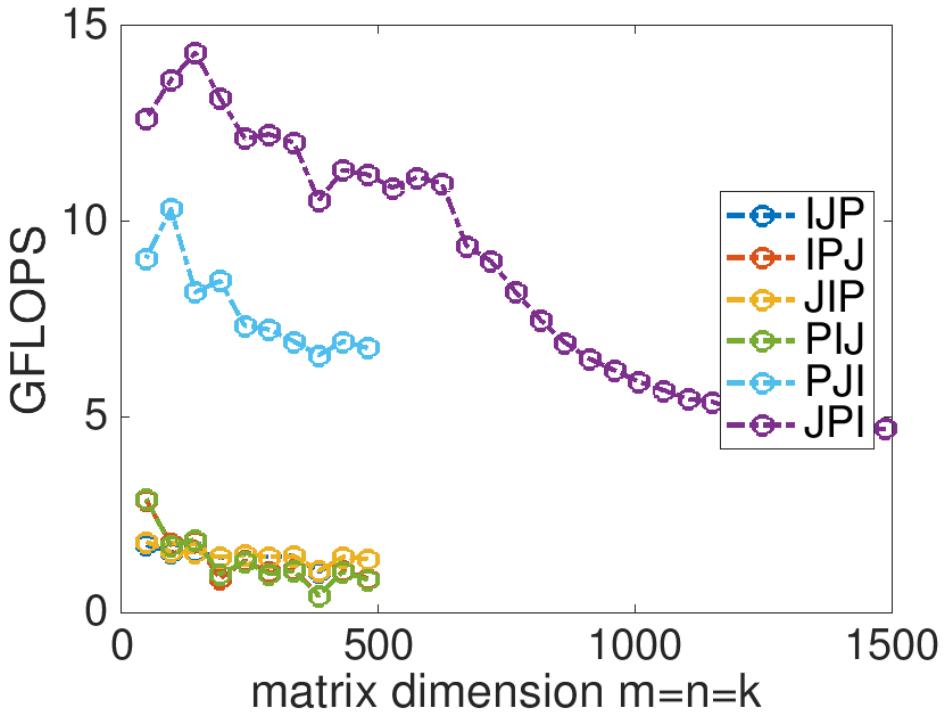
2.1 Opening Remarks

2.1.1 Launch



YouTube: https://www.youtube.com/watch?v=YJasouTMgXg

Last week, you compared the performance of a number of different implementations of matrix-matrix multiplication. At the end of Unit 1.2.4 you found that the JPI ordering did much better than the other orderings, and you probably felt pretty good with the improvement in performance you achieved:



Homework 2.1.1.1 In directory `Assignments/Week2/C/` execute
`make JPI`

and view the results with the Live Script in `Assignments/Week2/C/data/Plot_Opener.mlx`. (This may take a little while, since the Makefile now specifies that the largest problem to be executed is $m = n = k = 1500$.)

Next, change that Live Script to also show the performance of the reference implementation provided by BLIS: Change

```
% Optionally show the reference implementation performance data
if ( 0 )
```

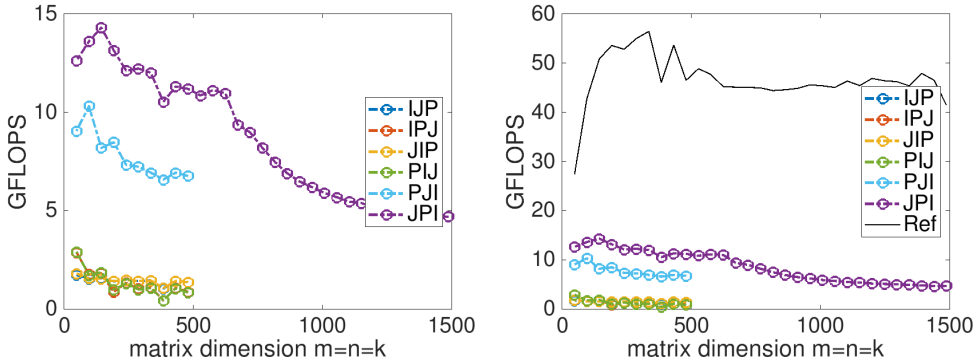
to

```
% Optionally show the reference implementation performance data
if ( 1 )
```

and rerun the Live Script. This adds a plot to the graph for the reference implementation.

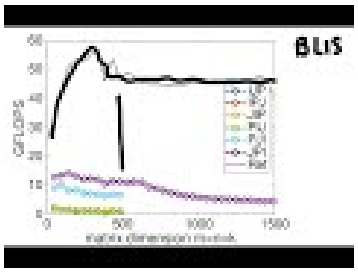
What do you observe? Now are you happy with the improvements you made in Week 1?

Solution. On Robert's laptop:



Left: Plotting only simple implementations from Week 1. Right: Adding the performance of the reference implementation provided by BLIS.

Note: the performance in the graph on the left may not exactly match that in the graph earlier in this unit. My laptop does not always attain the same performance. When a processor gets hot, it "clocks down." This means the attainable performance goes down. A laptop is not easy to cool, so one would expect more fluxuation than when using, for example, a desktop or a server.



YouTube: https://www.youtube.com/watch?v=eZaq451nuaE

Remark 2.1.1 What you notice is that we have a long way to go... By the end of Week 3, you will discover how to match the performance of the "reference" implementation.

It is useful to understand what the peak performance of a processor is. We always tell our students "What if you have a boss who simply keeps insisting that you improve performance. It would be nice to be able convince such a person that you are near the limit of what can be achieved." Unless, of course, you are paid per hour and have complete job security. Then you may decide to spend as much time as your boss wants you to!

Later this week, you learn that modern processors employ parallelism in the floating point unit so that multiple flops can be executed per cycle. In the case of the CPUs we target, 16 flops can be executed per cycle.

Homework 2.1.1.2 Next, you change the Live Script so that the top of the graph represents the theoretical peak of the core. Change

```
% Optionally change the top of the graph to capture the theoretical peak
      if ( 0 )
        turbo_clock_rate = 4.3;
        flops_per_cycle = 16;
```

to

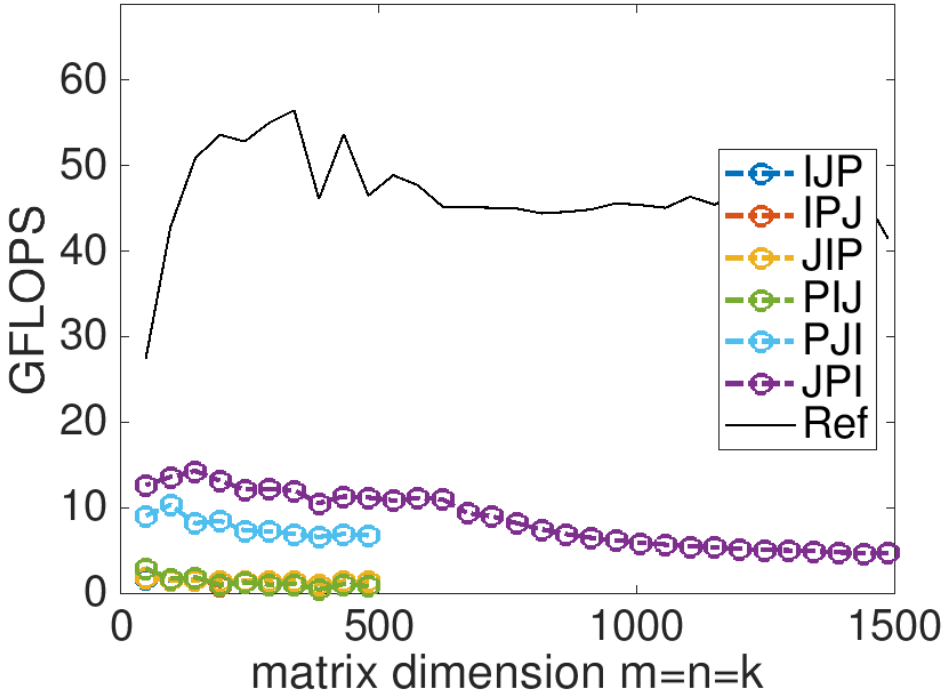
```
% Optionally change the top of the graph to capture the theoretical peak
       if ( 1 )
         turbo_clock_rate = ?.?;
         flops_per_cycle = 16;
```

where ?.? equals the turbo clock rate (in GHz) for your processor. (See Unit 0.3.1 on how to find out information about your processor).

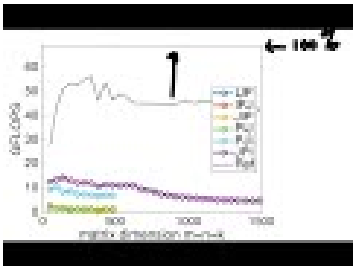
Rerun the Live Script. This changes the range of the y-axis so that the top of the graph represents the theoretical peak of one core of your processor.

What do you observe? Now are you happy with the improvements you made in Week 1?

Solution. Robert's laptop has a turbo clock rate of 4.3 GHz:



Notice that the clock rate of your processor changes with circumstances. The performance in the different graphs is not consistent and when one runs this experiment at different times on my laptop, it reports different performance. On a properly cooled Linux desktop or server, you will probably see much more consistent performance.



YouTube: https://www.youtube.com/watch?v=XfwPVkDnZF0

2.1.2 Outline Week 2

2.1.3 What you will learn

In this week, we learn how to attain high performance for small matrices by exploiting instruction-level parallelism.

Upon completion of this week, we will be able to

- Realize the limitations of simple implementations by comparing their performance to that of a high-performing reference implementation.
- Recognize that improving performance greatly doesn't necessarily yield good performance.

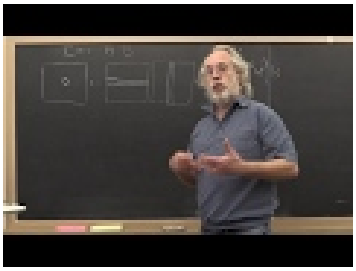
- Determine the theoretical peak performance of a processing core.
- Orchestrate matrix-matrix multiplication in terms of computations with submatrices.
- Block matrix-matrix multiplication for registers.
- Analyze the cost of moving data between memory and registers.
- Cast computation in terms of vector intrinsic functions that give access to vector registers and vector instructions.
- Optimize the micro-kernel that will become our unit of computation for future optimizations.

The enrichments introduce us to

- Theoretical lower bounds on how much data must be moved between memory and registers when executing a matrix-matrix multiplication.
- Strassen's algorithm for matrix-matrix multiplication.

2.2 Blocked Matrix-Matrix Multiplication

2.2.1 Basic idea



YouTube: https://www.youtube.com/watch?v=9t_Ux_7K-8M

Remark 2.2.1 If the material in this section makes you scratch your head, you may want to go through the materials in Week 5 of our other MOOC: Linear Algebra: Foundations to Frontiers.

Key to understanding how we are going to optimize matrix-matrix multiplication is to understand blocked matrix-matrix multiplication (the multiplication of matrices that have been partitioned into submatrices).

Consider $C := AB+C$. In terms of the elements of the matrices, this means that each $\gamma_{i,j}$ is computed by

$$\gamma_{i,j} := \sum_{p=0}^{k-1} \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}.$$

Now, partition the matrices into submatrices:

$$C = \left(\begin{array}{c|c|c|c} C_{0,0} & C_{0,1} & \cdots & C_{0,N-1} \\ \hline C_{1,0} & C_{1,1} & \cdots & C_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline C_{M-1,0} & C_{M-1,1} & \cdots & C_{M-1,N-1} \end{array}\right),$$

$$A = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \cdots & A_{0,K-1} \\ \hline A_{1,0} & A_{1,1} & \cdots & A_{1,K-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{M-1,0} & A_{M-1,1} & \cdots & A_{M-1,K-1} \end{array}\right),$$

and

$$B = \left(\begin{array}{c|c|c|c} B_{0,0} & B_{0,1} & \cdots & B_{0,N-1} \\ \hline B_{1,0} & B_{1,1} & \cdots & B_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline B_{K-1,0} & B_{K-1,1} & \cdots & B_{K-1,N-1} \end{array}\right),$$

where $C_{i,j}$ is $m_i \times n_j$, $A_{i,p}$ is $m_i \times k_p$, and $B_{p,j}$ is $k_p \times n_j$, with $\sum_{i=0}^{M-1} m_i = m$, $\sum_{j=0}^{N-1} n_i = n$, and $\sum_{p=0}^{K-1} k_i = k$. Then

$$C_{i,j} := \sum_{p=0}^{K-1} A_{i,p} B_{p,j} + C_{i,j}.$$

Remark 2.2.2 In other words, provided the partitioning of the matrices is "conformal," matrix-matrix multiplication with partitioned matrices proceeds exactly as does matrix-matrix multiplication with the elements of the matrices except that the individual multiplications with the submatrices do not necessarily commute.

We illustrate how to block C into $m_b \times n_b$ submatrices, A into $m_b \times k_b$ submatrices, and B into $k_b \times n_b$ submatrices in Figure 2.2.3

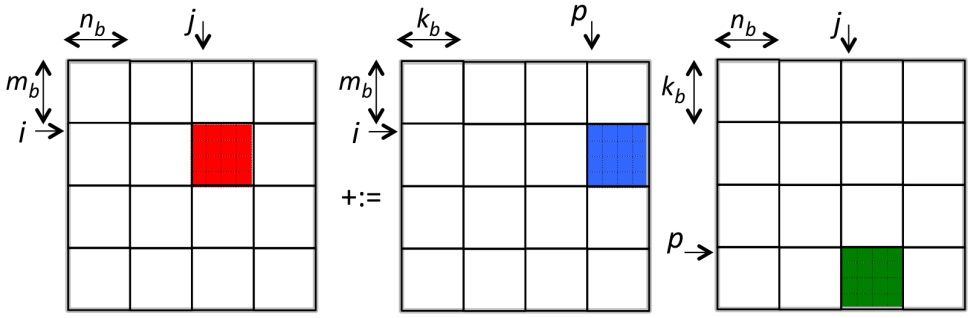


Figure 2.2.3: An illustration of a blocked algorithm where $m_b = n_b = k_b = 4$.

The blocks used when updating $C_{1,2} := A_{1,3}B_{3,2} + C_{1,2}$ is highlighted in that figure.

An implementation of one such algorithm is given in Figure 2.2.4.

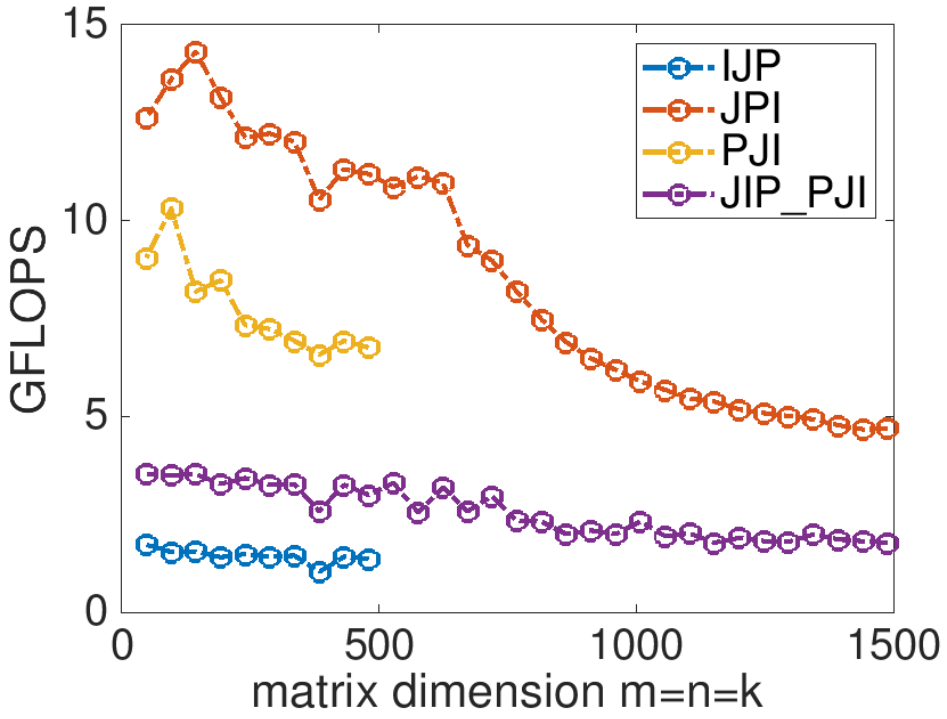
```
void MyGemm( int m, int n, int k, double *A, int ldA,
             double *B, int ldB, double *C, int ldC )
{
  for ( int j=0; j<n; j+=NB ){
    int jb = min( n-j, NB );    /* Size for "fringe" block */
    for ( int i=0; i<m; i+=MB ){
      int ib = min( m-i, MB );    /* Size for "fringe" block */
      for ( int p=0; p<k; p+=KB ){
        int pb = min( k-p, KB );    /* Size for "fringe" block */
        Gemm_PJI( ib, jb, pb, &alpha( i,p ), ldA, &beta( p,j ), ldB,
                              &gamma( i,j ), ldC );
      }
    }
  }
}

void Gemm_PJI( int m, int n, int k, double *A, int ldA,
               double *B, int ldB,  double *C, int ldC )
{
  for ( int p=0; p<k; p++ )
    for ( int j=0; j<n; j++ )
      for ( int i=0; i<m; i++ )
        gamma( i,j ) += alpha( i,p ) * beta( p,j );
}
```

Figure 2.2.4: Simple blocked algorithm that calls `Gemm_PJI` which in this setting updates a $m_b \times n_b$ submatrix of C with the result of multiplying an $m_b \times k_b$ submatrix of A times a $k_b \times n_b$ submatrix of B.

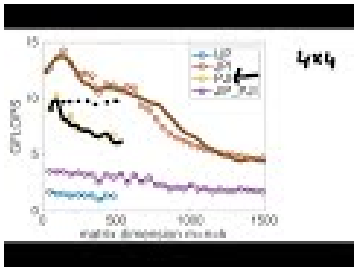
Homework 2.2.1.1 In directory `Assignments/Week2/C`, examine the code in file `Assignments/Week2/C/Gemm_JIP_PJI.c`. Time it by executing
`make JIP_PJI`

View its performance with `Assignments/Week2/C/data/Plot_Blocked_MMM.mlx`

Solution. When `MB=4`, `NB=4`, and `KB=4` in `Gemm_JIP_PJI`, the performance looks like



on Robert's laptop.



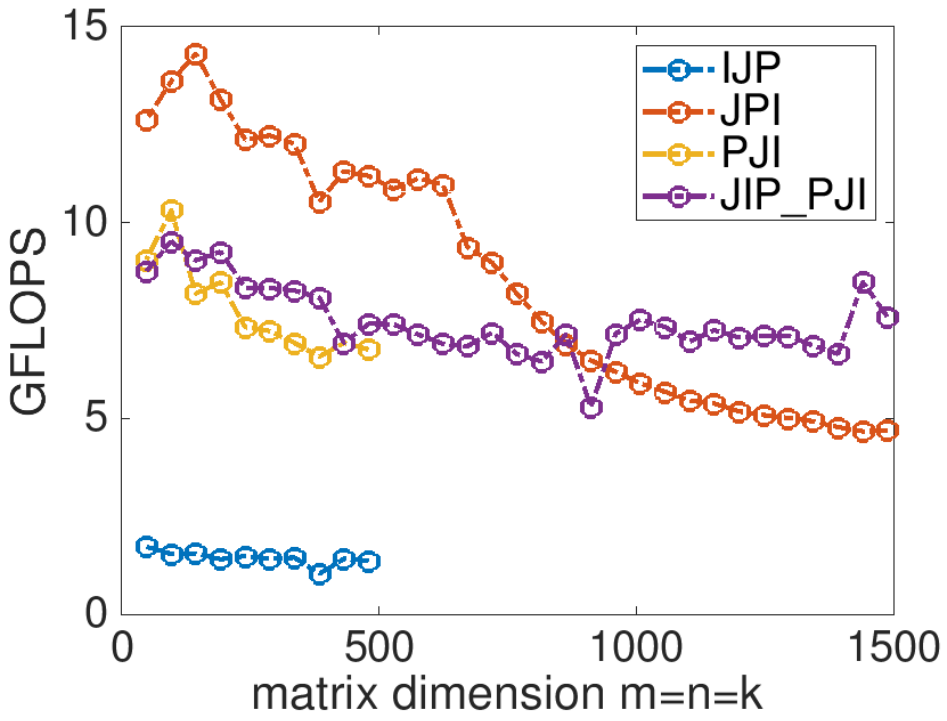
YouTube: https://www.youtube.com/watch?v=GIf4qRxYA6g

Homework 2.2.1.2 Examine the graph created by `Plot_Blocked_MMM.mlx`. What block sizes `MB` and `NB` would you expect to yield better performance? Change these in `Gemm_JIP_PJI.c` and execute

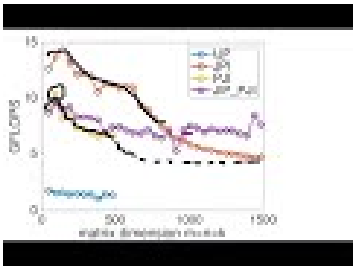
```
make JIP_PJI
```

View the updated performance with `Assignments/Week2/C/data/Plot_Blocked_MMM.mlx` (Don't worry about trying to pick the best block size. We'll get to that later.)

Solution. When we choose `MB=100`, `NB=100`, and `KB=100` better performance is maintained as the problem size gets larger:



This is because the subproblems now fit is one of the caches. More on this later in this week.



YouTube: https://www.youtube.com/watch?v=o9YEFPgDek0

2.2.2 Haven't we seen this before?

We already employed various cases of blocked matrix-matrix multiplication in Section 1.3.

Homework 2.2.2.1 In Section 1.3, we already used the blocking of matrices to cast matrix-matrix multiplications in terms of dot products and `axpy` operations. This is captured in

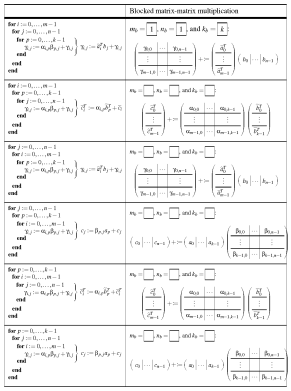


Figure 2.2.5: Click here to enlarge.

For each of partitionings in the right column, indicate how m_b, n_b, and k_b are chosen.

Homework 2.2.2.2 In Section 1.3 and Section 1.4, we also already used the blocking of matrices to cast matrix-matrix multiplications in terms of matrix-vector multiplication and rank-1 updates. This is captured in

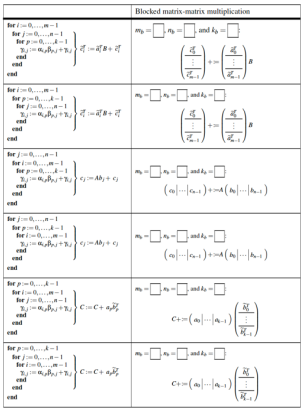
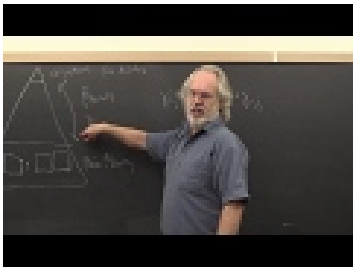


Figure 2.2.6: Click here to enlarge.

For each of partitionings in the right column, indicate how m_b, n_b, and k_b are chosen.

2.3 Blocking for Registers

2.3.1 A simple model of memory and registers



YouTube: https://www.youtube.com/watch?v=-DZuJfu1t1Q

For computation to happen, input data must be brought from memory into registers and, sooner or later, output data must be written back to memory.

For now let us make the following assumptions:

- Our processor has only one core.
- That core only has two levels of memory: registers and main memory, as illustrated in Figure 2.3.1.

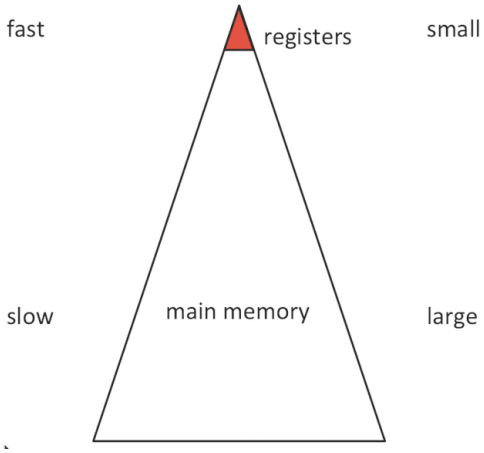
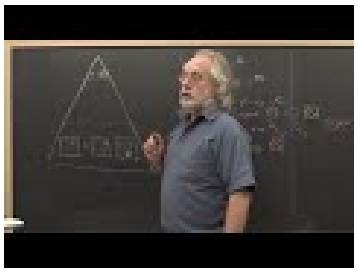


Figure 2.3.1: A simple model of the memory hierarchy: slow memory and fast registers.

- Moving data between main memory and registers takes time $\beta_{R\leftrightarrow M}$ per double. The $R \leftrightarrow M$ is meant to capture movement between registers (R) and memory (M).
- The registers can hold 64 doubles.
- Performing a flop with data in registers takes time γ_R.
- Data movement and computation cannot overlap.

Later, we will refine some of these assumptions.

2.3.2 Simple blocking for registers



YouTube: https://www.youtube.com/watch?v=9-cy_4JPLWw

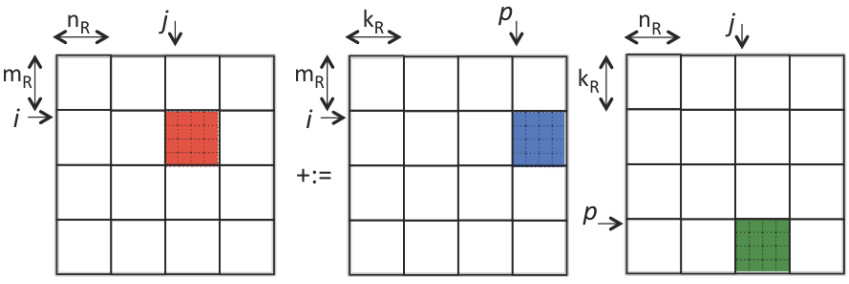
Let's first consider blocking C, A, and B into 4×4 submatrices. If we store all three submatrices in registers, then $3 \times 4 \times 4 = 48$ doubles are stored in registers.

Remark 2.3.2 Yes, we are not using all registers (since we assume registers can hold 64 doubles). We'll get to that later.

Let us call the routine that computes with three blocks "the kernel". Now the blocked algorithm is a triple-nested loop around a call to this kernel, as captured by the following triple-nested loop

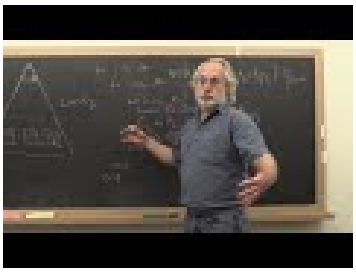
for $j := 0, \ldots, N-1$
  for $i := 0, \ldots, M-1$
    for $p := 0, \ldots, K-1$
      Load $C_{i,j}, A_{i,p},$ and $B_{p,j}$ into registers
      $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$
      Store $C_{i,j}$ to memory
    end
  end
end

and illustrated by



where we use the subscript "R" to indicate it is a block size that targets registers.

Remark 2.3.3 For our discussion, it is important to order the p loop last. Next week, it will become even more important to order loops around the kernel in a specific order. We are picking the JIP order here in preparation for that. If you feel like it, you can play with the IJP order (around the kernel) to see how it compares.



YouTube: https://www.youtube.com/watch?v=NurvjVEo9nA

The time required to execute this algorithm under our model is given by

$$
\begin{array}{l}
\underbrace{MNK}_{\substack{\text{numb.}\\ \text{of MMM}\\ \text{with blocks}}} \left(\underbrace{(m_R n_R + m_R k_R + k_R n_R)\beta_{R\leftrightarrow M}}_{\text{Load blocks of } A, B, \text{ and } C} + \underbrace{2m_R n_R k_R \gamma_R}_{\substack{\text{multiplication}\\ \text{with blocks}}} + \underbrace{m_R n_R \beta_{R\leftrightarrow M}}_{\text{Write block of } C} \right) \\
\quad = \quad < \text{rearrange} > \\
2(Mm_R)(Nn_R)(Kk_R)\gamma_R + 2(Mm_R)(Nn_R)K\beta_{R\leftrightarrow M} \\
\qquad\qquad + (Mm_R)N(Kk_R)\beta_{R\leftrightarrow M} + M(Nn_R)(Kk_R)\beta_{R\leftrightarrow M} \\
\quad = \quad < \text{simplify, distribute, commute, etc.} > \\
\quad \underbrace{2mnk\gamma_R}_{\substack{\text{useful}\\ \text{computation}}} + \underbrace{mnk(\frac{2}{k_R} + \frac{1}{n_R} + \frac{1}{m_R})\beta_{R\leftrightarrow M},}_{\text{overhead}}
\end{array}
$$

since $M = m/m_R$, $N = n/n_R$, and $K = k/k_R$. (Here we assume that m, n, and k are integer multiples of m_R, n_R, and k_R.)

Remark 2.3.4 The time spent in computation,

$$2mnk\gamma_R,$$

is useful time and the time spent in loading and storing data,

$$mnk(\frac{2}{k_R} + \frac{1}{n_R} + \frac{1}{m_R})\beta_{R\leftrightarrow M},$$

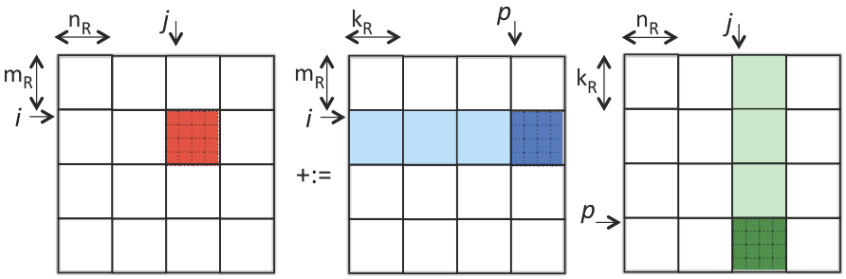
is overhead.

Next, we recognize that since the loop indexed with p is the inner-most loop, the loading and storing of $C_{i,j}$ does not need to happen before every call

to the kernel, yielding the algorithm

```
for i := 0, ..., M − 1
  for j := 0, ..., N − 1
    Load C_{i,j} into registers
    for p := 0, ..., K − 1
      Load A_{i,p}, and B_{p,j} into registers
      C_{i,j} := A_{i,p} B_{p,j} + C_{i,j}
    end
    Store C_{i,j} to memory
  end
end
```

as illustrated by



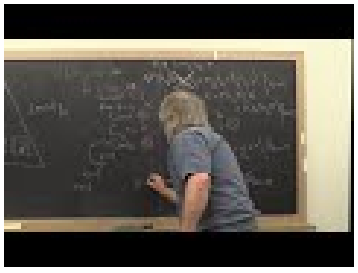
Homework 2.3.2.1 In detail explain the estimated cost of this modified algorithm.

Solution.

$$MNK(2m_Rn_Rk_R)\gamma_R + [MN(2m_Rn_R) + MNK(m_Rk_R + k_Rn_R)]\,\beta_{R\leftrightarrow M}$$
$$= 2mnk\gamma_R + \left[2mn + mnk(\tfrac{1}{n_R} + \tfrac{1}{m_R})\right]\beta_{R\leftrightarrow M}.$$

Here

- $2mnk\gamma_R$ is time spent in useful computation.
- $\left[2mn + mnk(\frac{1}{n_R} + \frac{1}{m_R})\right]\beta_{R\leftrightarrow M}$ is time spent moving data around, which is overhead.



YouTube: `https://www.youtube.com/watch?v=hV7FmIDR4r8`

Homework 2.3.2.2 In directory `Assignments/Week2/C` copy the file `Assignments/Week2/C/Gemm_JIP_PJI.c` into `Gemm_JI_PJI.c` and combine the loops indexed by P, in the process "removing" it from `MyGemm`. Think carefully about how the

call to `Gemm_PJI` needs to be changed:

- What is the matrix-matrix multiplication that will now be performed by `Gemm_PJI`?
- How do you express that in terms of the parameters that are passed to that routine?

(Don't worry about trying to pick the best block size. We'll get to that later.)

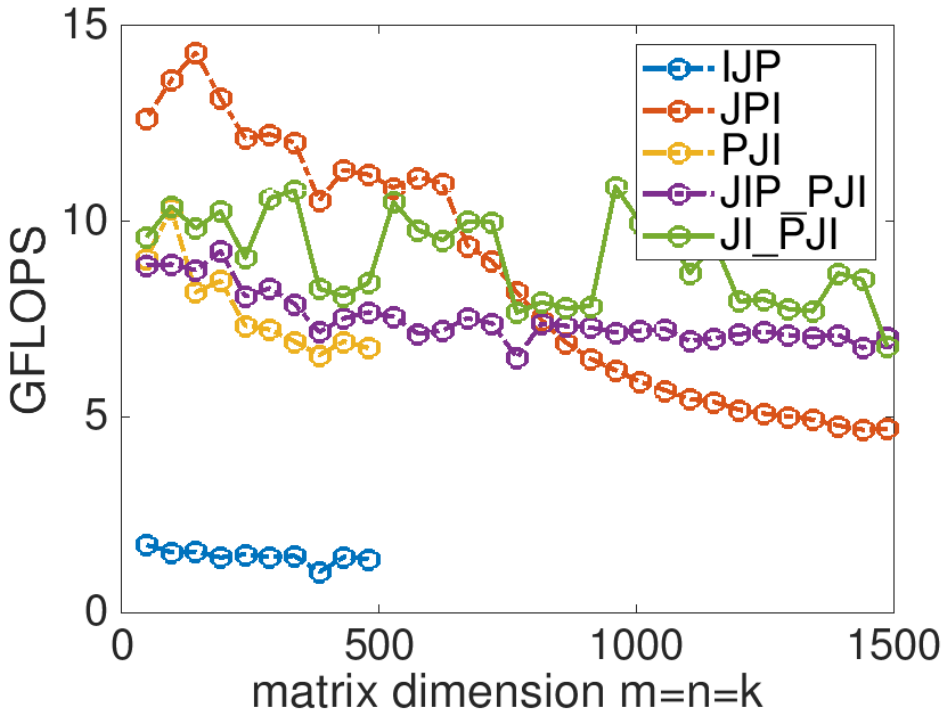
You can test the result by executing

```
make JI_PJI
```

View the performance by making the appropriate changes to `Assignments/Week2/C/data/Plot_Blocked_MMM.mlx`.

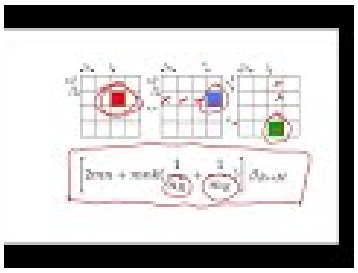
Solution. `Assignments/Week2/Answers/Gemm_JI_PJI.c`

This is the performance on Robert's laptop, with `MB=NB=KB=100`:



The performance you observe may not really be an improvement relative to `Gemm_JIP_PJI.c`. What we observe here may be due to variation in the conditions under which the performance was measured.

2.3.3 Streaming $A_{i,p}$ and $B_{p,j}$



YouTube: https://www.youtube.com/watch?v=62WAIASy1BA

Now, if the submatrix $C_{i,j}$ were larger (i.e., m_R and n_R were larger), then the time spent moving data around (overhead),

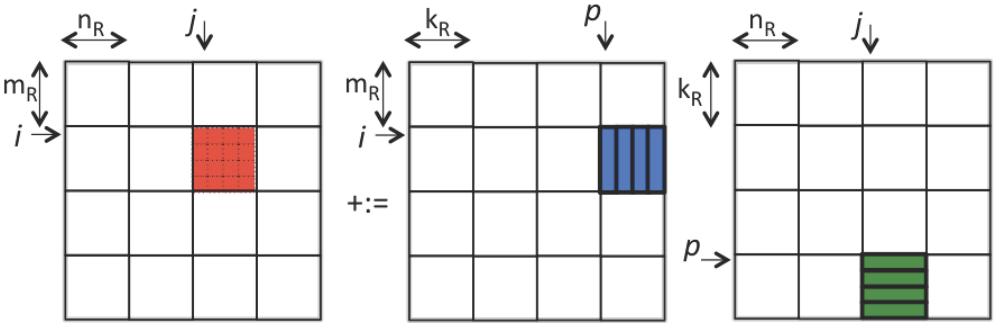
$$\left[2mn + mnk(\frac{1}{n_R} + \frac{1}{m_R})\right] \beta_{R\leftrightarrow M},$$

is reduced:

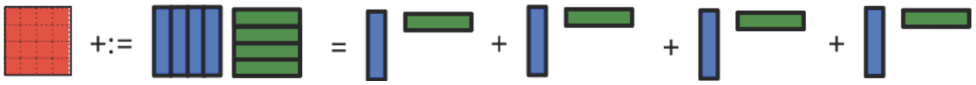
$$\frac{1}{n_R} + \frac{1}{m_R}$$

decreases as m_R and n_R increase. In other words, the larger the submatrix of C that is kept in registers, the lower the overhead (under our model). Before we increase m_R and n_R, we first show how for fixed m_R and n_R the number of registers that are needed for storing elements of A and B can be reduced, which then allows m_R and n_R to (later this week) be increased in size.

Recall from Week 1 that if the p loop of matrix-matrix multiplication is the outer-most loop, then the inner two loops implement a rank-1 update. If we do this for the kernel, then the result is illustrated by



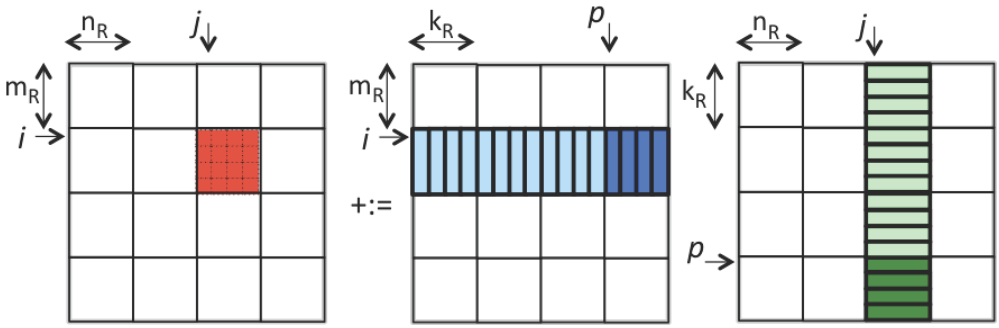
If we now consider the computation



we recognize that after each rank-1 update, the column of $A_{i,p}$ and row of $B_{p,j}$ that participate in that rank-1 update are not needed again in the computation of $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$. Thus, only room for one such column of $A_{i,p}$ and one such row of $B_{p,j}$ needs to be reserved in the registers, reducing the total number of doubles that need to be stored in registers to

$$m_R \times n_R + m_R + n_R.$$

If $m_R = n_R = 4$ this means that $16 + 4 + 4 = 24$ doubles in registers. If we now view the entire computation performed by the loop indexed with p, this can be illustrated by



and implemented as

```
void MyGemm( int m, int n, int k, double *A, int ldA,
                          double *B, int ldB, double *C, int ldC )
{
  for ( int i=0; i<m; i+=MR ) /* m assumed to be multiple of MR */
    for ( int j=0; j<n; j+=NR ) /* n assumed to be multiple of NR */
      for ( int p=0; p<k; p+=KR ) /* k assumed to be multiple of KR */
        Gemm_P_Ger( MR, NR, KR, &alpha( i,p ), ldA,
                                &beta( p,j ), ldB, &gamma( i,j ), ldC );
}

void Gemm_P_Ger( int m, int n, int k, double *A, int ldA,
                       double *B, int ldB, double *C, int ldC )
{
  for ( int p=0; p<k; p++ )
    MyGer( m, n, &alpha( 0,p ), 1, &beta( p,0 ), ldB, C, ldC );
}
```

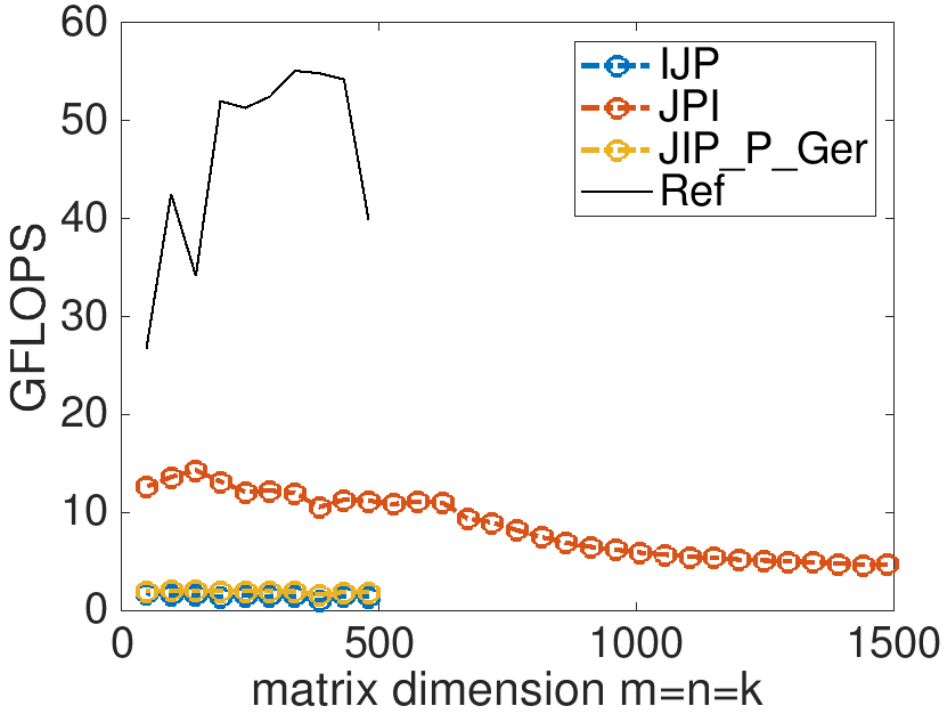
Figure 2.3.5: Blocked matrix-matrix multiplication with kernel that casts $C_{i,j} = A_{i,p}B_{p,j} + C_{i,j}$ in terms of rank-1 updates.

Homework 2.3.3.1 In directory `Assignments/Week2/C` examine the file `Assignments/Week2/C/Gemm_JIP_P_Ger.c` Execute it with

```
make JIP_P_Ger
```

and view its performance with `Assignments/Week2/C/data/Plot_register_blocking.mlx`.

Solution. This is the performance on Robert's laptop, with `MB=NB=KB=4`:

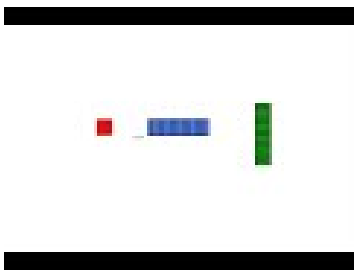


The performance is pathetic. It will improve!

It is tempting to start playing with the parameters `MB`, `NB`, and `KB`. However, the point of this exercise is to illustrate the discussion about casting the multiplication with the blocks in terms of a loop around rank-1 updates.

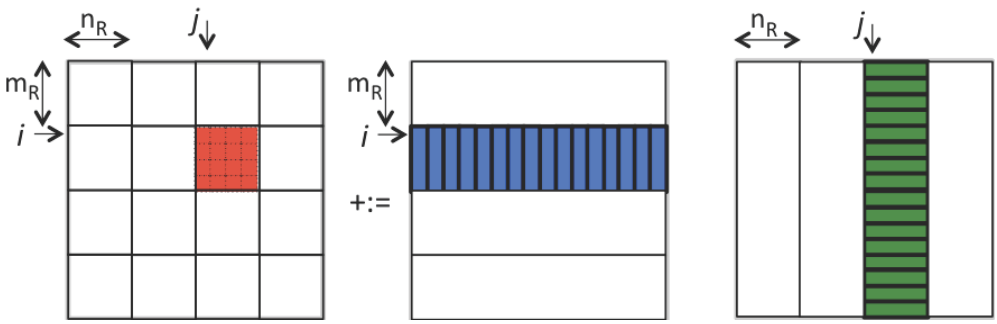
Remark 2.3.6 The purpose of this implementation is to emphasize, again, that the matrix-matrix multiplication with blocks can be orchestratred as a loop around rank-1 updates, as you already learned in Unit 1.4.2.

2.3.4 Combining loops

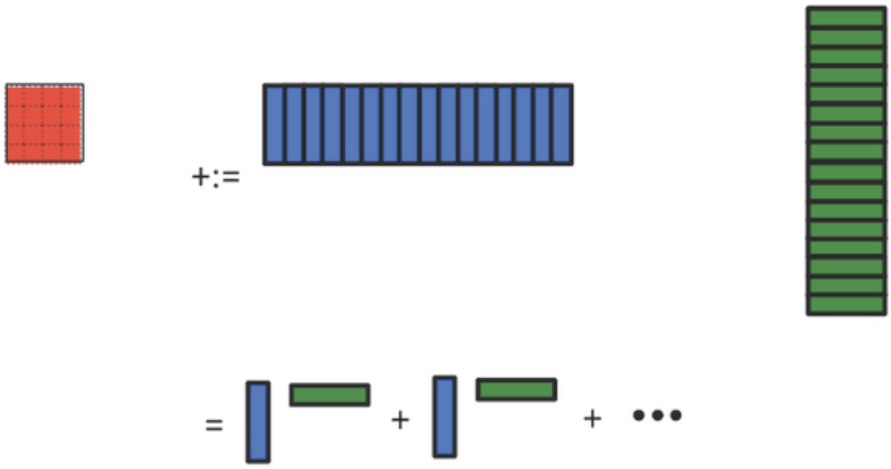


YouTube: `https://www.youtube.com/watch?v=H8jvyd0NO4M`

What we now recognize is that matrices A and B need not be partitioned into $m_R \times k_R$ and $k_R \times n_R$ submatrices (respectively). Instead, A can be partitioned into $m_R \times k$ row panels and B into $k \times n_R$ column panels:



Another way of looking at this is that the inner loop indexed with p in the blocked algorithm can be combined with the outer loop of the kernel. The entire update of the $m_R \times n_R$ submatrix of C can then be pictured as



This is the computation that will later become instrumental in optimizing and will be called the micro-kernel.

Homework 2.3.4.1 In directory `Assignments/Week2/C` copy the file `Assignments/Week2/C/Gemm_JIP_P_Ger.c` into `Gemm_JI_P_Ger.c` and remove the p loop from `MyGemm`. Execute it with

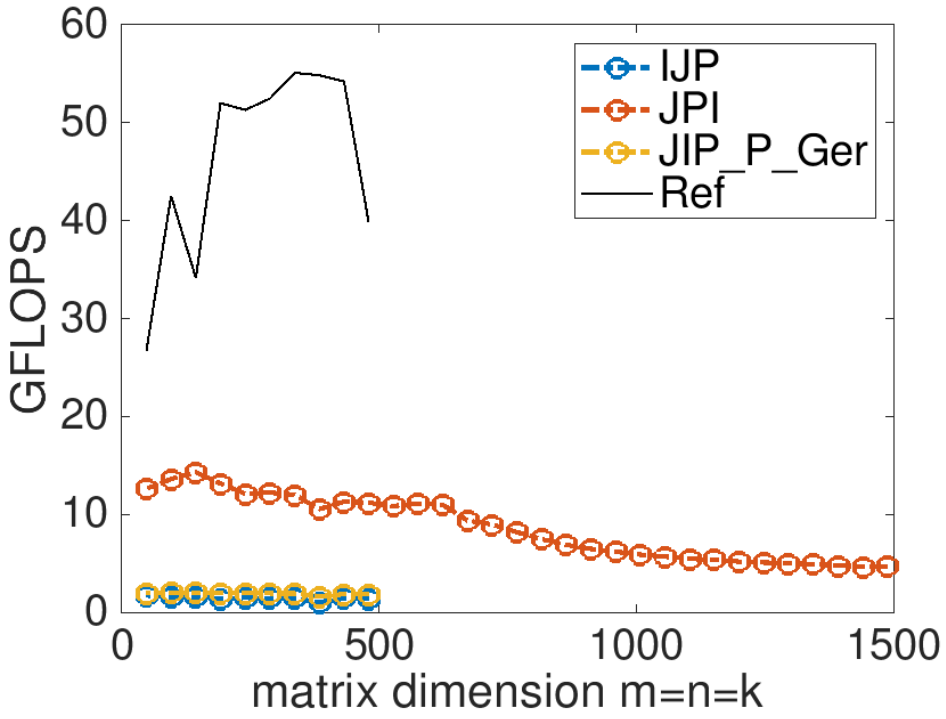
```
make JI_P_Ger
```

and view its performance with `Assignments/Week2/C/data/Plot_register_blocking.mlx`

Solution.

- `Assignments/Week2/Answers/Gemm_JI_P_Ger.c`

This is the performance on Robert's laptop, with `MB=NB=KB=4`:



The performance does not really improve, and continues to be pathetic.

Again, it is tempting to start playing with the parameters `MB`, `NB`, and `KB`. However, the point of this exercise is to illustrate the discussion about how the loops indexed with p can be combined, casting the multiplication with the blocks in terms of a loop around rank-1 updates.

Homework 2.3.4.2 In detail explain the estimated cost of the implementation in Homework 2.3.4.1.

Solution. Bringing $A_{i,p}$ one column at a time into registers and $B_{p,j}$ one row at a time into registers does not change the cost of reading that data. So, the cost of the resulting approach is still estimated as

$$\begin{aligned} & MNK(2m_R n_R k_R)\gamma_R + [MN(2m_R n_R) + MNK(m_R k_R + k_R n_R)]\,\beta_{R\leftrightarrow M} \\ & \quad = 2mnk\gamma_R + \left[2mn + mnk(\tfrac{1}{n_R} + \tfrac{1}{m_R})\right]\beta_{R\leftrightarrow M}. \end{aligned}$$

Here

- $2mnk\gamma_R$ is time spent in useful computation.
- $\left[2mn + mnk(\frac{1}{n_R} + \frac{1}{m_R})\right]\beta_{R\leftrightarrow M}$ is time spent moving data around, which is overhead.

The important insight is that now only the $m_R \times n_R$ micro-tile of C, one small column A of size m_R one small row of B of size n_R need to be stored in registers. In other words, $m_R \times n_R + m_R + n_R$ doubles are stored in registers at any one time.

One can go one step further: Each individual rank-1 update can be imple-

mented as a sequence of axpy operations:

$$\left(\begin{array}{c|c|c|c} \gamma_{0,0} & \gamma_{0,1} & \gamma_{0,2} & \gamma_{0,3} \\ \gamma_{1,0} & \gamma_{1,1} & \gamma_{1,2} & \gamma_{1,3} \\ \gamma_{2,0} & \gamma_{2,1} & \gamma_{2,2} & \gamma_{2,3} \\ \gamma_{3,0} & \gamma_{3,1} & \gamma_{3,2} & \gamma_{3,3} \end{array}\right) +:= \left(\begin{array}{c} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{array}\right) \left(\begin{array}{c|c|c|c} \beta_{p,0} & \beta_{p,1} & \beta_{p,2} & \beta_{p,3} \end{array}\right)$$
$$= \left(\begin{array}{c|c|c|c} \beta_{p,0} \left(\begin{array}{c} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{array}\right) & \beta_{p,1} \left(\begin{array}{c} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{array}\right) & \beta_{p,2} \left(\begin{array}{c} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{array}\right) & \beta_{p,3} \left(\begin{array}{c} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{array}\right) \end{array}\right).$$

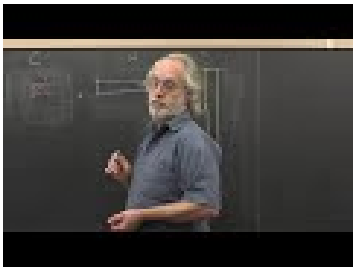
Here we note that if the block of C is updated one column at a time, the corresponding element of B, $\beta_{p,j}$ in our discussion, is not reused and hence only one register is needed for the elements of B. Thus, when $m_R = n_R = 4$, we have gone from storing 48 doubles in registers to 24 and finally to only 21 doubles.. More generally, we have gone from $m_R n_R + m_R k_R + k_R n_R$ to $m_R n_R + m_R + n_R$ to

$$m_R n_R + m_R + 1$$

doubles.

Obviously, one could have gone further yet, and only store one element $\alpha_{i,p}$ at a time. However, in that case one would have to reload such elements multiple times, which would adversely affect overhead.

2.3.5 Alternative view



YouTube: https://www.youtube.com/watch?v=FsRIYsoqrms

We can arrive at the same algorithm by partitioning

$$C = \left(\begin{array}{c|c|c|c} C_{0,0} & C_{0,1} & \cdots & C_{0,N-1} \\ \hline C_{1,0} & C_{1,1} & \cdots & C_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline C_{M-1,0} & C_{M,1} & \cdots & C_{M-1,N-1} \end{array}\right), A = \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline \vdots \\ \hline A_{M-1} \end{array}\right),$$

and

$$B = \left(\begin{array}{c|c|c|c} B_0 & B_1 & \cdots & B_{0,N-1} \end{array}\right),$$

where $C_{i,j}$ is $m_R \times n_R$, A_i is $m_R \times k$, and B_j is $k \times n_R$. Then computing

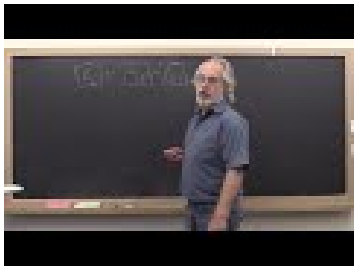
$C := AB + C$ means updating $C_{i,j} := A_i B_j + C_{i,j}$ for all i, j:

for $j := 0, \ldots, N-1$
  for $i := 0, \ldots, M-1$
    $C_{i,j} := A_i B_j + C_{i,j}$ Computed with the micro-kernel
  end
end

Obviously, the order of the two loops can be switched. Again, the computation $C_{i,j} := A_i B_j + C_{i,j}$ where $C_{i,j}$ fits in registers now becomes what we will call the micro-kernel.

2.4 Optimizing the Micro-kernel

2.4.1 Vector registers and instructions



YouTube: https://www.youtube.com/watch?v=9L_0XNGuhv4

While the last unit introduced the notion of registers, modern CPUs accelerate computation by computing with small vectors of of numbers (double) simultaneously.

As the reader should have noticed by now, in matrix-matrix multiplication for every floating point multiplication a corresponding floating point addition is encountered to accumulate the result:

$$\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$$

For this reason, such floating point computations are usually cast in terms of fused multiply add (FMA) operations, performed by a floating point unit (FPU) of the core.

What is faster than computing one FMA at a time? Computing multiple FMAs at a time! For this reason, modern cores compute with small vectors of data, performing the same FMA on corresponding elements in those vectors, which is referred to as "SIMD" computation: Single-Instruction, Multiple Data.

This exploits instruction-level parallelism.

Let's revisit the computation

$$\begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} & \gamma_{0,2} & \gamma_{0,3} \\ \gamma_{1,0} & \gamma_{1,1} & \gamma_{1,2} & \gamma_{1,3} \\ \gamma_{2,0} & \gamma_{2,1} & \gamma_{2,2} & \gamma_{2,3} \\ \gamma_{3,0} & \gamma_{3,1} & \gamma_{3,2} & \gamma_{3,3} \end{pmatrix} +:= \begin{pmatrix} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{pmatrix} \begin{pmatrix} \beta_{p,0} & \beta_{p,1} & \beta_{p,2} & \beta_{p,3} \end{pmatrix}$$
$$= \beta_{p,0} \begin{pmatrix} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{pmatrix} + \beta_{p,1} \begin{pmatrix} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{pmatrix} + \beta_{p,2} \begin{pmatrix} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{pmatrix} + \beta_{p,3} \begin{pmatrix} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{pmatrix}$$

that is at the core of the micro-kernel.

If a vector register has length four, then it can store four (double precision) numbers. Let's load one such vector register with a column of the submatrix of C, a second vector register with the vector from A, and a third with an element of B that has been duplicated:

$\gamma_{0,0}$
$\gamma_{1,0}$
$\gamma_{2,0}$
$\gamma_{3,0}$

$\alpha_{0,p}$
$\alpha_{1,p}$
$\alpha_{2,p}$
$\alpha_{3,p}$

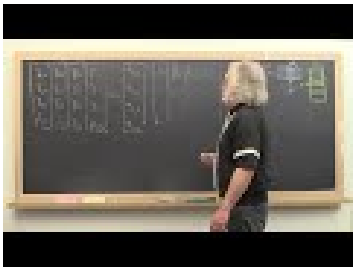
$\beta_{p,0}$
$\beta_{p,0}$
$\beta_{p,0}$
$\beta_{p,0}$

A vector instruction that simultaneously performs FMAs with each tuple $(\gamma_{i,0}, \alpha_{i,p}, \beta_{p,0})$ can then be performed:

$\gamma_{0,0}$	$+:=$	$\alpha_{0,p}$	$\times$	$\beta_{p,0}$
$\gamma_{1,0}$	$+:=$	$\alpha_{1,p}$	$\times$	$\beta_{p,0}$
$\gamma_{2,0}$	$+:=$	$\alpha_{2,p}$	$\times$	$\beta_{p,0}$
$\gamma_{3,0}$	$+:=$	$\alpha_{3,p}$	$\times$	$\beta_{p,0}$

You may recognize that this setup is ideal for performing an axpy operation with a small vector (of size 4 in this example).

2.4.2 Implementing the micro-kernel with vector instructions



YouTube: https://www.youtube.com/watch?v=VhCUf9tPteU

Remark 2.4.1 This unit is one of the most important ones in the course. Take your time to understand it. The syntax of the intrinsic library is less important than the concepts: as new architectures become popular, new vector instructions will be introduced.

In this section, we are going to AVX2 vector instruction set to illustrate the ideas. Vector intrinsic functions support the use of these instructions from within the C programming language. You discover how the intrinsic operations are used by incorporating them into an implemention of the micro-kernel for the case where $m_R \times n_R = 4 \times 4$. Thus, for the remainder of this unit, C is $m_R \times n_R = 4 \times 4$, A is $m_R \times k = 4 \times k$, and B is $k \times n_R = k \times 4$.

The architectures we target have 256 bit vector registers, which mean they can store four double precision floating point numbers. Vector operations with vector registers hence operate with vectors of size four.

Let's recap: Partitioning

$$C = \left(\begin{array}{c|c|c} C_{0,0} & \cdots & C_{0,N-1} \\ \hline \vdots & & \vdots \\ \hline C_{M-1,0} & \cdots & C_{M-1,N-1} \end{array} \right),$$

$$A = \left(\begin{array}{c} A_0 \\ \hline \vdots \\ \hline A_{M-1} \end{array} \right), \quad B = \left(\begin{array}{c|c|c} B_0 & \cdots & B_{N-1} \end{array} \right),$$

the algorithm we discovered in Section 2.3 is given by

for $i := 0, \ldots, M-1$
  for $j := 0, \ldots, N-1$
    Load $C_{i,j}$ into registers
    $C_{i,j} := A_i B_j + C_{i,j}$ with micro − kernel
    Store $C_{i,j}$ to memory
  end
end

and translates into the code in Figure 2.4.2.

```
void MyGemm( int m, int n, int k, double *A, int ldA,
                  double *B, int ldB, double *C, int ldC )
{
  if ( m % MR != 0 || n % NR != 0 ){
    printf( "m and n must be multiples of MR and NR, respectively \n" );
    exit( 0 );
  }

  for ( int j=0; j<n; j+=NR ) /* n is assumed to be a multiple of NR */
    for ( int i=0; i<m; i+=MR ) /* m is assumed to be a multiple of MR */
      Gemm_MRxNRKernel( k, &alpha( i,0 ), ldA, &beta( 0,j ), ldB, &gamma( i,j ), ldC );
}
```

Figure 2.4.2: General routine for calling a $m_R \times n_R$ kernel in Assignments/Week2/C/Gemm_JI_MRxNRKernel.c. The constants MR and NR are specified at compile time by passing -D'MR=??' and -D'NR=??' to the compiler, where the ??s equal the desired choices of m_R and n_R (see also the Makefile in Assignments/Week2/C).

Let's drop the subscripts, and focus on computing $C+ := AB$ when C is 4×4, with the micro-kernel. This translates to the computation

$$
\begin{array}{l}
\left(\begin{array}{c|c|c|c} \gamma_{0,0} & \gamma_{0,1} & \gamma_{0,2} & \gamma_{0,3} \\ \gamma_{1,0} & \gamma_{1,1} & \gamma_{1,2} & \gamma_{1,3} \\ \gamma_{2,0} & \gamma_{2,1} & \gamma_{2,2} & \gamma_{2,3} \\ \gamma_{3,0} & \gamma_{3,1} & \gamma_{3,2} & \gamma_{3,3} \end{array}\right) \\
\quad + := \left(\begin{array}{c|c|c} \alpha_{0,0} & \alpha_{0,1} & \cdots \\ \alpha_{1,0} & \alpha_{1,1} & \cdots \\ \alpha_{2,0} & \alpha_{2,1} & \cdots \\ \alpha_{3,0} & \alpha_{3,1} & \cdots \end{array}\right) \left(\begin{array}{cccc} \beta_{0,0} & \beta_{0,1} & \beta_{0,2} & \beta_{0,3} \\ \beta_{1,0} & \beta_{1,1} & \beta_{1,2} & \beta_{1,3} \\ \vdots & \vdots & \vdots & \vdots \end{array}\right) \\
\quad = \left(\begin{array}{c|c|c|c} \alpha_{0,0}\beta_{0,0}+\alpha_{0,1}\beta_{1,0}+\cdots & \alpha_{0,0}\beta_{0,1}+\alpha_{0,1}\beta_{1,1}+\cdots & \alpha_{0,0}\beta_{0,2}+\alpha_{0,1}\beta_{1,2}+\cdots & \alpha_{0,0}\beta_{0,2}+\alpha_{0,1}\beta_{1,2}+\cdots \\ \alpha_{1,0}\beta_{0,0}+\alpha_{1,1}\beta_{1,0}+\cdots & \alpha_{1,0}\beta_{0,1}+\alpha_{1,1}\beta_{1,1}+\cdots & \alpha_{1,0}\beta_{0,2}+\alpha_{1,1}\beta_{1,2}+\cdots & \alpha_{1,0}\beta_{0,3}+\alpha_{1,1}\beta_{1,3}+\cdots \\ \alpha_{2,0}\beta_{0,0}+\alpha_{2,1}\beta_{1,0}+\cdots & \alpha_{2,0}\beta_{0,1}+\alpha_{2,1}\beta_{1,1}+\cdots & \alpha_{2,0}\beta_{0,2}+\alpha_{2,1}\beta_{1,2}+\cdots & \alpha_{2,0}\beta_{0,3}+\alpha_{2,1}\beta_{1,3}+\cdots \\ \alpha_{3,0}\beta_{0,0}+\alpha_{3,1}\beta_{1,0}+\cdots & \alpha_{3,0}\beta_{0,1}+\alpha_{3,1}\beta_{1,1}+\cdots & \alpha_{3,0}\beta_{0,2}+\alpha_{3,1}\beta_{1,2}+\cdots & \alpha_{3,0}\beta_{0,3}+\alpha_{3,1}\beta_{1,3}+\cdots \end{array}\right) \\
\quad = \left(\begin{array}{c} \alpha_{0,0} \\ \alpha_{1,0} \\ \alpha_{2,0} \\ \alpha_{3,0} \end{array}\right) \left(\begin{array}{cccc} \beta_{0,0} & \beta_{0,1} & \beta_{0,2} & \beta_{0,3} \end{array}\right) + \left(\begin{array}{c} \alpha_{0,1} \\ \alpha_{1,1} \\ \alpha_{2,1} \\ \alpha_{3,1} \end{array}\right) \left(\begin{array}{cccc} \beta_{1,0} & \beta_{1,1} & \beta_{1,2} & \beta_{1,3} \end{array}\right) + \cdots
\end{array}
$$

Thus, updating 4×4 matrix C can be implemented as a loop around rank-1 updates:

for $p = 0, \ldots, k-1$

$$
\left(\begin{array}{c|c|c|c} \gamma_{0,0} & \gamma_{0,1} & \gamma_{0,2} & \gamma_{0,3} \\ \gamma_{1,0} & \gamma_{1,1} & \gamma_{1,2} & \gamma_{1,3} \\ \gamma_{2,0} & \gamma_{2,1} & \gamma_{2,2} & \gamma_{2,3} \\ \gamma_{3,0} & \gamma_{3,1} & \gamma_{3,2} & \gamma_{3,3} \end{array}\right) + := \left(\begin{array}{c} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{array}\right) \left(\begin{array}{cccc} \beta_{p,0} & \beta_{p,1} & \beta_{p,2} & \beta_{p,3} \end{array}\right)
$$

end

or, equivalently to emphasize computations with vectors,

for $p = 0, \ldots, k-1$

$$
\left(\begin{array}{c|c|c|c} \gamma_{0,0}+ := \alpha_{0,p} \times \beta_{p,0} & \gamma_{0,1}+ := \alpha_{0,p} \times \beta_{p,1} & \gamma_{0,2}+ := \alpha_{0,p} \times \beta_{p,2} & \gamma_{0,3}+ := \alpha_{0,p} \times \beta_{p,3} \\ \gamma_{1,0}+ := \alpha_{1,p} \times \beta_{p,0} & \gamma_{1,1}+ := \alpha_{1,p} \times \beta_{p,1} & \gamma_{1,2}+ := \alpha_{1,p} \times \beta_{p,2} & \gamma_{1,3}+ := \alpha_{1,p} \times \beta_{p,3} \\ \gamma_{2,0}+ := \alpha_{2,p} \times \beta_{p,0} & \gamma_{2,1}+ := \alpha_{2,p} \times \beta_{p,1} & \gamma_{2,2}+ := \alpha_{2,p} \times \beta_{p,2} & \gamma_{2,3}+ := \alpha_{2,p} \times \beta_{p,3} \\ \gamma_{3,0}+ := \alpha_{3,p} \times \beta_{p,0} & \gamma_{3,1}+ := \alpha_{3,p} \times \beta_{p,1} & \gamma_{3,2}+ := \alpha_{3,p} \times \beta_{p,2} & \gamma_{3,3}+ := \alpha_{3,p} \times \beta_{p,3} \end{array}\right)
$$

end

This, once again, shows how matrix-matrix multiplication can be cast in terms of a sequence of rank-1 updates, and that a rank-1 update can be implemented in terms of `axpy` operations with columns of C. This micro-kernel translates into the code that employs vector instructions, given in Figure 2.4.3. We hope that the instrinsic function calls are relatively self-explanatory. However, you may want to consult Intel's Intrinsics Reference Guide. When doing so, you may want to search on the name of the routine, without checking the AVX2 box on the left.

```
#include <immintrin.h>

void Gemm_MRxNRKernel( int k, double *A, int ldA, double *B, int ldB,
                       double *C, int ldC )
{
  /* Declare vector registers to hold 4x4 C and load them */
  __m256d gamma_0123_0 = _mm256_loadu_pd( &gamma( 0,0 ) );
  __m256d gamma_0123_1 = _mm256_loadu_pd( &gamma( 0,1 ) );
  __m256d gamma_0123_2 = _mm256_loadu_pd( &gamma( 0,2 ) );
  __m256d gamma_0123_3 = _mm256_loadu_pd( &gamma( 0,3 ) );

  for ( int p=0; p<k; p++ ){
    /* Declare vector register for load/broadcasting beta( p,j ) */
    __m256d beta_p_j;

    /* Declare a vector register to hold the current column of A and load
       it with the four elements of that column. */
    __m256d alpha_0123_p = _mm256_loadu_pd( &alpha( 0,p ) );

    /* Load/broadcast beta( p,0 ). */
    beta_p_j = _mm256_broadcast_sd( &beta( p, 0) );

    /* update the first column of C with the current column of A times
       beta ( p,0 ) */
    gamma_0123_0 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_0 );

    /* REPEAT for second, third, and fourth columns of C.  Notice that the
       current column of A needs not be reloaded. */



  }

  /* Store the updated results */
  _mm256_storeu_pd( &gamma(0,0), gamma_0123_0 );
  _mm256_storeu_pd( &gamma(0,1), gamma_0123_1 );
  _mm256_storeu_pd( &gamma(0,2), gamma_0123_2 );
  _mm256_storeu_pd( &gamma(0,3), gamma_0123_3 );
}
```

Figure 2.4.3: Partially instantiated $m_R \times n_R$ kernel for the case where $m_R = n_R = 4$ (see `Assignments/Week2/C/Gemm_4x4Kernel.c`).

An illustration of how Figure 2.4.2 and Figure 2.4.3 compute is found in Figure 2.4.4.

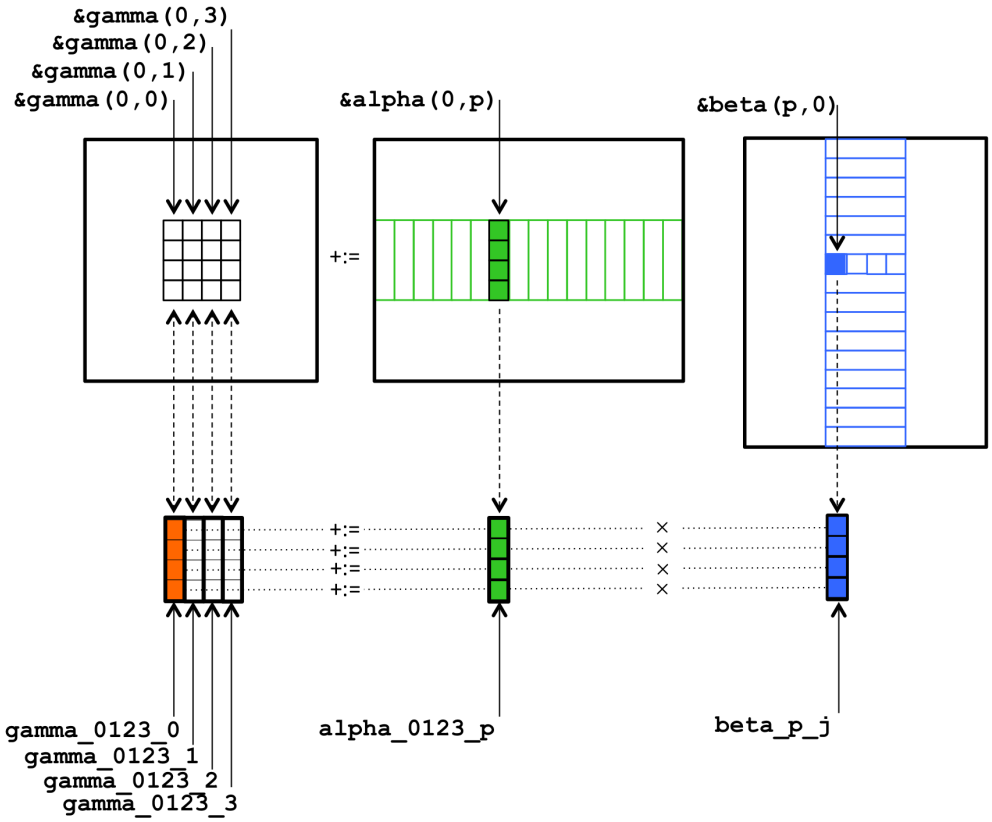


Figure 2.4.4: Illustration of how the routines in Figure 2.4.2 and Figure 2.4.3 indexes into the matrices.

2.4.3 Details

In this unit, we give details regarding the partial code in Figure 2.4.3 and illustrated in Figure 2.4.4.

To use the intrinsic functions, we start by including the header file `immintrin.h`.

```
#include <immintrin.h>
```

The declaration

```
__m256d gamma_0123_0 = _mm256_loadu_pd( &gamma( 0,0 ) );
```

creates `gamma_0123_0` as a variable that references a vector register with four double precision numbers and loads it with the four numbers that are stored starting at address

```
&gamma( 0,0 )
```

In other words, it loads that vector register with the original values

$$\begin{pmatrix} \gamma_{0,0} \\ \gamma_{1,0} \\ \gamma_{2,0} \\ \gamma_{3,0} \end{pmatrix}.$$

This is repeated for the other three columns of C:

```
__m256d gamma_0123_1 = _mm256_loadu_pd( &gamma( 0,1 ) );
__m256d gamma_0123_2 = _mm256_loadu_pd( &gamma( 0,2 ) );
__m256d gamma_0123_3 = _mm256_loadu_pd( &gamma( 0,3 ) );
```

The loop in Figure 2.4.2 implements

for $p = 0, \ldots, k-1$

$$\left(\begin{array}{c|c|c|c} \gamma_{0,0} += \alpha_{0,p} \times \beta_{p,0} & \gamma_{0,1} += \alpha_{0,p} \times \beta_{p,1} & \gamma_{0,2} += \alpha_{0,p} \times \beta_{p,2} & \gamma_{0,3} += \alpha_{0,p} \times \beta_{p,3} \\ \gamma_{1,0} += \alpha_{1,p} \times \beta_{p,0} & \gamma_{1,1} += \alpha_{1,p} \times \beta_{p,1} & \gamma_{1,2} += \alpha_{1,p} \times \beta_{p,2} & \gamma_{1,3} += \alpha_{1,p} \times \beta_{p,3} \\ \gamma_{2,0} += \alpha_{2,p} \times \beta_{p,0} & \gamma_{2,1} += \alpha_{2,p} \times \beta_{p,1} & \gamma_{2,2} += \alpha_{2,p} \times \beta_{p,2} & \gamma_{2,3} += \alpha_{2,p} \times \beta_{p,3} \\ \gamma_{3,0} += \alpha_{3,p} \times \beta_{p,0} & \gamma_{3,1} += \alpha_{3,p} \times \beta_{p,1} & \gamma_{3,2} += \alpha_{3,p} \times \beta_{p,2} & \gamma_{3,3} += \alpha_{3,p} \times \beta_{p,3} \end{array}\right)$$

end

leaving the result in the vector registers. Each iteration starts by declaring vector register variable `alpha_0123_p` and loading it with the contents of

$$\begin{pmatrix} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{pmatrix}.$$

```
__m256d alpha_0123_p = _mm256_loadu_pd( &alpha( 0,p ) );
```

Next, $\beta_{p,0}$ is loaded into a vector register, broadcasting (duplicating) that value to each entry in that register:

```
beta_p_j = _mm256_broadcast_sd( &beta( p, 0) );
```

This variable is declared earlier in the routine as `beta_p_j` because it is reused for $j = 0, 1, \ldots,$.

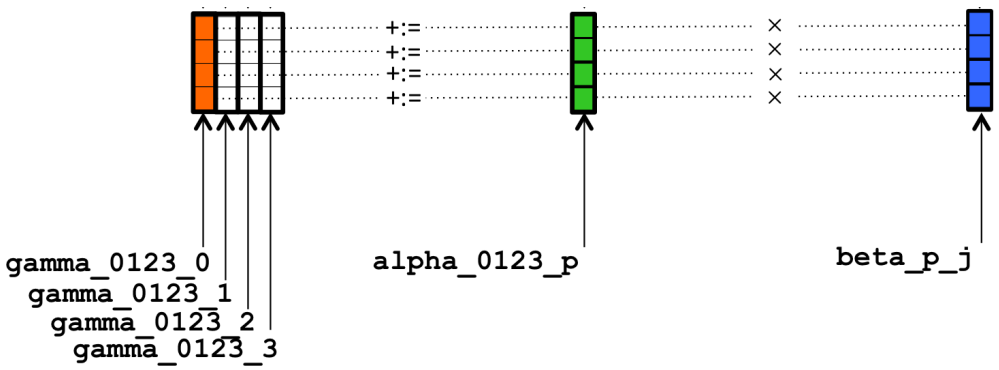
The command

```
gamma_0123_0 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_0 );
```

then performs the computation

$$\begin{pmatrix} \gamma_{0,0} += \alpha_{0,p} \times \beta_{p,0} \\ \gamma_{1,0} += \alpha_{1,p} \times \beta_{p,0} \\ \gamma_{2,0} += \alpha_{2,p} \times \beta_{p,0} \\ \gamma_{3,0} += \alpha_{3,p} \times \beta_{p,0} \end{pmatrix}$$

illustrated by



in Figure 2.4.3. Notice that we use `beta_p_j` for $\beta_{p,0}$ because that same vector register will be used for $\beta_{p,j}$ with $j = 0, 1, 2, 3$.

We leave it to the reader to add the commands that compute

$$\begin{pmatrix} \gamma_{0,1} += \alpha_{0,p} \times \beta_{p,1} \\ \gamma_{1,1} += \alpha_{1,p} \times \beta_{p,1} \\ \gamma_{2,1} += \alpha_{2,p} \times \beta_{p,1} \\ \gamma_{3,1} += \alpha_{3,p} \times \beta_{p,1} \end{pmatrix}, \begin{pmatrix} \gamma_{0,2} += \alpha_{0,p} \times \beta_{p,2} \\ \gamma_{1,2} += \alpha_{1,p} \times \beta_{p,2} \\ \gamma_{2,2} += \alpha_{2,p} \times \beta_{p,2} \\ \gamma_{3,2} += \alpha_{3,p} \times \beta_{p,2} \end{pmatrix}, \text{and} \begin{pmatrix} \gamma_{0,3} += \alpha_{0,p} \times \beta_{p,3} \\ \gamma_{1,3} += \alpha_{1,p} \times \beta_{p,3} \\ \gamma_{2,3} += \alpha_{2,p} \times \beta_{p,3} \\ \gamma_{3,3} += \alpha_{3,p} \times \beta_{p,3} \end{pmatrix}.$$

in `Assignments/Week2/C/Gemm_4x4Kernel.c`. Upon completion of the loop, the results are stored back into the original arrays with the commands

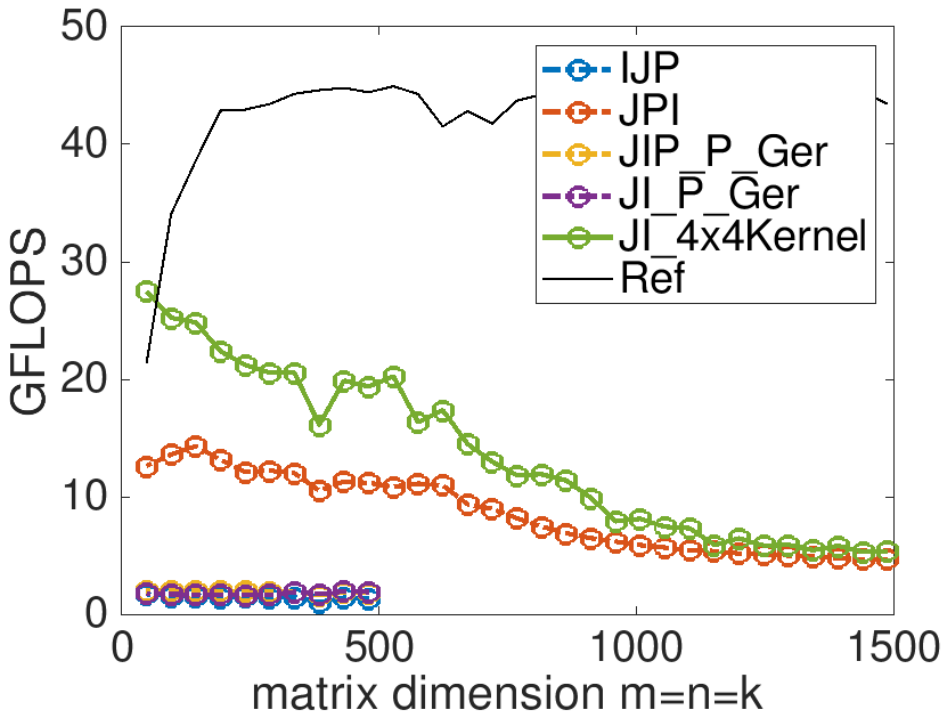
```
_mm256_storeu_pd( &gamma(0,0), gamma_0123_0 );
_mm256_storeu_pd( &gamma(0,1), gamma_0123_1 );
_mm256_storeu_pd( &gamma(0,2), gamma_0123_2 );
_mm256_storeu_pd( &gamma(0,3), gamma_0123_3 );
```

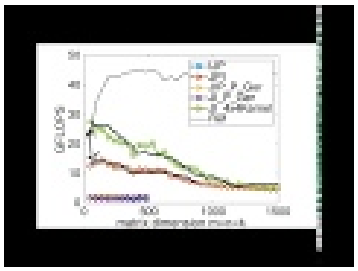
Homework 2.4.3.1 Complete the code in `Assignments/Week2/C/Gemm_4x4Kernel.c` and execute it with
`make JI_4x4Kernel`

View its performance with `Assignments/Week2/C/data/Plot_register_blocking.mlx`

Solution. `Assignments/Week2/Answers/Gemm_4x4Kernel.c`

This is the performance on Robert's laptop, with `MB=NB=KB=4`:





YouTube: https://www.youtube.com/watch?v=hUW--tJPPcw

Remark 2.4.5 The form of parallelism illustrated here is often referred to as Single Instruction, Multiple Data (SIMD) parallelism since the same operation (an FMA) is executed with multiple data (four values of C and A and a duplicated value from B).

We are starting to see some progress towards higher performance

2.4.4 More options

You have now experienced the fact that modern architectures have vector registers and how to (somewhat) optimize the update of a 4×4 submatrix of C with vector instructions.

In the implementation in Unit 2.4.2, the block of C kept in registers is 4×4. In general, the block is $m_R \times n_R$. If we assume we will use R_C registers for elements of the submatrix of C, what should m_R and n_R be to attain the best performance?

A big part of optimizing is to amortize the cost of moving data over useful computation. In other words, we want the ratio between the number of flops that are performed to the number of data that are moved between, for now, registers and memory to be high.

Let's recap: Partitioning

$$C = \left(\begin{array}{c|c|c} C_{0,0} & \cdots & C_{0,N-1} \\ \hline \vdots & & \vdots \\ \hline C_{M-1,0} & \cdots & C_{M-1,N-1} \end{array}\right),$$

$$A = \left(\begin{array}{c} A_0 \\ \hline \vdots \\ \hline A_{M-1} \end{array}\right), \quad B = \left(\begin{array}{c|c|c} B_0 & \cdots & B_{N-1} \end{array}\right),$$

the algorithm we discovered in Section 2.3 is given by

for $i := 0, \ldots, M-1$
  for $j := 0, \ldots, N-1$
    Load $C_{i,j}$ into registers
    $C_{i,j} := A_i B_j + C_{i,j}$ with micro − kernel
    Store $C_{i,j}$ to memory
  end
end

We analyzed that its cost is given by

$$2mnk\gamma_R + \left[2mn + mnk(\frac{1}{n_R} + \frac{1}{m_R})\right]\beta_{R\leftrightarrow M}.$$

The ratio between flops and memory operations between the registers and memory is then

$$\frac{2mnk}{2mn + mnk(\frac{1}{n_R} + \frac{1}{m_R})}$$

If k is large, then $2mn$ (the cost of loading and storing the $m_R \times n_R$ submatrices of C) can be ignored in the denominator, yielding, approximately,

$$\frac{2mnk}{mnk(\frac{1}{n_R} + \frac{1}{m_R})} = \frac{2}{\frac{1}{n_R} + \frac{1}{m_R}} = \frac{2}{\frac{m_R}{m_R n_R} + \frac{n_R}{m_R n_R}} = \frac{2m_R n_R}{m_R + n_R}.$$

This is the ratio of floating point operations to memory operations that we want to be high.

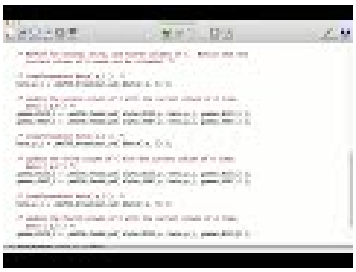
If $m_R = n_R = 4$ then this ratio is 4. For every memory operation (read) of an element of A or B, approximately 4 floating point operations are performed with data that resides in registers.

Homework 2.4.4.1 Modify `Assignments/Week2/C/Gemm_4x4Kernel.c` to implement the case where $m_R = 8$ and $n_R = 4$, storing the result in `Gemm_8x4Kernel.c`. You can test the result by executing
`make JI_8x4Kernel`

in that directory. View the resulting performance by appropriately modifying `Assignments/Week2/C/data/Plot_optimize_MRxNR.mlx`.

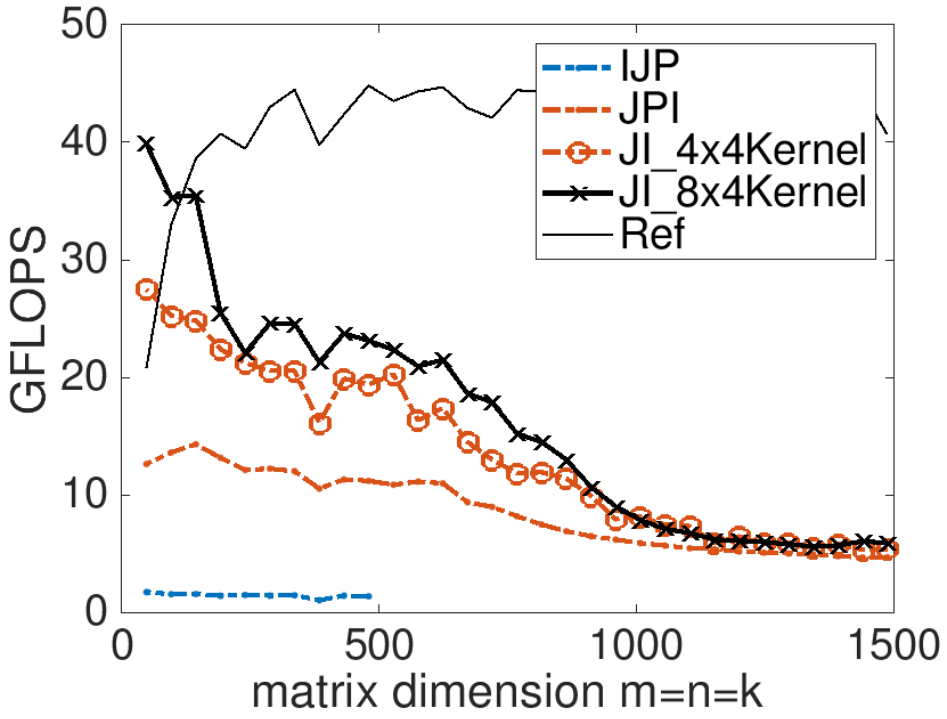
Hint. A vector register can only hold four doubles... Each column now has four doubles. How many vector registers do you need to store a column?

Solution.



YouTube: `https://www.youtube.com/watch?v=aDFpX3PsfPA`

Assignments/Week2/Answers/Gemm_8x4Kernel.c.



Homework 2.4.4.2 Consider the implementation in Homework 2.4.4.1 with $m_R \times n_R = 8 \times 4$.

- How many vector registers are needed?
- Ignoring the cost of loading the registers with the 8×4 submatrix of C, what is the ratio of flops to loads?

Solution. Number of registers:

$$8 + 3 = 11$$

Ratio:

$$64 \text{ flops}/12 \text{ loads} \approx 5.33 \text{ flops/load}.$$

Homework 2.4.4.3 We have considered $m_R \times n_R = 4 \times 4$ and $m_R \times n_R = 8 \times 4$, where elements of A are loaded without duplication into vector registers (and hence m_R must be a multiple of 4), and elements of B are loaded/broadcast. Extending this approach to loading A and B, complete the entries in the fol-

lowing table:

	# of vector regs	flops/load
4×1		/ =
4×2		/ =
4×4	6	$32/8 = 4$
4×8		/ =
4×12		/ =
4×14		/ =
8×1		/ =
8×2		/ =
8×4		/ =
8×6		/ =
8×8		/ =
12×1		/ =
12×2		/ =
12×4		/ =
12×6		/ =

Solution.

	# of vector regs	flops/load
4×1	3	$8/5 = 1.60$
4×2	4	$16/6 = 2.66$
4×4	6	$32/8 = 4.00$
4×8	10	$64/12 = 5.33$
4×12	14	$96/16 = 6.00$
4×14	16	$112/18 = 6.22$
8×1	5	$16/9 = 1.78$
8×2	7	$32/10 = 3.20$
8×4	11	$64/12 = 5.33$
8×6	15	$96/14 = 6.86$
8×8	19	$128/16 = 8.00$
12×1	7	$24/13 = 1.85$
12×2	10	$48/14 = 3.43$
12×4	16	$96/16 = 6.00$
12×6	22	$144/18 = 8.00$

Remark 2.4.6 Going forward, keep in mind that the cores of the architectures we target only have 16 vector registers that can store four doubles each.

Homework 2.4.4.4 At this point, you have already implemented the following kernels: `Gemm_4x4Kernel.c` and `Gemm_8x4Kernel.c`. Implement as many of the more promising of the kernels you analyzed in the last homework as you like. Your implementations can be executed by typing
`make JI_?x?Kernel`

where the ?'s are replaced with the obvious choices of m_R and n_R. The resulting performance can again be viewed with Live Script in `Assignments/Week2/C/data/Plot_optimize_MRxNR.mlx`.

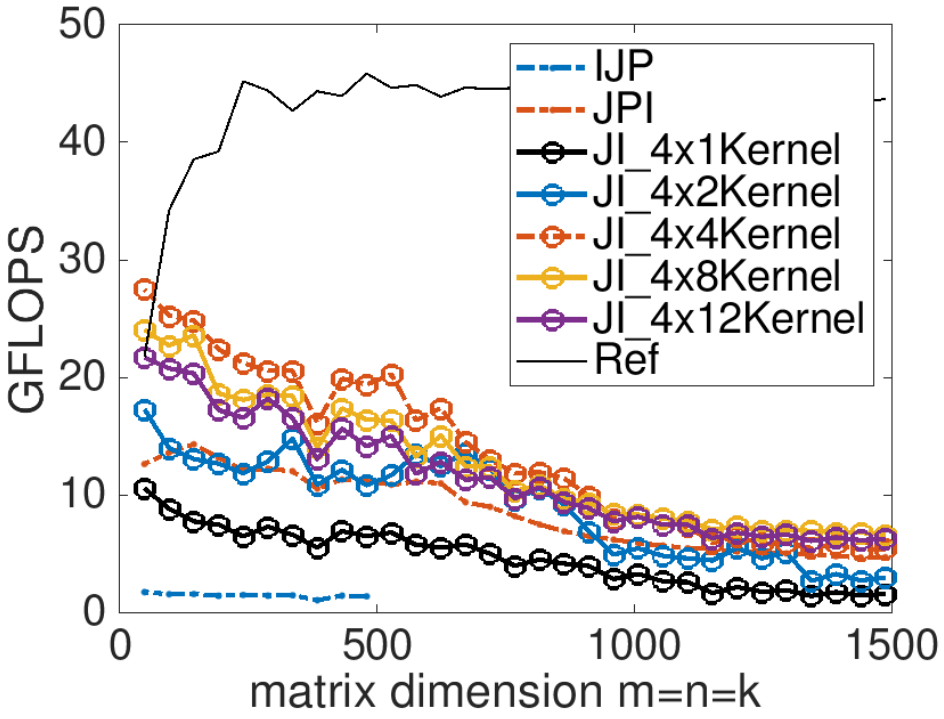
Solution.

- Assignments/Week2/Answers/Gemm_4x1Kernel.c
- Assignments/Week2/Answers/Gemm_4x2Kernel.c
- Assignments/Week2/Answers/Gemm_4x4Kernel.c
- Assignments/Week2/Answers/Gemm_4x8Kernel.c
- Assignments/Week2/Answers/Gemm_4x12Kernel.c
- Assignments/Week2/Answers/Gemm_4x14Kernel.c

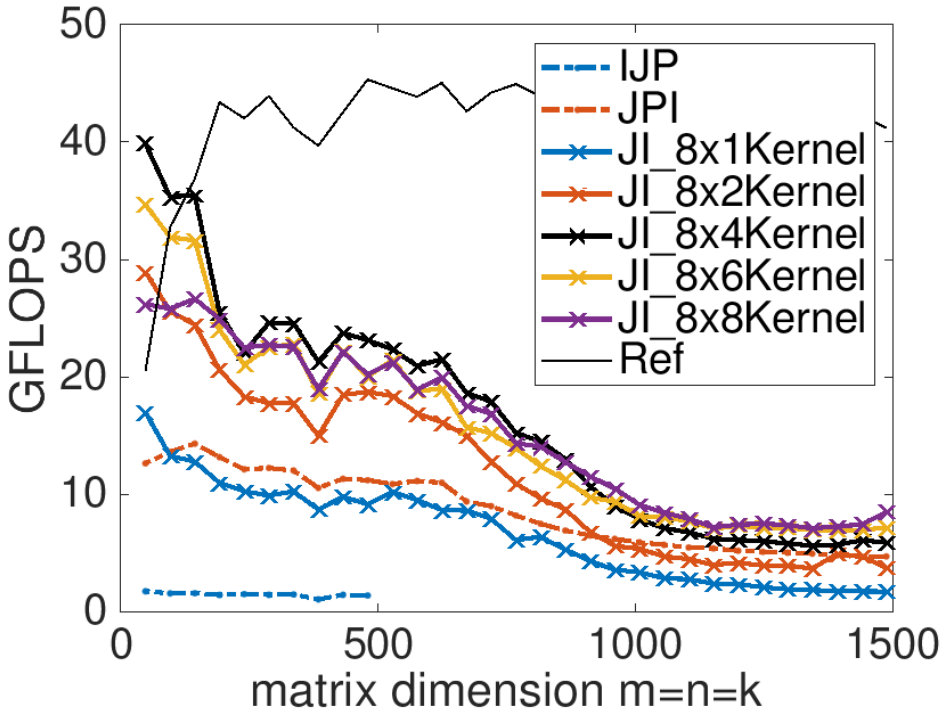
- Assignments/Week2/Answers/Gemm_8x1Kernel.c
- Assignments/Week2/Answers/Gemm_8x2Kernel.c
- Assignments/Week2/Answers/Gemm_8x4Kernel.c
- Assignments/Week2/Answers/Gemm_8x6Kernel.c
- Assignments/Week2/Answers/Gemm_8x8Kernel.c

- Assignments/Week2/Answers/Gemm_12x1Kernel.c
- Assignments/Week2/Answers/Gemm_12x2Kernel.c
- Assignments/Week2/Answers/Gemm_12x4Kernel.c
- Assignments/Week2/Answers/Gemm_12x6Kernel.c

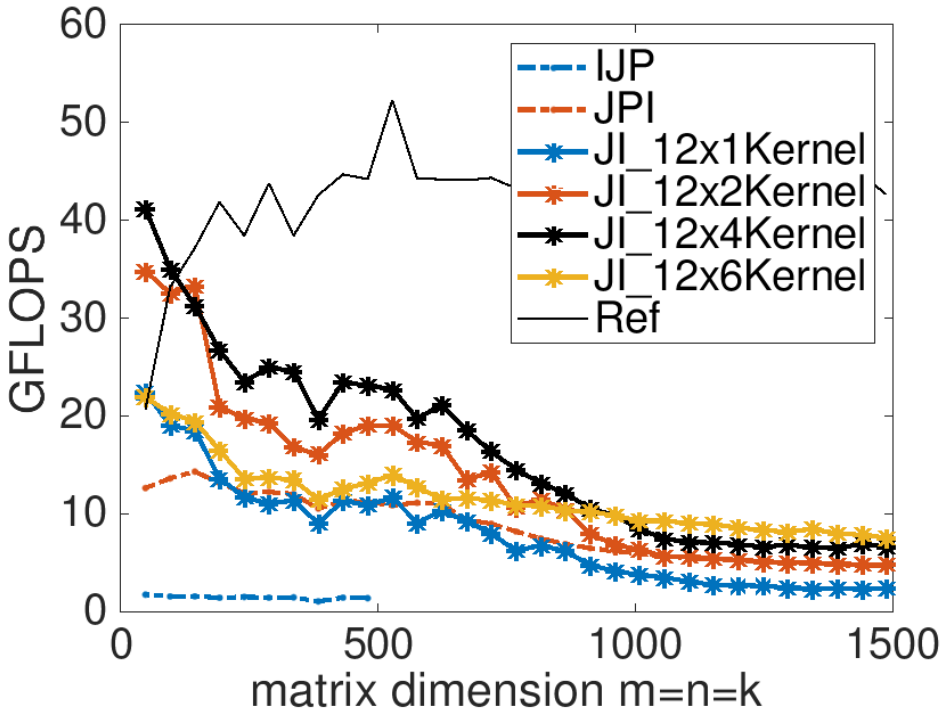
On Robert's laptop:
Performance of implementations for $m_R = 4$ and various choices for n_R:



Performance of implementations for $m_R = 8$ and various choices for n_R:



Performance of implementations for $m_R = 12$ and various choices for n_R:



2.4.5 Optimally amortizing data movement

Homework 2.4.5.1 In the discussion in the last unit, assume that a fixed number of elements in $m_R \times n_R$ submatrix $C_{i,j}$ can be stored in registers: $m_R n_R = K$. What choices of m_R and n_R maximize

$$\frac{2m_R n_R}{m_R + n_R},$$

the ratio of useful operations (floating point operations) to overhead (memory operations)?

Of course, our approach so far restricts m_R to be an integer multiple of the vector length, so that adds another contraint. Let's not worry about that.

Hint. You want to maximize

$$\frac{2m_R n_R}{m_R + n_R},$$

under the constraint that $m_R n_R = K$.

If you took a course in calculus, you may have been asked to maximize xy under the constraint that $2(x + y)$ is constant. It might have been phrased as "You have a fixed amount of fencing to enclose a rectangular area. How should you pick the length, x, and width y of the rectangle?" If you think about it, the answer to this homework is closely related to this question about fencing.

Answer.

$$m_R = n_R = \sqrt{K}.$$

Solution. Let's use x for m_R and y for n_R. We want to find x and y that maximize

$$\frac{2K}{x + y}$$

under the constrant that $xy = K$. This is equivalent to finding x and y that minimize $x + y$ under the constraint that $xy = K$.

Letting $y = K/x$ we find that we need to minimize

$$f(x) = x + \frac{K}{x}.$$

By setting the derivative to zero, we find that the desired x must satisfy

$$1 - \frac{K}{x^2} = 0.$$

Manipulating this yields

$$x^2 = K$$

and hence $x = \sqrt{K}$ and $y = K/x = \sqrt{K}$.

Strictly speaking, you should to check that it is a minimum of $f(x)$ by examining the second derivative and checking that it is positive at the extremum.

Remark 2.4.7 In practice, we have seen that the shape of the block of C kept

in registers is not square for a number of reasons:

- The formula for the number of vector registers that are used, when the vector length is four, is

$$\frac{m_R}{4} n_R + \frac{m_R}{4} + 1.$$

  Even if the number of registers is a "nice number", it may not be optimal for m_R to equal n_R.

- For now, m_R has to be a multiple of the vector length. Later we will see that we could instead choose n_R to be a multiple of the vector length. Regardless, this limits the choices.

- The (load and) broadcast may be a more expensive operation than the load, on a per double that is loaded basis.

- There are other issues that have to do with how instructions are pipelined that are discussed in a paper mentioned in Unit 2.5.1.

2.5 Enrichments

2.5.1 Lower bound on data movement

The discussion in this enrichment was inspired by the paper [23]

Tyler Michael Smith, Bradley Lowery, Julien Langou, Robert A. van de Geijn. Tight I/O Lower Bound for Matrix Multiplication. Submitted to ACM Transactions on Mathematical Software. Draft available from arXiv.org.

For more details, we encourage you to read that paper.

2.5.1.1 Reasoning about optimality

Early in our careers, we learned that if you say that an implementation is optimal, you better prove that it is optimal.

In our empirical studies, graphing the measured performance, we can compare our achieved results to the theoretical peak. Obviously, if we achieved theoretical peak performance, then we would know that the implementation is optimal. The problem is that we rarely achieve the theoretical peak performance as computed so far (multiplying the clock rate by the number of floating point operations that can be performed per clock cycle).

In order to claim optimality, one must carefully model an architecture and compute, through analysis, the exact limit of what it theoretically can achieve. Then, one can check achieved performance against the theoretical limit, and make a claim. Usually, this is also not practical.

In practice, one creates a model of computation for a simplified architecture. With that, one then computes a theoretical limit on performance. The next step is to show that the theoretical limit can be (nearly) achieved by an algorithm that executes on that simplified architecture. This then says something about the optimality of the algorithm under idealized circumstances. By

finally comparing and contrasting the simplified architecture with an actual architecture, and the algorithm that targets the simplified architecture with an actual algorithm designed for the actual architecture, one can reason about the optimality, or lack thereof, of the practical algorithm.

2.5.1.2 A simple model

Let us give a simple model of computation that matches what we have assumed so far when programming matrix-matrix multiplication:

- We wish to compute $C := AB + C$ where C, A, and C are $m \times n$, $m \times k$, and $k \times n$, respectively.
- The computation is cast in terms of FMAs.
- Our machine has two layers of memory: fast memory (registers) and slow memory (main memory).
- Initially, data reside in main memory.
- To compute a FMA, all three operands must be in fast memory.
- Fast memory can hold at most S floats.
- Slow memory is large enough that its size is not relevant to this analysis.
- Computation cannot be overlapped with data movement.

Notice that this model matches pretty well how we have viewed our processor so far.

2.5.1.3 Minimizing data movement

We have seen that matrix-matrix multiplication requires $m \times n \times k$ FMA operations, or $2mnk$ flops. Executing floating point operations constitutes useful computation. Moving data between slow memory and fast memory is overhead since we assume it cannot be overlapped. Hence, under our simplified model, if an algorithm only performs the minimum number of flops (namely $2mnk$), minimizes the time spent moving data between memory layers, and we at any given time are either performing useful computation (flops) or moving data, then we can argue that (under our model) the algorithm is optimal.

We now focus the argument by reasoning about a lower bound on the number of data that must be moved from fast memory to slow memory. We will build the argument with a sequence of observations.

Consider the loop

for $p := 0, \ldots, k-1$
  for $j := 0, \ldots, n-1$
    for $i := 0, \ldots, m-1$
      $\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$
    end
  end
end

One can view the computations

$$\gamma_{i,j} := \alpha_{i,p}\beta_{p,j} + \gamma_{i,j}$$

as a cube of points in 3D, (i, j, k) for $0 \leq i < m$, $0 \leq j < n$, $0 \leq p < k$. The set of all possible algorithms that execute each such update only once can be viewed as an arbitrary ordering on that set of points. We can this as indexing the set of all such triples with i_r, j_r, p_r, $0 \leq r < m \times n \times k$:

$$(i_r, j_r, k_r).$$

so that the algorithm that computes $C := AB + C$ can then be written as

for $r := 0, \ldots, mnk - 1$
  $\gamma_{i_r,j_r} := \alpha_{i_r,p_r}\beta_{p_r,j_r} + \gamma_{i_r,j_r}$
end

Obviously, this puts certain restrictions on i_r, j_r, and p_r. Articulating those exactly is not important right now.

We now partition the ordered set $0, \ldots, mnk - 1$ into ordered contiguous subranges (phases) each of which requires $S + M$ distinct elements from A, B, and C (e.g., S elements from A, $M/3$ elements of B, and $2M/3$ elements of C), except for the last phase, which will contain fewer. (Strictly speaking, it is a bit more complicated than splitting the range of the iterations, since the last FMA may require anywhere from 0 to 3 new elements to be loaded from slow memory. Fixing this is a matter of thinking of the loads that are required as separate from the computation (as our model does) and then splitting the operations - loads, FMAs, and stores - into phases rather than the range. This does not change our analysis.)

Recall that S equals the number of floats that fit in fast memory. A typical phase will start with S elements in fast memory, and will in addition read M elements from slow memory (except for the final phase). % We will call such an ordered subset a phase of triples. Such a typical phase will start at $r = R$ and consists of F triples, (i_R, j_R, p_R) through $(i_{R+F-1}, j_{R+F-1}, p_{R+F-1})$. Let us denote the set of these triples by $\mathbf{D}$. These represent F FMAs being performed in our algorithm. Even the first phase needs to read at least M elements since it will require $S + M$ elements to be read.

The key question now becomes what the upper bound on the number of FMAs is that can be performed with $S + M$ elements. Let us denote this bound by $F_{\max}$. If we know $F_{\max}$ as a function of $S + M$, then we know that at least $\frac{mnk}{F_{\max}} - 1$ phases of FMAs need to be executed, where each of those phases requires at least S reads from slow memory. The total number of reads required for any algorithm is thus at least

$$\left(\frac{mnk}{F_{\max}} - 1\right) M. \tag{2.5.1}$$

To find $F_{\max}$ we make a few observations:

- A typical triple $(i_r, j_r, p_r) \in \mathbf{D}$ represents a FMA that requires one element from each of matrices C, A, and B: $\gamma_{i,j}$, $\alpha_{i,p}$, and $\beta_{p,j}$.

- The set of all elements from C, γ_{i_r,j_r}, that are needed for the computations represented by the triples in $\mathbf{D}$ are indexed with the tuples $\mathbf{C_D} = \{(i_r, j_r) \mid R \leq r < R + F\}$. If we think of the triples in $\mathbf{D}$ as points in 3D, then $\mathbf{C_D}$ is the projection of those points onto the i, j plane. Its size, $|\mathbf{C_D}|$, tells us how many elements of C must at some point be in fast memory during that phase of computations.

- The set of all elements from A, α_{i_r,p_r}, that are needed for the computations represented by the triples in $\mathbf{D}$ are indexed with the tuples $\mathbf{A_D} = \{(i_r, p_r) \mid R \leq r < R + F\}$. If we think of the triples (i_r, j_r, p_r) as points in 3D, then $\mathbf{A_D}$ is the projection of those points onto the i, p plane. Its size, $|\mathbf{A_D}|$, tells us how many elements of A must at some point be in fast memory during that phase of computations.

- The set of all elements from C, β_{p_r,j_r}, that are needed for the computations represented by the triples in $\mathbf{D}$ are indexed with the tuples $\mathbf{B_D} = \{(p_r, j_r) \mid R \leq r < R + F\}$. If we think of the triples (i_r, j_r, p_r) as points in 3D, then $\mathbf{B_D}$ is the projection of those points onto the p, j plane. Its size, $|\mathbf{B_D}|$, tells us how many elements of B must at some point be in fast memory during that phase of computations.

Now, there is a result known as the discrete Loomis-Whitney inequality that tells us that in our situation $|\mathbf{D}| \leq \sqrt{|\mathbf{C_D}||\mathbf{A_D}||\mathbf{B_D}|}$. In other words, $F_{\max} \leq \sqrt{|\mathbf{C_D}||\mathbf{A_D}||\mathbf{B_D}|}$. The name of the game now becomes to find the largest value $F_{\max}$ that satisfies

$$\text{maximize } F_{\max} \text{ such that } \begin{cases} F_{\max} \leq \sqrt{|\mathbf{C_D}||\mathbf{A_D}||\mathbf{B_D}|} \\ |\mathbf{C_D}| > 0, |\mathbf{A_D}| > 0, |\mathbf{B_D}| > 0 \\ |\mathbf{C_D}| + |\mathbf{A_D}| + |\mathbf{B_D}| = S + M. \end{cases}$$

An application known as Lagrange multipliers yields the solution

$$|\mathbf{C_D}| = |\mathbf{A_D}| = |\mathbf{B_D}| = \frac{S+M}{3} \quad \text{and} \quad F_{\max} = \frac{(S+M)\sqrt{S+M}}{3\sqrt{3}}.$$

With that largest $F_{\max}$ we can then establish a lower bound on the number of memory reads given by (2.5.1):

$$\left(\frac{mnk}{F_{\max}} - 1\right) M = \left(3\sqrt{3}\frac{mnk}{(S+M)\sqrt{S+M}} - 1\right) M.$$

Now, M is a free variable. To come up with the sharpest (best) lower bound, we want the largest lower bound. It turns out that, using techniques from calculus, one can show that $M = 2S$ maximizes the lower bound. Thus, the best lower bound our analysis yields is given by

$$\left(3\sqrt{3}\frac{mnk}{(3S)\sqrt{3S}} - 1\right)(2S) = 2\frac{mnk}{\sqrt{S}} - 2S.$$

2.5.1.4 A nearly optimal algorithm

We now discuss a (nearly) optimal algorithm for our simplified architecture. Recall that we assume fast memory can hold S elements. For simplicity, assume S is a perfect square. Partition

$$C = \begin{pmatrix} C_{0,0} & C_{0,1} & \cdots \\ C_{1,0} & C_{1,1} & \cdots \\ \vdots & \vdots & \end{pmatrix}, \quad A = \begin{pmatrix} A_0 \\ A_1 \\ \vdots \end{pmatrix}, \quad \text{and} \quad B = \begin{pmatrix} B_0 & B_1 & \cdots \end{pmatrix}.$$

where $C_{i,j}$ is $m_R \times n_R$, A_i is $m_R \times k$, and B_j is $k \times n_R$. Here we choose $m_R \times n_R = (\sqrt{S}-1) \times \sqrt{S}$ so that fast memory can hold one submatrix $C_{i,j}$, one column of A_i, and one element of B_j: $m_R \times n_R + m_R + 1 = (\sqrt{S}-1) \times \sqrt{S} + \sqrt{S} - 1 + 1 = S$.

When computing $C := AB + C$, we recognize that $C_{i,j} := A_i B_j + C_{i,j}$. Now, let's further partition

$$A_i = \begin{pmatrix} a_{i,0} & a_{i,1} & \cdots \end{pmatrix} \quad \text{and} \quad B_j = \begin{pmatrix} b_{0,j}^T \\ b_{1,j}^T \\ \vdots \end{pmatrix}.$$

We now recognize that $C_{i,j} := A_i B_j + C_{i,j}$ can be computed as

$$C_{i,j} := a_{i,0} b_{0,j}^T + a_{i,1} b_{1,j}^T + \cdots,$$

the by now very familiar sequence of rank-1 updates that makes up the micro-kernel discussed in Unit 2.4.1. The following loop exposes the computation $C := AB + C$, including the loads and stores from and to slow memory:

for $j := 0, \ldots, N-1$
  for $i := 0, \ldots, M-1$
    Load $C_{i,j}$ into fast memory
    for $p := 0, \ldots, k-1$
      Load $a_{i,p}$ and $b_{p,j}^T$ into fast memory
      $C_{i,j} := a_{i,p} b_{p,j}^T + C_{i,j}$
    end
    Store $C_{i,j}$ to slow memory
  end
end

For simplicity, here $M = m/m_r$ and $N = n/n_r$.

On the surface, this seems to require $C_{i,j}$, $a_{i,p}$, and $b_{p,j}^T$ to be in fast memory at the same time, placing $m_R \times n_R + m_R + n_r = S + \sqrt{S} - 1$ floats in fast memory. However, we have seen before that the rank-1 update $C_{i,j} := a_{i,p} b_{p,j}^T + C_{i,j}$ can be implemented as a loop around `axpy` operations, so that the elements of $b_{p,j}^T$ only need to be in fast memory one at a time.

Let us now analyze the number of memory operations incurred by this algorithm:

- Loading and storing all $C_{i,j}$ incurs mn loads and mn stores, for a total of $2mn$ memory operations. (Each such block is loaded once and stored once, meaning every element of C is loaded once and stored once.)

- Loading all $a_{i,p}$ requires

$$MNkm_R = (Mm_R)Nk = m\frac{n}{n_R}k = \frac{mnk}{\sqrt{S}}$$

  memory operations.

- Loading all $b^T_{p,j}$ requires

$$MNkn_R = M(Nn_R)k = \frac{m}{m_R}nk = \frac{mnk}{\sqrt{S}-1}$$

  memory operations.

The total number of memory operations is hence

$$2mn + \frac{mnk}{\sqrt{S}} + \frac{mnk}{\sqrt{S}-1} = 2\frac{mnk}{\sqrt{S}} + 2mn + \frac{mnk}{S-\sqrt{S}}.$$

We can now compare this to the lower bound from the last unit:

$$2\frac{mnk}{\sqrt{S}} - 2S.$$

The cost of reading and writing elements of C, $2mn$, contributes a lower order term, as does $\frac{mnk}{S-\sqrt{S}}$ if S (the size of fast memory) is reasonably large. Thus, the proposed algorithm is nearly optimal with regards to the amount of data that is moved between slow memory and fast memory.

2.5.1.5 Discussion

What we notice is that the algorithm presented in the last unit is quite similar to the algorithm that in the end delivered good performance in [23]. It utilizes most of fast memory (registers in [23]) with a submatrix of C. Both organize the computation in terms of a kernel that performs rank-1 updates of that submatrix of C.

The theory suggests that the number of memory operations are minimized if the block of C is chosen to be (roughly) square. In Unit 2.4.1, the best performance was observed with a kernel that chose the submatrix of C in registers to be 8×6, so for this architecture, the theory is supported by practice. For other architectures, there may be issues that skew the aspect ratio to be less square.

2.5.2 Strassen's algorithm

So far, we have discussed algorithms for computing $C := AB + C$ via a triple-nested loop that then perform $2mnk$ flops. A question is whether this is the best we can do.

A classic result regarding this is Strassen's algorithm [26]. It shows that, for problems of size $m = n = k = 2^r$ for some integer r, matrix-matrix multiplication can be computed in time $O(n^{\log_2 7}) \approx O(n^{2.807})$. Since Strassen

proposed this, a succession of results further improved upon the exponent. In this discussion, we stick to the original result.

How can this be? For simplicity, assume $m = n = k$ are all even. Partition

$$C = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix}, A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}, B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix},$$

where X_{ij} are $n/2 \times n/2$ for $X \in \{C, A, B\}$ and $ij \in \{00, 01, 10, 11\}$. Now that you understand how partitioned matrix-matrix multiplication works, you know that the following computations compute $C := AB + C$:

$$\begin{array}{rcl} C_{00} & = & \alpha(A_{00}B_{00} + A_{01}B_{10}) + C_{00} \\ C_{01} & = & \alpha(A_{00}B_{01} + A_{01}B_{11}) + C_{01} \\ C_{10} & = & \alpha(A_{10}B_{00} + A_{11}B_{10}) + C_{10} \\ C_{11} & = & \alpha(A_{10}B_{01} + A_{11}B_{11}) + C_{11}. \end{array}$$

Each of the eight matrix-matrix multiplications requires $2(n/2)^3 = 1/4n^3$ flops and hence the total cost is our usual $2n^3$ flops.

Surprisingly, the following computations also compute $C := AB + C$:

$$\begin{array}{ll} M_0 = (A_{00} + A_{11})(B_{00} + B_{11}); & C_{00} \mathrel{+}= M_0; C_{11} \mathrel{+}= M_0; \\ M_1 = (A_{10} + A_{11})B_{00}; & C_{10} \mathrel{+}= M_1; C_{11} \mathrel{-}= M_1; \\ M_2 = A_{00}(B_{01} - B_{11}); & C_{01} \mathrel{+}= M_2; C_{11} \mathrel{+}= M_2; \\ M_3 = A_{11}(B_{10} - B_{00}); & C_{00} \mathrel{+}= M_3; C_{10} \mathrel{+}= M_3; \\ M_4 = (A_{00} + A_{01})B_{11}; & C_{01} \mathrel{+}= M_4; C_{00} \mathrel{-}= M_4; \\ M_5 = (A_{10} - A_{00})(B_{00} + B_{01}); & C_{11} \mathrel{+}= M_5; \\ M_6 = (A_{01} - A_{11})(B_{10} + B_{11}); & C_{00} \mathrel{+}= M_6. \end{array}$$

If you count carefully, this requires 22 additions of $n/2 \times n/2$ matrices and 7 multiplications with $n/2 \times n/2$ matrices. Adding matrices together requires $O(n^2)$ flops, which are insignificant if n is large. Ignoring this, the cost now becomes

$$7 \times 2(n/2)^3 = 2\frac{7}{8}n^3$$

flops. The cost of the matrix-matrix multiplication is now 7/8 of what it was before!

But it gets better! Each of the matrix-matrix multiplications can themselves be computed via this scheme. If you do that, applying the idea at two levels, the cost is reduced to

$$7 \times (7 \times 2(n/4)^3) = 2\left(\frac{7}{8}\right)^2 n^3$$

flops. How many times can we do that? If $n = 2^r$ we can half the size of the matrices $r = \log_2(n)$ times. If you do that, the cost becomes

$$2\left(\frac{7}{8}\right)^r n^3 \text{ flops.}$$

Now,

$$(7/8)^{\log_2(n)}\, 2n^3 = n^{\log_2(7/8)} 2n^3 = 2n^{\log_2 7} \approx 2n^{2.807}.$$

Here we ignored the cost of the additions. However, it can be analyzed that this recursive approach requires $O(n^{2.807})$ flops.

Remark 2.5.1 Learn more about the Strassen's algorithm entry on Wikipedia. In Unit 3.5.4 we will discuss how our insights support a practical implementation of Strassen's algorithm.

2.6 Wrap Up

2.6.1 Additional exercises

If you want to get more practice, you may want to repeat the exercises in this week with single precision arithmetic. Note: this sounds like a simple suggestion. However, it requires changes to essentially all programming assignments you have encountered so far. This is a lot of work, to be pursued by only those who are really interested in the details of this course.

2.6.2 Summary

2.6.2.1 The week in pictures

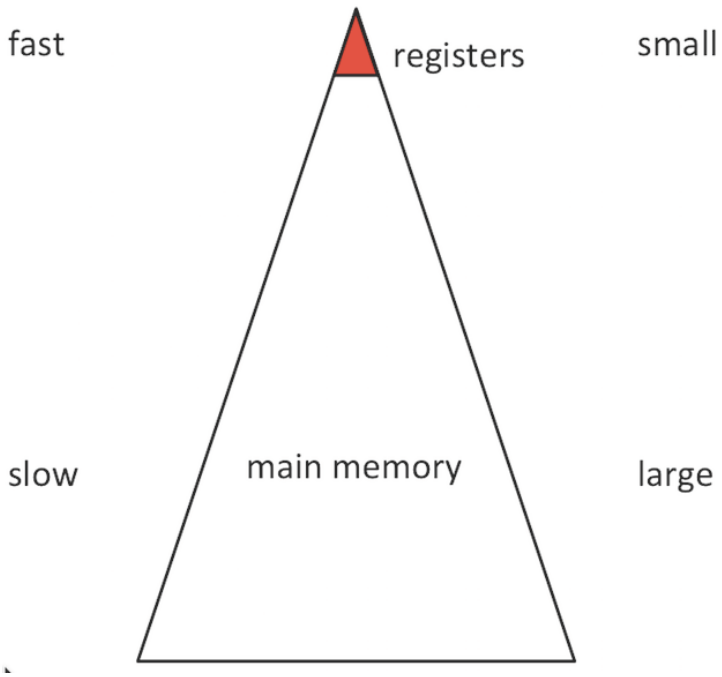


Figure 2.6.1: A simple model of the memory hierarchy, with registers and main memory.

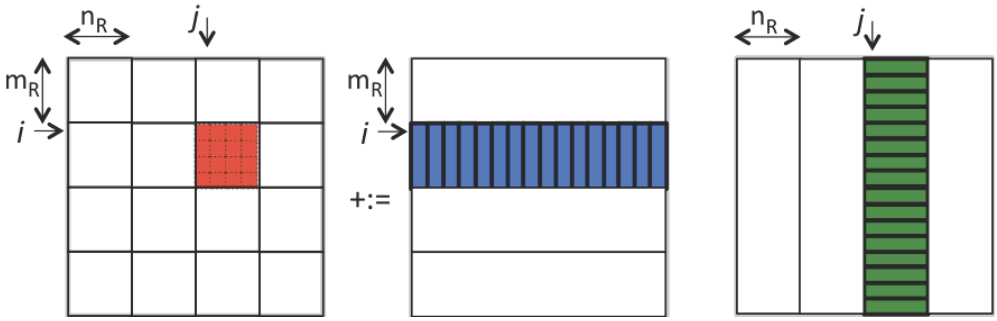


Figure 2.6.2: A simple blocking for registers, where micro-tiles of C are loaded into registers.

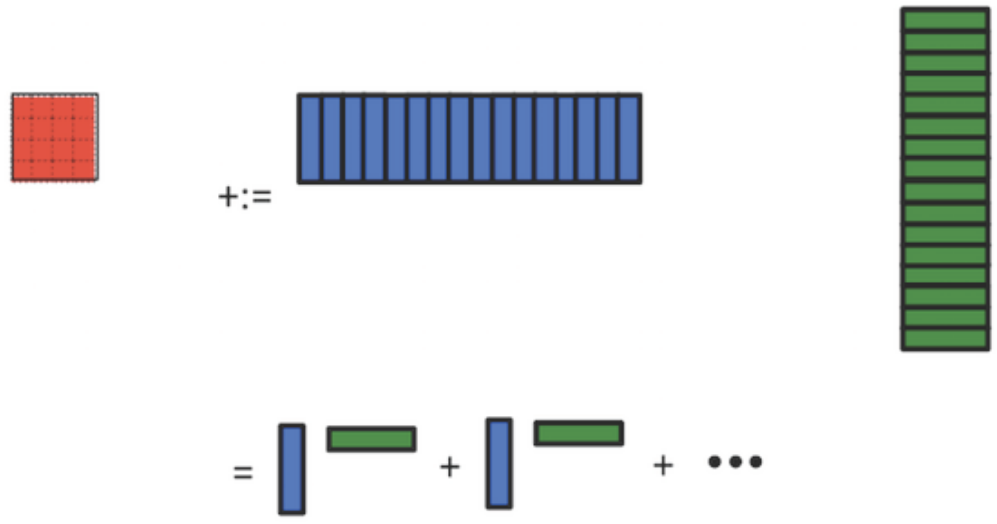


Figure 2.6.3: The update of a micro-tile with a sequence of rank-1 updates.

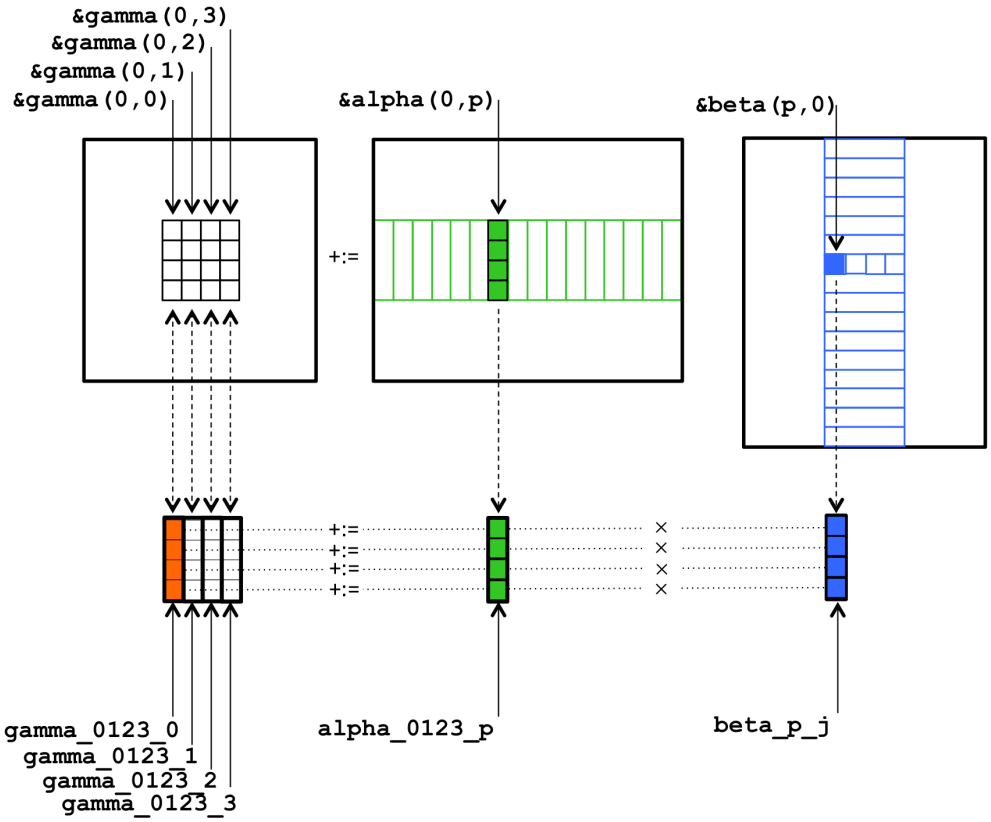


Figure 2.6.4: Mapping the micro-kernel to registers.

2.6.2.2 Useful intrinsic functions

From Intel's Intrinsics Reference Guide

- `__m256d _mm256_loadu_pd (double const * mem_addr)`

  Description

  Load 256-bits (composed of 4 packed double-precision (64-bit) floating-point elements) from memory into `dst` (output). `mem_addr` does not need to be aligned on any particular boundary.

- `__m256d _mm256_broadcast_sd (double const * mem_addr)`

  Description

  Broadcast a double-precision (64-bit) floating-point element from memory to all elements of `dst` (output).

- `__m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c)`

  Description

  Multiply packed double-precision (64-bit) floating-point elements in `a` and `b`, add the intermediate result to packed elements in `c`, and store the results in `dst` (output).

Week 3

Pushing the Limits

3.1 Opening Remarks

3.1.1 Launch

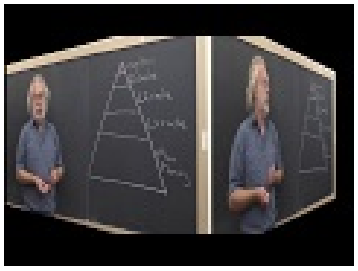
Remark 3.1.1 As you may have noticed, some of the programming assignments are still in flux. This means

- You will want to do

  ```
  git stash save
  git pull
  git stash pop
  ```

  in your LAFF-On-PfHP directory.

- You will want to upload the .mlx files form LAFF-On-PfHP.Assignments/Week3/C/data/ to the corresponding folder of Matlab Online.



YouTube: `https://www.youtube.com/watch?v=kOBCe3-B1BI`

The inconvenient truth is that floating point computations can be performed very fast while bringing data in from main memory is relatively slow. How slow? On a typical architecture it takes two orders of magnitude more time to bring a floating point number in from main memory than it takes to compute with it.

The reason why main memory is slow is relatively simple: there is not enough room on a chip for the large memories that we are accustomed to and hence they are off chip. The mere distance creates a latency for retrieving the data. This could then be offset by retrieving a lot of data simultaneously, increasing bandwidth. Unfortunately there are inherent bandwidth limitations:

there are only so many pins that can connect the central processing unit (CPU) with main memory.

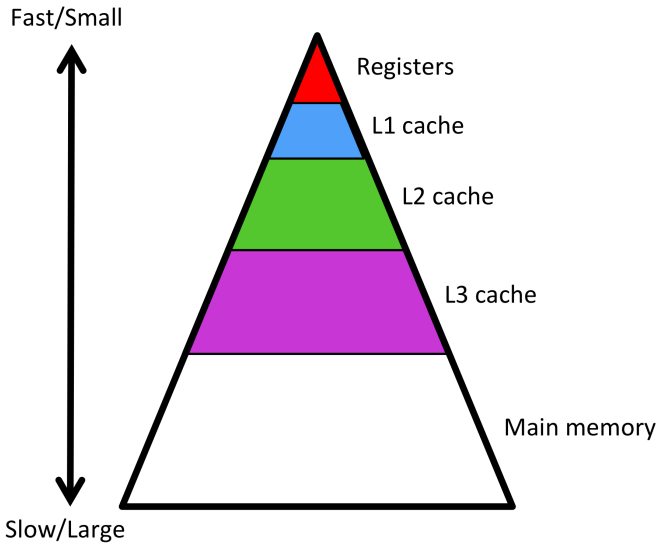
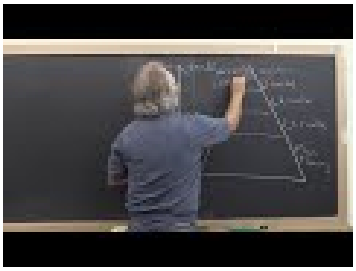


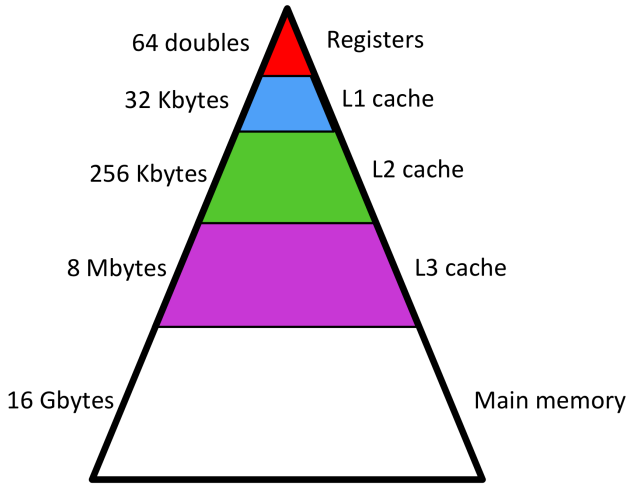
Figure 3.1.2: Illustration of the memory hierarchy.

To overcome this limitation, a modern processor has a hierarchy of memories. We have already encountered the two extremes: registers and main memory. In between, there are smaller but faster cache memories. These cache memories are on-chip and hence do not carry the same latency as does main memory and also can achieve greater bandwidth. The hierarchical nature of these memories is often depicted as a pyramid as illustrated in Figure 3.1.2.



YouTube: `https://www.youtube.com/watch?v=8u2wzTPZLtI`

To put things in perspective: We have discussed that a core on the kind of modern CPU we target in this course has sixteen vector registers that can store 4 double precision floating point numbers (doubles) each, for a total of 64 doubles. Built into the CPU it has a 32Kbytes level-1 cache (L1 cache) that can thus store 4,096 doubles. Somewhat further it has a 256Kbytes level-2 cache (L2 cache) that can store 32,768 doubles. Further away yet, but still on chip, it has an 8Mbytes level-3 cache (L3 cache) that can hold 1,048,576 doubles.



Remark 3.1.3 In further discussion, we will pretend that one can place data in a specific cache and keep it there for the duration of computations. In fact, caches retain data using some cache replacement policy that evicts data that has not been recently used. By carefully ordering computations, we can encourage data to remain in cache, which is what happens in practice.

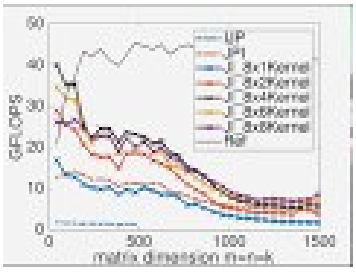
You may want to read up on cache replacement policies on Wikipedia: https://en.wikipedia.org/wiki/Cache_replacement_policies.

Homework 3.1.1.1 Since we compute with (sub)matrices, it is useful to have some idea of how big of a matrix one can fit in each of the layers of cache. Assuming each element in the matrix is a double precision number, which requires 8 bytes, complete the following table:

Layer	Size	Largest $n \times n$ matrix
Registers	16×4 doubles	$n =$
L1 cache	32 Kbytes	$n =$
L2 cache	256 Kbytes	$n =$
L3 cache	8 Mbytes	$n =$
Main Memory	16 Gbytes	$n =$

Solution.

Layer	Size	Largest $n \times n$ matrix
Registers	16×4 doubles	$n = \sqrt{64} = 8$
L1 cache	32 Kbytes	$n = \sqrt{32 \times 1024/8} = 64$
L2 cache	256 Kbytes	$n = \sqrt{256 \times 1024/8} = 181$
L3 cache	8 Mbytes	$n = \sqrt{8 \times 1024 \times 1024/8} = 1024$
Main memory	16 Gbytes	$n = \sqrt{16 \times 1024 \times 1024 \times 1024/8} \approx 46,341$



YouTube: https://www.youtube.com/watch?v=-nPXW8NiPmM

3.1.2 Outline Week 3

- 3.1 Opening Remarks
  - 3.1.1 Launch
  - 3.1.2 Outline Week 3
  - 3.1.3 What you will learn
- 3.2 Leveraging the Caches
  - 3.2.1 Adding cache memory into the mix
  - 3.2.2 Streaming submatrices of C and B
  - 3.2.3 Which cache to target?
  - 3.2.4 Blocking for the L1 and L2 caches
  - 3.2.5 Blocking for the L1, L2, and L3 caches
  - 3.2.6 Translating into code
- 3.3 Packing
  - 3.3.1 Stride matters
  - 3.3.2 Packing blocks of A and panels of B
  - 3.3.3 Implementation: packing row panel $B_{p,j}$
  - 3.3.4 Implementation: packing block $A_{i,p}$
  - 3.3.5 Implementation: five loops around the micro-kernel, with packing
  - 3.3.6 Micro-kernel with packed data
- 3.4 Further Tricks of the Trade
  - 3.4.1 Alignment
  - 3.4.2 Avoiding repeated memory allocations
  - 3.4.3 Play with the block sizes
  - 3.4.4 Broadcasting elements of A and loading elements of B
  - 3.4.5 Loop unrolling
  - 3.4.6 Prefetching
  - 3.4.7 Using in-lined assembly code

3.1.3 What you will learn

In this week, we discover the importance of amortizing the cost of moving data between memory layers.

Upon completion of this week, we will be able to

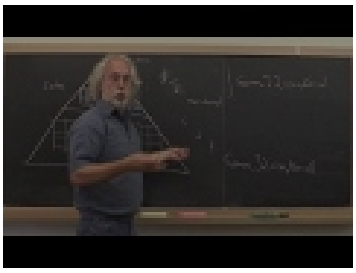
- Identify layers in the memory hierarchy.
- Orchestrate matrix-matrix multiplication in terms of computation with submatrices (blocks) so as to improve the ratio of computation to memory operations.
- Rearrange data to improve how memory is contiguously accessed.
- Organize loops so data is effectively reused at multiple levels of the memory hierarchy.
- Analyze the cost of moving data between memory layers.
- Improve performance by experimenting with different blocking parameters.

The enrichments introduce us to

- The origins of the algorithm that you developed and how it is incorporated into a widely used open source software library, BLIS.
- Analytical methods for determining optimal blocking sizes.
- Alternative algorithms that may become important when the ratio between the rate at which a processor computes and the bandwidth to memory further deteriorates.
- Practical techniques, based on the approach exposed in this course, for implementing Strassen's algorithm.

3.2 Leveraging the Caches

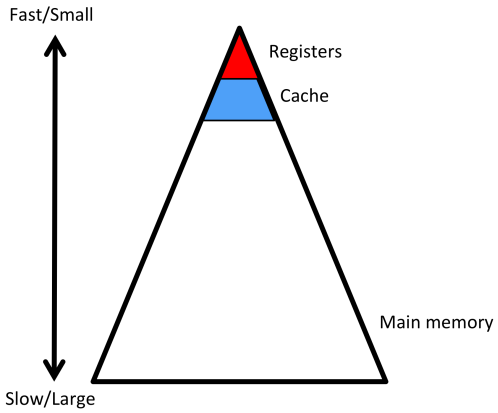
3.2.1 Adding cache memory into the mix



YouTube: https://www.youtube.com/watch?v=Kaq2WgLChv4

We now refine our model of the processor slightly, adding one layer of cache memory into the mix.

- Our processor has only one core.
- That core has three levels of memory: registers, a cache memory, and main memory.



- Moving data between the cache and registers takes time $\beta_{C\leftrightarrow R}$ per double while moving it between main memory and the cache takes time $\beta_{M\leftrightarrow C}$
- The registers can hold 64 doubles.
- The cache memory can hold one or more smallish matrices.
- Performing a floating point operation (multiply or add) with data in cache takes time γ_C.
- Data movement and computation cannot overlap. (In practice, it can. For now, we keep things simple.)

The idea now is to figure out how to block the matrices into submatrices and then compute while these submatrices are in cache to avoid having to access

memory more than necessary.

A naive approach partitions C, A, and B into (roughly) square blocks:

$$C = \left(\begin{array}{c|c|c|c} C_{0,0} & C_{0,1} & \cdots & C_{0,N-1} \\ \hline C_{1,0} & C_{1,1} & \cdots & C_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline C_{M-1,0} & C_{M,1} & \cdots & C_{M-1,N-1} \end{array}\right),$$

$$A = \left(\begin{array}{c|c|c|c} A_{0,0} & A_{0,1} & \cdots & A_{0,K-1} \\ \hline A_{1,0} & A_{1,1} & \cdots & A_{1,K-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline A_{M-1,0} & A_{M,1} & \cdots & A_{M-1,K-1} \end{array}\right),$$

and

$$B = \left(\begin{array}{c|c|c|c} B_{0,0} & B_{0,1} & \cdots & B_{0,N-1} \\ \hline B_{1,0} & B_{1,1} & \cdots & B_{1,N-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline B_{K-1,0} & B_{K,1} & \cdots & B_{K-1,N-1} \end{array}\right),$$

where $C_{i,j}$ is $m_C \times n_C$, $A_{i,p}$ is $m_C \times k_C$, and $B_{p,j}$ is $k_C \times n_C$. Then

$$C_{i,j} := \sum_{p=0}^{K-1} A_{i,p}B_{p,j} + C_{i,j},$$

which can be written as the triple-nested loop

```
for i := 0, ..., M − 1
    for j := 0, ..., N − 1
        for p := 0, ..., K − 1
            C_{i,j} := A_{i,p} B_{p,j} + C_{i,j}
        end
    end
end
```

This is one of $3! = 6$ possible loop orderings.

If we choose m_C, n_C, and k_C such that $C_{i,j}$, $A_{i,p}$, and $B_{p,j}$ all fit in the cache, then we meet our conditions. We can then compute $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$ by "bringing these blocks into cache" and computing with them before writing out the result, as before. The difference here is that while one can explicitly load registers, the movement of data into caches is merely encouraged by careful ordering of the computation, since replacement of data in cache is handled by the hardware, which has some cache replacement policy similar to "least recently used" data gets evicted.

Homework 3.2.1.1 For reasons that will become clearer later, we are going to assume that "the cache" in our discussion is the L2 cache, which for the CPUs we currently target is of size 256Kbytes. If we assume all three (square) matrices fit in that cache during the computation, what size should they be?

In our discussions 12 becomes a magic number because it is a multiple of

2, 3, 4, and 6, which allows us to flexibly play with a convenient range of block sizes. Therefore we should pick the size of the block to be the largest multiple of 12 less than that number. What is it?

Later we will further tune this parameter.

Solution.

- The number of doubles that can be stored in 256KBytes is

$$256 \times 1024/8 = 32 \times 1024$$

- Each of the three equally sized square matrices can contain $\frac{32}{3} \times 1024$ doubles.

- Each square matrix can hence be at most

$$\begin{array}{rcl} \sqrt{\frac{32}{3}} \times \sqrt{1024} \times \sqrt{\frac{32}{3}} \times \sqrt{1024} & = & \sqrt{\frac{32}{3}} \times 32 \times \sqrt{\frac{32}{3}} \times \sqrt{1024} \\ & \approx & 104 \times 104. \end{array}$$

- The largest multiple of 12 smaller than 104 is 96.

```
#define MC 96
#define NC 96
#define KC 96

void MyGemm( int m, int n, int k, double *A, int ldA,
             double *B, int ldB, double *C, int ldC )
{
  if ( m % MR != 0 || MC % MR != 0 ){
    printf( "m and MC must be multiples of MR\n" );
    exit( 0 );
  }
  if ( n % NR != 0 || NC % NR != 0 ){
    printf( "n and NC must be multiples of NR\n" );
    exit( 0 );
  }

  for ( int i=0; i<m; i+=MC ) {
    int ib = min( MC, m-i );        /* Last block may not be a full block */
    for ( int j=0; j<n; j+=NC ) {
      int jb = min( NC, n-j );        /* Last block may not be a full block */
      for ( int p=0; p<k; p+=KC ) {
        int pb = min( KC, k-p );        /* Last block may not be a full block */
        Gemm_JI_MRxNRKernel
          ( ib, jb, pb, &alpha( i,p ), ldA, &beta( p,j ), ldB, &gamma( i,j ), ldC );
      }
    }
  }
}

void Gemm_JI_MRxNRKernel( int m, int n, int k, double *A, int ldA,
                                  double *B, int ldB, double *C, int ldC )
{
  for ( int j=0; j<n; j+=NR ) /* n is assumed to be a multiple of NR */
    for ( int i=0; i<m; i+=MR ) /* m is assumed to be a multiple of MR */
      Gemm_MRxNRKernel( k, &alpha( i,0 ), ldA, &beta( 0,j ), ldB, &gamma( i,j ), ldC );
}
```

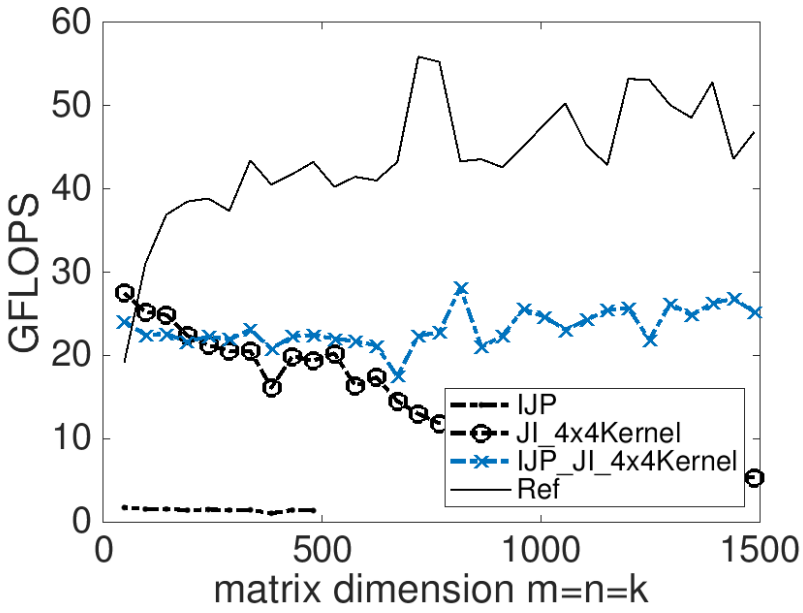
Figure 3.2.1: Triple loop around $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$.

Homework 3.2.1.2 In Figure 3.2.1, we give an IJP loop ordering around the computation of $C_{i,j} := A_{i,p}B_{p,j} + C_{i,j}$, which itself is implemented by Gemm_JI_4x4Kernel. It can be found in Assignments/Week3/C/Gemm_IJP_JI_MRxNRKernel.c. Copy Gemm_4x4Kernel.c from Week 2 into Assignments/Week3/C/ and then execute

```
make IJP_JI_4x4Kernel
```

in that directory. The resulting performance can be viewed with Live Script Assignments/Week3/C/data/Plot_XYZ_JI_MRxNRKernel.mlx.

Solution. On Robert's laptop:



Notice that now performance is (mostly) maintained as the problem size increases.

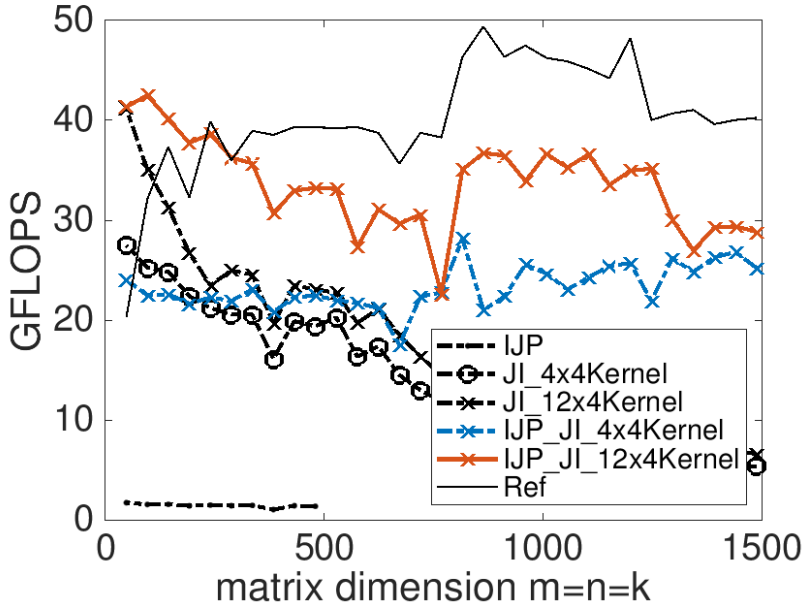
Homework 3.2.1.3 You can similarly link Gemm_IJP_JI_MRxNRKernel.c with your favorite micro-kernel by copying it from Week 2 into this directory and executing

```
make IJP_JI_??x??Kernel
```

where ??x?? reflects the $m_R \times n_R$ you choose. The resulting performance can be viewed with Live Script `Assignments/Week3/C/data/Plot_XYZ_JI_MRxNRKernel.mlx`, by making appropriate changes to that Live Script.

The Live Script assumes ??x?? to be 12x4. If you choose (for example) 8x6 instead, you may want to do a global "search and replace." You may then want to save the result into another file.

Solution. On Robert's laptop, for $m_R \times n_R = 12 \times 4$:



Remark 3.2.2 Throughout the remainder of this course, we often choose to proceed with $m_R \times n_R = 12 \times 4$, as reflected in the Makefile and the Live Script. If you pick m_R and n_R differently, then you will have to make the appropriate changes to the Makefile and the Live Script.

Homework 3.2.1.4 Create all six loop orderings by copying `Assignments/Week3/C/Gemm_IJP_JI_MRxNRKernel.c` into

```
Gemm_JIP_JI_MRxNRKernel.c
Gemm_IPJ_JI_MRxNRKernel.c
Gemm_PIJ_JI_MRxNRKernel.c
Gemm_PJI_JI_MRxNRKernel.c
Gemm_JPI_JI_MRxNRKernel.c
```

(or your choice of m_R and n_R), reordering the loops as indicated by the XYZ. Collect performance data by executing

```
make XYZ_JI_12x4Kernel
```

for `XYZ` $\in$ {`IPJ`, `JIP`, `JPI`, `PIJ`, `PJI`}.

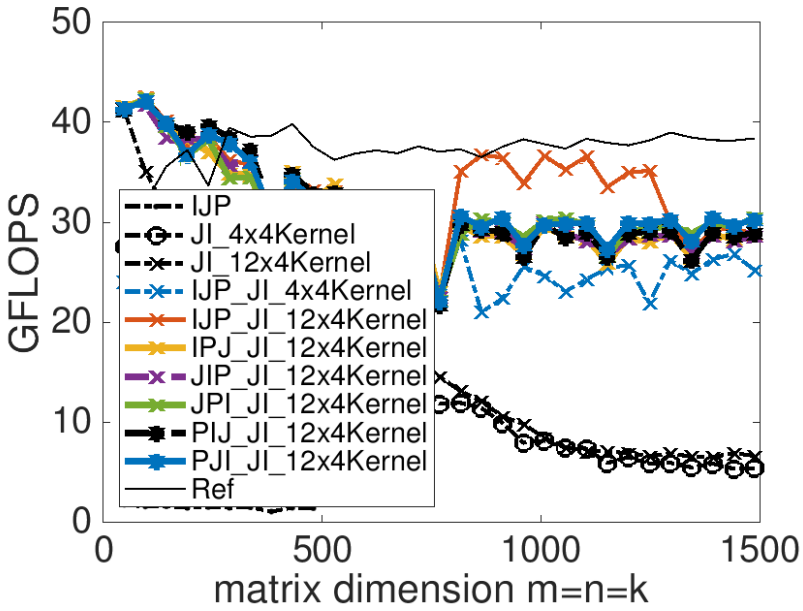
(If you don't want to do them all, implement at least Gemm_PIJ_JI_MRxNRKernel.c.)

The resulting performance can be viewed with Live Script `Assignments/Week3/C/data/Plot_XYZ_JI_MRxNRKernel.mlx`.

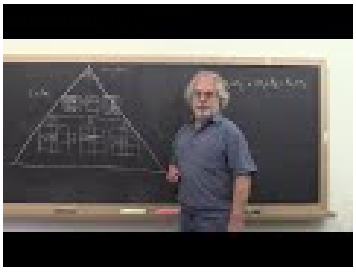
Solution.

- `Assignments/Week3/Answers/Gemm_IPJ_JI_MRxNRKernel.c`
- `Assignments/Week3/Answers/Gemm_JIP_JI_MRxNRKernel.c`
- `Assignments/Week3/Answers/Gemm_JPI_JI_MRxNRKernel.c`
- `Assignments/Week3/Answers/Gemm_PIJ_JI_MRxNRKernel.c`
- `Assignments/Week3/Answers/Gemm_PJI_JI_MRxNRKernel.c`

On Robert's laptop, for $m_R \times n_R = 12 \times 4$:



Things are looking up!



YouTube: https://www.youtube.com/watch?v=SwCYsVlo0lo

We can analyze the cost of moving submatrices $C_{i,j}$, $A_{i,p}$, and $B_{p,j}$ into the cache and computing with them:

$$\begin{array}{ccccc} \underbrace{m_C n_C \beta_{C\leftrightarrow M}}_{\text{load } C_{i,j}} & + & \underbrace{m_C k_C \beta_{C\leftrightarrow M}}_{\text{load } A_{i,p}} & + & \underbrace{k_C n_C \beta_{C\leftrightarrow M}}_{\text{load } B_{p,j}} \\ & + & \underbrace{2 m_C n_C k_C \gamma_C}_{\text{update } C_{i,j} += A_{i,p} B_{p,j}} & + & \underbrace{m_C n_C \beta_{C\leftrightarrow M}}_{\text{store } C_{i,j}} \end{array}$$

which equals

$$\underbrace{\left(2 m_C n_C + m_C k_C + k_C n_C\right) \beta_{C\leftrightarrow M}}_{\text{data movement overhead}} + \underbrace{2 m_C n_C k_C \gamma_C}_{\text{useful computation}}.$$

Hence, the ratio of time spent in useful computation and time spent in moving data between main memory a cache is given by

$$\frac{2 m_C n_C k_C \gamma_C}{\left(2 m_C n_C + m_C k_C + k_C n_C\right) \beta_C}.$$

Another way of viewing this is that the ratio between the number of floating point operations and number of doubles loaded/stored is given by

$$\frac{2m_C n_C k_C}{2m_C n_C + m_C k_C + k_C n_C}.$$

If, for simplicity, we take $m_C = n_C = k_C = b$ then we get that

$$\frac{2m_C n_C k_C}{2m_C n_C + m_C k_C + k_C n_C}. = \frac{2b^3}{4b^2} = \frac{b}{2}.$$

Thus, the larger b, the better the cost of moving data between main memory and cache is amortized over useful computation.

Above, we analyzed the cost of computing with and moving the data back and forth between the main memory and the cache for computation with three blocks (a $m_C \times n_C$ block of C, a $m_C \times k_C$ block of A, and a $k_C \times n_C$ block of B). If we analyze the cost of naively doing so with all the blocks, we get

$$\frac{m}{m_C}\frac{n}{n_C}\frac{k}{k_C}\left(\underbrace{\left(2m_C n_C + m_C k_C + k_C n_C\right)\beta_{C\leftrightarrow M}}_{\text{data movement overhead}} + \underbrace{2m_C n_C k_C \gamma_C}_{\text{useful computation}} \right).$$

What this means is that the effective ratio is the same if we consider the entire computation $C := AB + C$:

$$\frac{\frac{m}{m_C}\frac{n}{n_C}\frac{k}{k_C}\left(2m_C n_C + m_C k_C + k_C n_C\right)\beta_{C\leftrightarrow M}}{\frac{m}{m_C}\frac{n}{n_C}\frac{k}{k_C} 2m_C n_C k_C \gamma_C} = \frac{\left(2m_C n_C + m_C k_C + k_C n_C\right)\beta_{C\leftrightarrow M}}{2m_C n_C k_C \gamma_C}$$

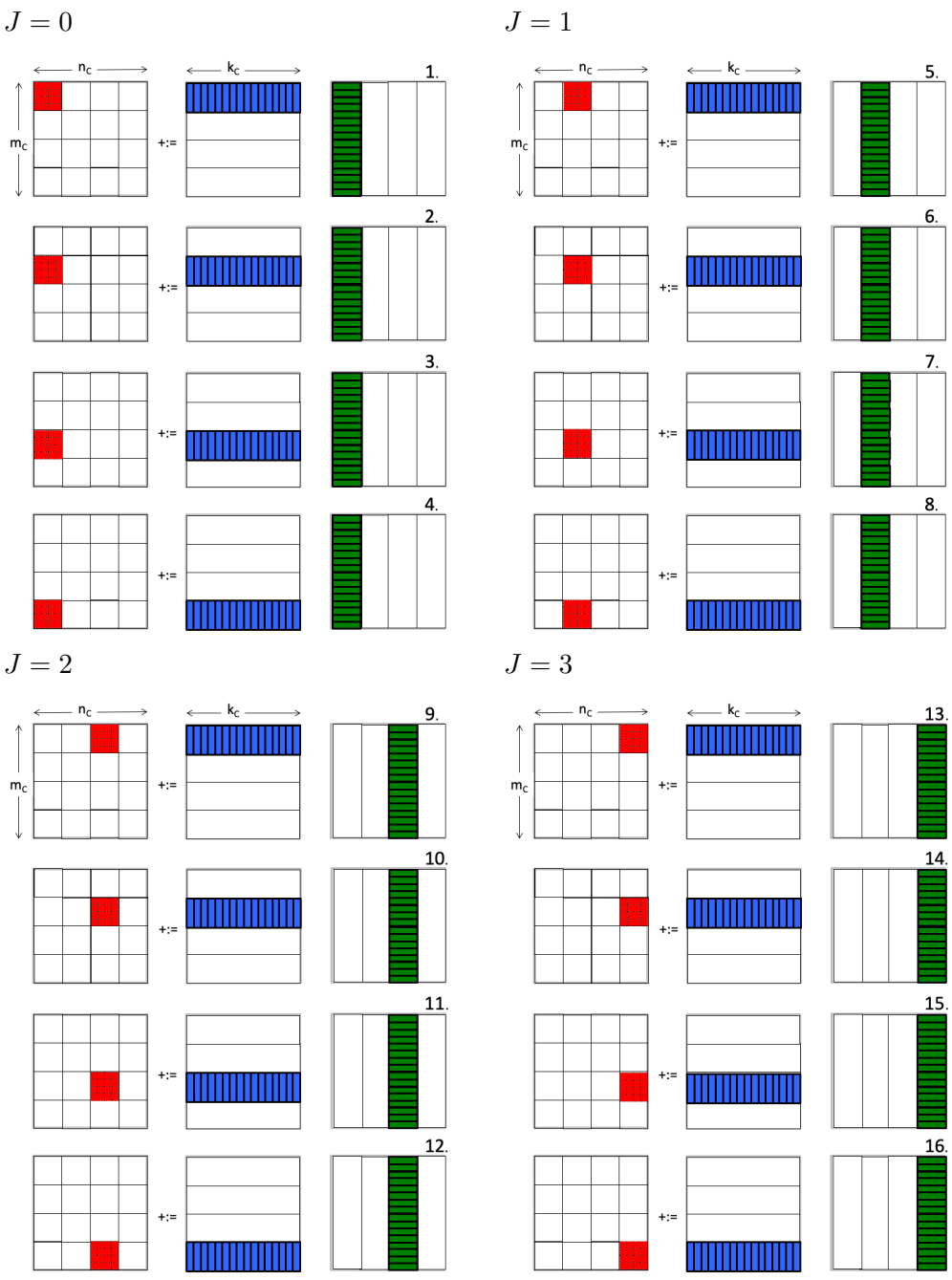
Remark 3.2.3

- Retrieving a double from main memory into cache can easily take 100 times more time than performing a floating point operation does.
- In practice the movement of the data can often be overlapped with computation (this is known as prefetching). However, clearly only so much can be overlapped, so it is still important to make the ratio favorable.

3.2.2 Streaming submatrices of C and B

YouTube: https://www.youtube.com/watch?v=W-oDAQYUExE

We illustrate the execution of Gemm_JI_4x4Kernel with one set of three submatrices $C_{i,j}$, $A_{i,p}$, and $B_{p,j}$ in Figure 3.2.4 for the case where $m_R = n_R = 4$ and $m_C = n_C = k_C = 4 \times m_R$.



Download PowerPoint source for illustration. PDF version.

Figure 3.2.4: Illustration of computation in the cache with one block from each of A, B, and C.

Notice that

- The $m_R \times n_R$ micro-tile of $C_{i,j}$ is not reused again once it has been updated. This means that at any given time only one such matrix needs to be in the cache memory (as well as registers).
- Similarly, a micro-panel of $B_{p,j}$ is not reused and hence only one such

panel needs to be in cache. It is submatrix $A_{i,p}$ that must remain in cache during the entire computation.

By not keeping the entire submatrices $C_{i,j}$ and $B_{p,j}$ in the cache memory, the size of the submatrix $A_{i,p}$ can be increased, which can be expected to improve performance. We will refer to this as "streaming" micro-tiles of $C_{i,j}$ and micro-panels of $B_{p,j}$, while keeping all of block $A_{i,p}$ in cache.

Remark 3.2.5 We could have chosen a different order of the loops, and then we may have concluded that submatrix $B_{p,j}$ needs to stay in cache while streaming a micro-tile of C and small panels of A. Obviously, there is a symmetry here and hence we will just focus on the case where $A_{i,p}$ is kept in cache.

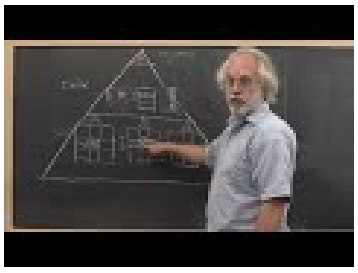
Homework 3.2.2.1 Reorder the updates in Figure 3.2.4 so that B stays in cache while panels of A are streamed.

You may want to use the PowerPoint source for that figure: Download PowerPoint source.

Answer. Answer (referring to the numbers in the top-right corner of each iteration):

$$\begin{array}{ccccccccccccccc} 1 & \rightarrow & 5 & \rightarrow & 9 & \rightarrow & 13 & \rightarrow & 2 & \rightarrow & 6 & \rightarrow & 10 & \rightarrow & 14 & \rightarrow \\ 3 & \rightarrow & 7 & \rightarrow & 11 & \rightarrow & 15 & \rightarrow & 4 & \rightarrow & 8 & \rightarrow & 12 & \rightarrow & 16 & \end{array}$$

Solution. Here is the PowerPoint source for one solution: Download PowerPoint source.



YouTube: https://www.youtube.com/watch?v=CAqIC2ixDrM

The second observation is that now that $C_{i,j}$ and $B_{p,j}$ are being streamed, there is no need for those submatrices to be square. Furthermore, for

```
Gemm_PIJ_JI_12x4Kernel.c
```

and

```
Gemm_IPJ_JI_12x4Kernel.c,
```

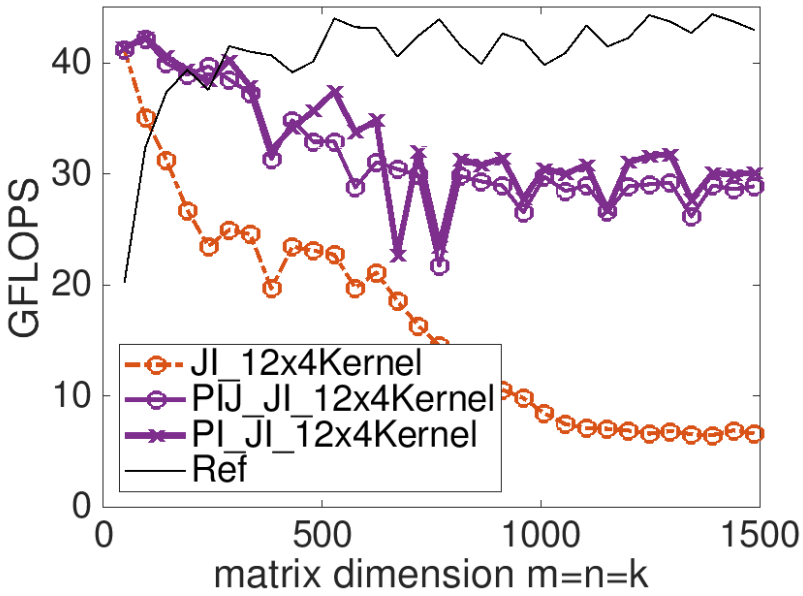
the larger n_C, the more effectively the cost of bringing $A_{i,p}$ into cache is amortized over computation: we should pick $n_C = n$. (Later, we will reintroduce n_C.) Another way of thinking about this is that if we choose the PIJ loop around the JI ordering around the micro-kernel (in other words, if we consider the Gemm_PIJ_JI_12x4Kernel implementation), then we notice that the inner loop of PIJ (indexed with J) matches the outer loop of the JI double loop, and hence those two loops can be combined, leaving us with an implementation that is a double loop (PI) around the double loop JI.

Homework 3.2.2.2 Copy `Assignments/Week3/C/Gemm_PIJ_JI_MRxNRKernel.c` into Gemm_PI_JI_MRxNRKernel.c and remove the loop indexed with j from MyGemm, making the appropriate changes to the call to the routine that implements the micro-kernel. Collect performance data by executing
`make PI_JI_12x4Kernel`

in that directory. The resulting performance can be viewed with Live Script `Assignments/Week3/C/data/Plot_XY_JI_MRxNRKernel.mlx`.

Solution. `Assignments/Week3/Answers/Gemm_PI_JI_MRxNRKernel.c`.

On Robert's laptop, for $m_R \times n_R = 12 \times 4$:



By removing that loop, we are simplifying the code without it taking a performance hit.

By combining the two loops we reduce the number of times that the blocks $A_{i,p}$ need to be read from memory:

$$\underbrace{m_C n \beta_{C\leftrightarrow M}}_{\text{load } C_{i,j}} + \underbrace{m_C k_C \beta_{C\leftrightarrow M}}_{\text{load } A_{i,p}} + \underbrace{k_C n \beta_{C\leftrightarrow M}}_{\text{load } B_{p,j}} + \underbrace{2 m_C n k_C \gamma_C}_{\text{update } C_i += A_{i,p} B_p} + \underbrace{m_C n \beta_{C\leftrightarrow M}}_{\text{store } C_{i,j}}$$

which equals

$$\underbrace{m_C k_C + (2 m_C n + k_C n)\, \beta_{C\leftrightarrow M}}_{\text{data movement overhead}} + \underbrace{2 m_C n k_C \gamma_C}_{\text{useful computation}}.$$

Hence, the ratio of time spent in useful computation, and time spent in moving data between main memory and cache is given by

$$\frac{2 m_C n k_C \gamma_C}{(2 m_C n + m_C k_C + k_C n)\, \beta_C}.$$

Another way of viewing this is that the ratio between the number of floating point operations, and number of doubles loaded and stored is given by

$$\frac{2m_C n k_C}{2m_C n + m_C k_C + k_C n}.$$

If, for simplicity, we take $m_C = k_C = b$ then we get that

$$\frac{2m_C n k_C}{2m_C n + m_C k_C + k_C n}. = \frac{2b^2 n}{3bn + b^2} \approx \frac{2b^2 n}{3bn} \approx \frac{2b}{3}$$

if b is much smaller than n. This is a slight improvement over the analysis from the last unit.

Remark 3.2.6 In our future discussions, we will use the following terminology:

- A $m_R \times n_R$ submatrix of C that is being updated we will call a micro-tile.
- The $m_R \times k_C$ submatrix of A and $k_C \times n_R$ submatrix of B we will call micro-panels.
- The routine that updates a micro-tile by multiplying two micro-panels we will call the micro-kernel.

Remark 3.2.7 The various block sizes we will further explore are

- $m_R \times n_R$: the sizes of the micro-tile of C that is kept in the registers.
- $m_C \times k_C$: the sizes of the block of A that is kept in the L2 cache.

3.2.3 Which cache to target?

YouTube: https://www.youtube.com/watch?v=lm8hG3LpQmo

Homework 3.2.3.1 In Assignments/Week3/C, execute
`make PI_JI_12x4_MCxKC`

and view the performance results with Live Script Assignments/Week3/C/data/Plot_MC_KC_Performance.mlx.

This experiment tries many different choices for m_C and k_C, and presents them as a scatter graph so that the optimal choice can be determined and visualized.

It will take a long time to execute. Go take a nap, catch a movie, and have some dinner! If you have a relatively old processor, you may even want to run it over night. Try not to do anything else on your computer, since that may affect the performance of an experiment. We went as far as to place something under the laptop to lift it off the table, to improve air circulation. You will

notice that the computer will start making a lot of noise, as fans kick in to try to cool the core.

Solution. The last homework on Robert's laptop yields the performance in Figure 3.2.8.

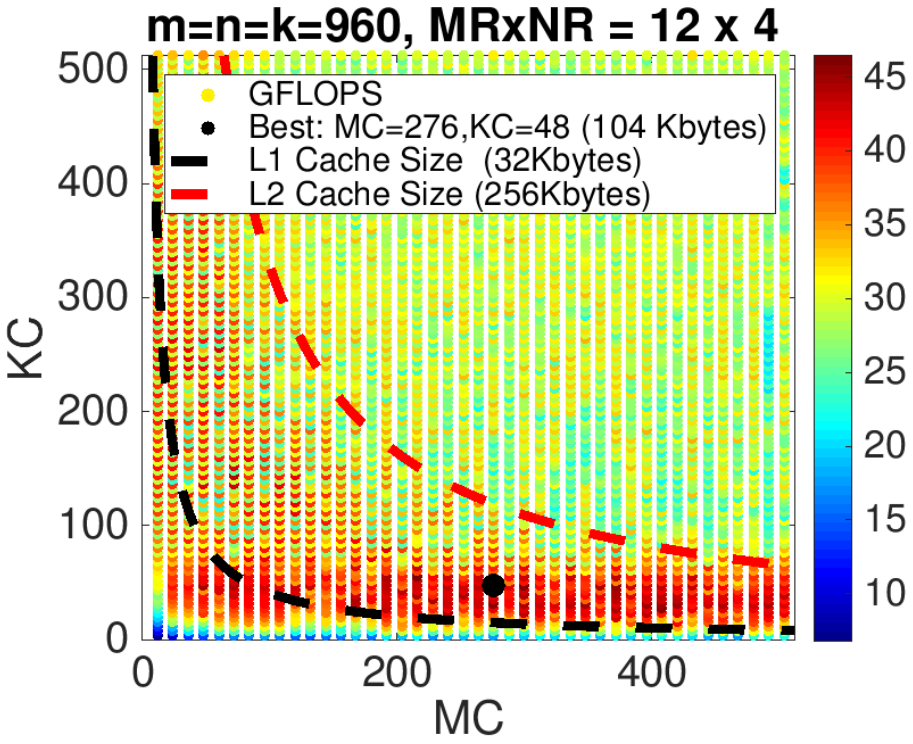
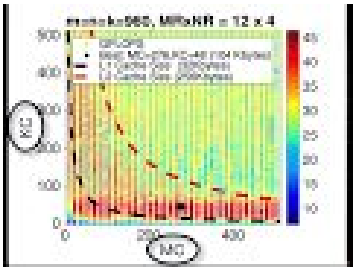


Figure 3.2.8: Performance when $m_R = 12$ and $n_R = 4$ and m_C and k_C are varied. The best performance is empirically determined by searching the results. We notice that the submatrix $A_{i,p}$ is sized to fit in the L2 cache. As we apply additional optimizations, the optimal choice for m_C and k_C will change. Notice that on Robert's laptop, the optimal choice varies greatly from one experiment to the next. You may observe the same level of variability on your computer.



YouTube: https://www.youtube.com/watch?v=aw7OqLVGSUA

To summarize, our insights suggest

1. Bring an $m_C \times k_C$ submatrix of A into the cache, at a cost of $m_C \times k_C \beta_{M \leftrightarrow C}$, or $m_C \times k_C$ memory operations (memops). (It is the order of the loops and instructions that encourages the submatrix of A to

stay in cache.) This cost is amortized over $2m_C n k_C$ flops for a ratio of $2m_C n k_C/(m_C k_C) = 2n$. The larger n, the better.

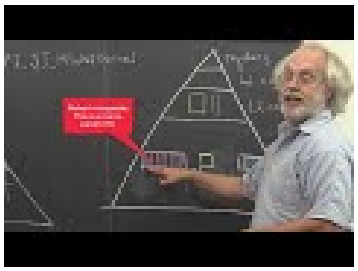
2. The cost of reading an $k_C \times n_R$ submatrix of B is amortized over $2m_C n_R k_C$ flops, for a ratio of $2m_C n_R k_C/(k_C n_R) = 2m_C$. Obviously, the greater m_C, the better.

3. The cost of reading and writing an $m_R \times n_R$ submatrix of C is now amortized over $2m_R n_R k_C$ flops, for a ratio of $2m_R n_R k_C/(2m_R n_R) = k_C$. Obviously, the greater k_C, the better.

Item 2 and Item 3 suggest that $m_C \times k_C$ submatrix $A_{i,p}$ be squarish.

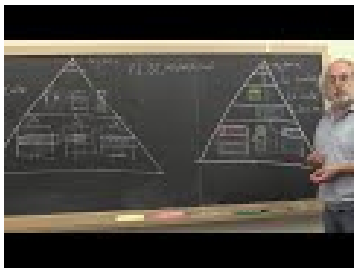
If we revisit the performance data plotted in Figure 3.2.8, we notice that matrix $A_{i,p}$ fits in part of the L2 cache, but is too big for the L1 cache. What this means is that the cost of bringing elements of A into registers from the L2 cache is low enough that computation can offset that cost. Since the L2 cache is larger than the L1 cache, this benefits performance, since the m_C and k_C can be larger.

3.2.4 Blocking for the L1 and L2 caches



YouTube: https://www.youtube.com/watch?v=pBtwu3E9lCQ

3.2.5 Blocking for the L1, L2, and L3 caches



YouTube: https://www.youtube.com/watch?v=U-tWXMAkmIs

The blocking for the various memory layers is captured in the following figure:

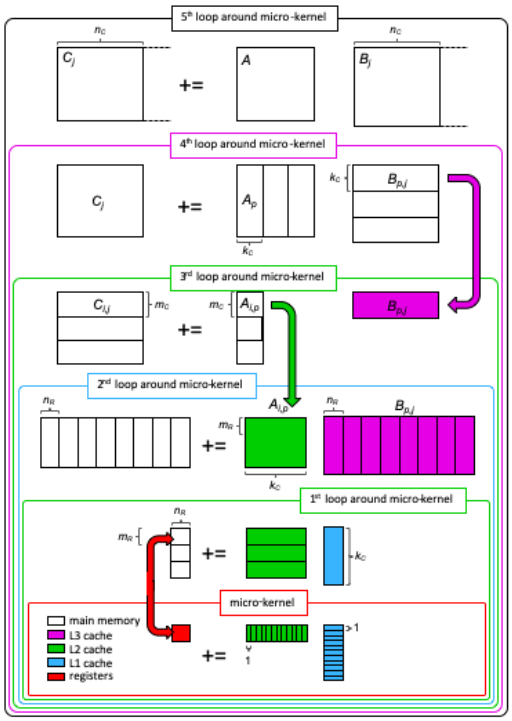
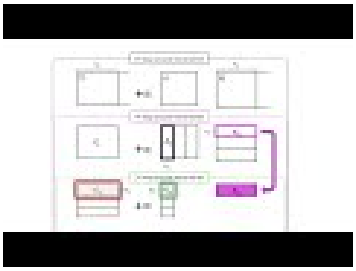


Figure 3.2.9: Illustration of the five loops around the micro-kernel. PowerPoint source for figure.



YouTube: https://www.youtube.com/watch?v=wfaXtM376iQ
PowerPoint source used in video.

Homework 3.2.5.1 Using our prior naming convention, which of the implementations

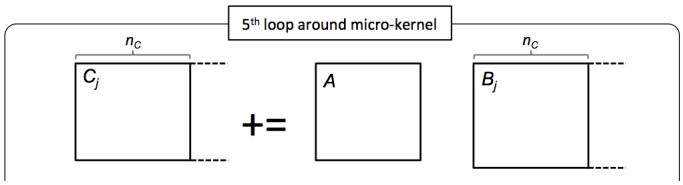
- Gemm_IJP_JI_MRxNRKernel.c
- Gemm_JPI_JI_MRxNRKernel.c
- Gemm_PJI_JI_MRxNRKernel.c

best captures the loop structure illustrated in Figure 3.2.9?

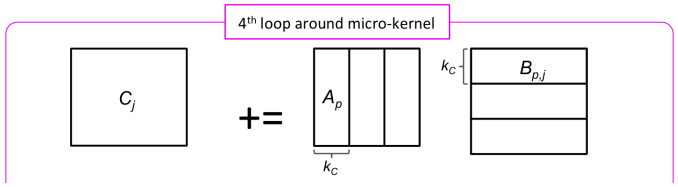
Answer. Gemm_JPI_JI_MRxNRKernel.c

3.2.6 Translating into code

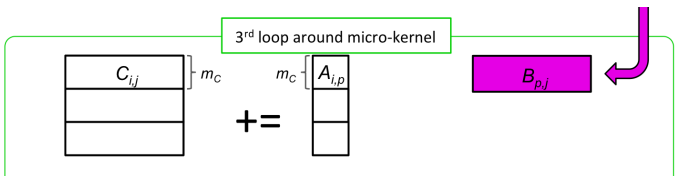
We always advocate, when it does not substantially impede performance, to instantiate an algorithm in code in a way that closely resembles how one explains it. In this spirit, the algorithm described in Figure 3.2.9 can be coded by making each loop (box) in the figure a separate routine. An outline of how this might be accomplished is illustrated next as well as in file `Assignments/Week3/C/Gemm_Five_Loops_MRxNRKernel.c`. You are asked to complete this code in the next homework.



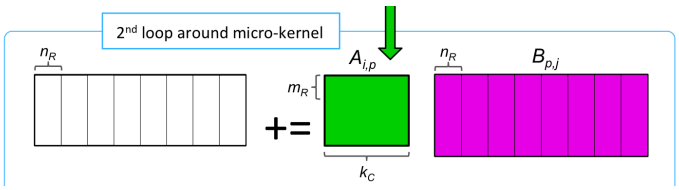

```
void LoopFive( int m, int n, int k, double *A, int ldA,
                   double *B, int ldB, double *C, int ldC )
{
  for ( int j=0; j<n; j+=NC ) {
    int jb = min( NC, n-j );    /* Last loop may not involve a full block */

    LoopFour( m, jb, k, A, ldA, &beta(  ,   ), ldB, &gamma(   ,   ), ldC );
  }
}
```



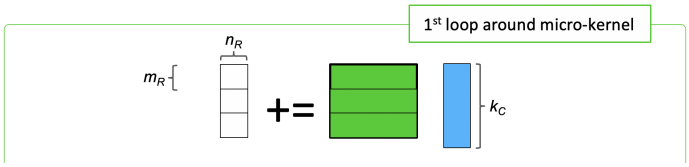

```
void LoopFour( int m, int n, int k, double *A, int ldA,
               double *B, int ldB, double *C, int ldC )
{
  for ( int p=0; p<k; p+=KC ) {
    int pb = min( KC, k-p );    /* Last loop may not involve a full block */

    LoopThree( m, n, pb, &alpha(  ,   ), ldA, &beta(  ,   ), ldB, C, ldC );
  }
}
```



```
void LoopThree( int m, int n, int k, double *A, int ldA,
                double *B, int ldB, double *C, int ldC )
{
  for ( int i=0; i<m; i+=MC ) {
    int ib = min( MC, m-i );    /* Last loop may not involve a full block */

    LoopTwo( ib, n, k, &alpha(  ,   ), ldA, B, ldB, &gamma(   ,   ), ldC );
  }
}
```




```
void LoopTwo( int m, int n, int k, double *A, int ldA,
              double *B, int ldB, double *C, int ldC )
{
  for ( int j=0; j<n; j+=NR ) {
    int jb = min( NR, n-j );

    LoopOne( m, jb, k, A, ldA, &beta(   ,   ), ldB, &gamma(   ,   ), ldC );
  }
}
```




```
void LoopOne( int m, int n, int k, double *A, int ldA,
              double *B, int ldB, double *C, int ldC )
{
  for ( int i=0; i<m; i+=MR ) {
    int ib = min( MR, m-i );
    Gemm_MRxNRKernel( k, &alpha(   ,    ), ldA, B, ldB, &gamma(   ,   ), ldC );
  }
}
```

Homework 3.2.6.1 Complete the code in Assignments/Week3/C/Gemm_Five_Loops_MRxNRK and execute it with

```
make Five_Loops_4x4Kernel
make Five_Loops_12x4Kernel
```

View the performance with Live Script Assignments/Week3/C/data/Plot_Five_

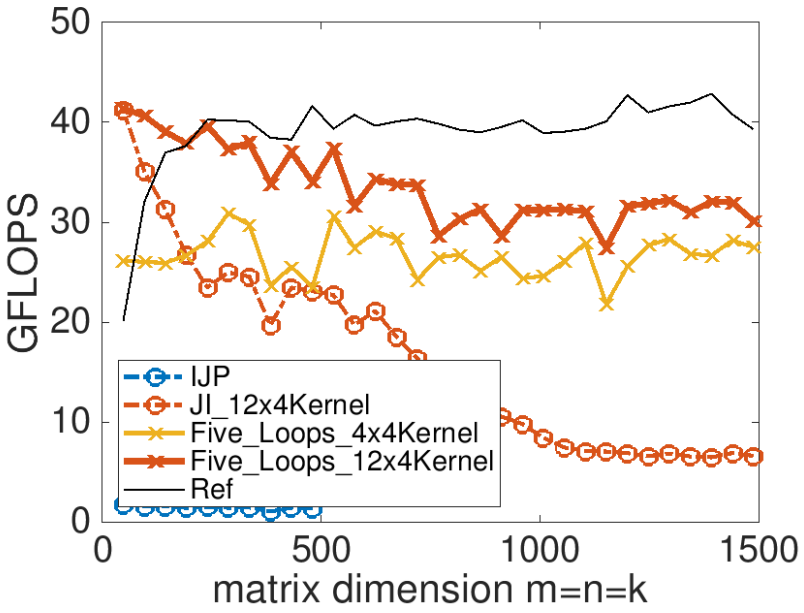
Loops.mlx.

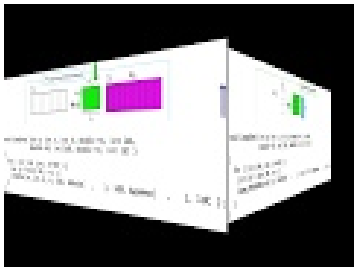
Notice that these link the routines Gemm_4x4Kernel.c and Gemm_12x4Kernel.c that you completed in Homework 2.4.3.1 and Homework 2.4.4.4.

You may want to play with the different cache block sizes (MB, NB, and KB) in the Makefile. In particular, you may draw inspiration from Figure 3.2.8. Pick them to be multiples of 12 to keep things simple. However, we are still converging to a high-performance implementation, and it is still a bit premature to try to pick the optimal parameters (which will change as we optimize further).

Solution. Assignments/Week3/Answers/Gemm_Five_Loops_MRxNRKernel.c.

On Robert's laptop:

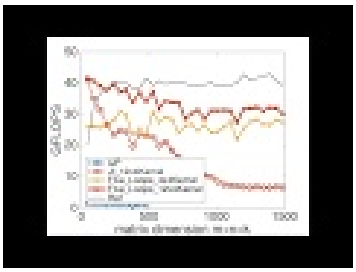




YouTube: https://www.youtube.com/watch?v=HUbbR2HTrHY

3.3 Packing

3.3.1 Stride matters



YouTube: https://www.youtube.com/watch?v=cEDMRH928CA

Homework 3.3.1.1 One would think that once one introduces blocking, the higher performance attained for small matrix sizes will be maintained as the problem size increases. This is not what you observed: there is still a decrease in performance. To investigate what is going on, copy the driver routine in Assignments/Week3/C/driver.c into driver_ldim.c and in that new file change the line

```
ldA = ldB = ldC = size;
```

to

```
ldA = ldB = ldC = last;
```

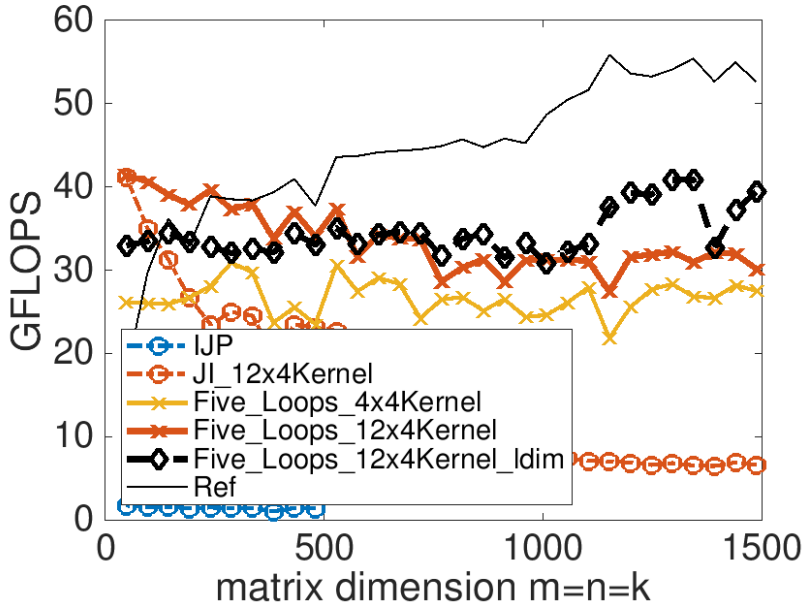
This sets the leading dimension of matrices A, B, and C to the largest problem size to be tested, for all experiments. Collect performance data by executing

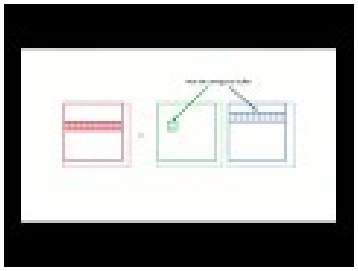
```
make Five_Loops_12x4Kernel_ldim
```

in that directory. The resulting performance can be viewed with Live Script Assignments/Week3/C/data/Plot_Five_Loops.mlx.

Solution. Assignments/Week3/Answers/driver_ldim.c.

On Robert's laptop:





YouTube: https://www.youtube.com/watch?v=zBKo7AvF7-c

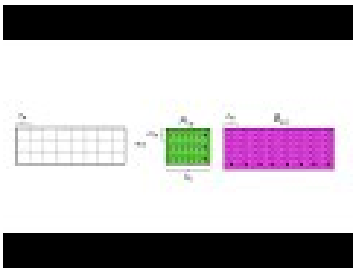
What is going on here? Modern processors incorporate a mechanism known as "hardware prefetching" that speculatively preloads data into caches based on observed access patterns so that this loading is hopefully overlapped with computation. Processors also organize memory in terms of pages, for reasons that have to do with virtual memory, details of which go beyond the scope of this course. For our purposes, a page is a contiguous block of memory of 4 Kbytes (4096 bytes or 512 doubles). Prefetching only occurs within a page. Since the leading dimension is now large, when the computation moves from column to column in a matrix data is not prefetched.

The bottom line: early in the course we discussed that marching contiguously through memory is a good thing. While we claim that we do block for the caches, in practice the data is typically not contiguously stored and hence we can't really control how the data is brought into caches, where it exists at a particular time in the computation, and whether prefetching is activated. Next, we fix this through packing of data at strategic places in the nested loops.

Learn more:

- Memory page: https://en.wikipedia.org/wiki/Page_(computer_memory).
- Cache prefetching: https://en.wikipedia.org/wiki/Cache_prefetching.

3.3.2 Packing blocks of A and panels of B



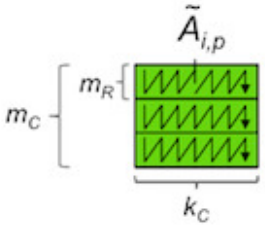
YouTube: https://www.youtube.com/watch?v=UdBix7fYVQg

The next step towards optimizing the micro-kernel exploits that computing with contiguous data (accessing data with a "stride one" access pattern) improves performance. The fact that the $m_R \times n_R$ micro-tile of C is not in contiguous memory is not particularly important. The cost of bringing it into the vector registers from some layer in the memory is mostly inconsequential because a lot of computation is performed before it is written back out. It is the repeated accessing of the elements of A and B that can benefit from stride one access.

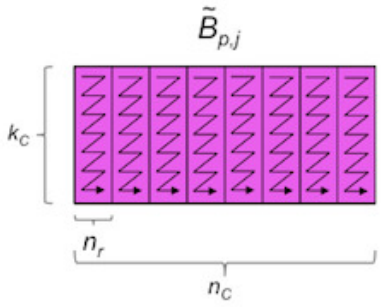
Two successive rank-1 updates of the micro-tile can be given by

$$
\begin{pmatrix} \gamma_{0,0} & \gamma_{0,1} & \gamma_{0,2} & \gamma_{0,3} \\ \gamma_{1,0} & \gamma_{1,1} & \gamma_{1,2} & \gamma_{0,3} \\ \gamma_{2,0} & \gamma_{2,1} & \gamma_{2,2} & \gamma_{0,3} \\ \gamma_{3,0} & \gamma_{3,1} & \gamma_{3,2} & \gamma_{0,3} \end{pmatrix} +:=
$$
$$
\begin{pmatrix} \alpha_{0,p} \\ \alpha_{1,p} \\ \alpha_{2,p} \\ \alpha_{3,p} \end{pmatrix} \begin{pmatrix} \beta_{p,0} & \beta_{p,1} & \beta_{p,2} & \beta_{p,3} \end{pmatrix}
$$
$$
+ \begin{pmatrix} \alpha_{0,p+1} \\ \alpha_{1,p+1} \\ \alpha_{2,p+1} \\ \alpha_{3,p+1} \end{pmatrix} \begin{pmatrix} \beta_{p+1,0} & \beta_{p+1,1} & \beta_{p+1,2} & \beta_{p+1,3} \end{pmatrix}.
$$

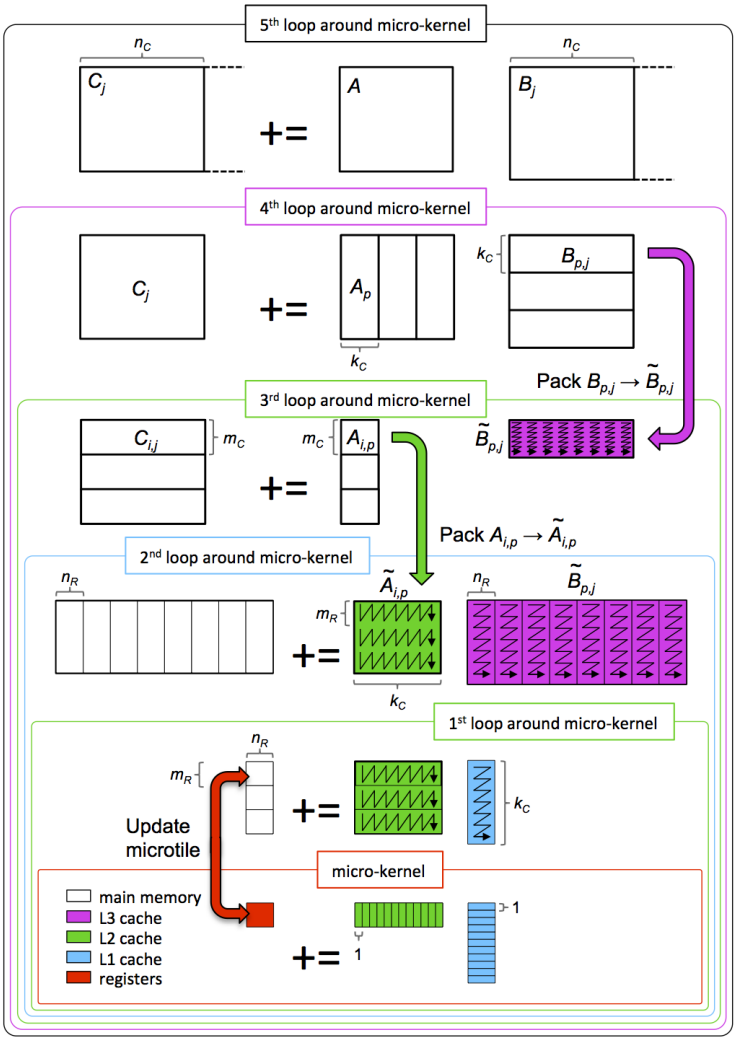
Since A and B are stored with column-major order, the four elements of $\alpha_{[0:3],p}$ are contiguous in memory, but they are (generally) not contiguously stored with $\alpha_{[0:3],p+1}$. Elements $\beta_{p,[0:3]}$ are (generally) also not contiguous. The access pattern during the computation by the micro-kernel would be much more favorable if the A involved in the micro-kernel was packed in column-major order with leading dimension m_R:



and B was packed in row-major order with leading dimension n_R:



If this packing were performed at a strategic point in the computation, so that the packing is amortized over many computations, then a benefit might result. These observations are captured in Figure 3.3.1.

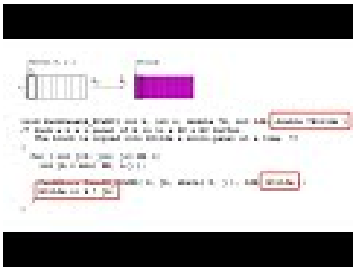


Picture adapted from [27]. Click to enlarge. PowerPoint source (step-by-step).

Figure 3.3.1: Blocking for multiple levels of cache, with packing.

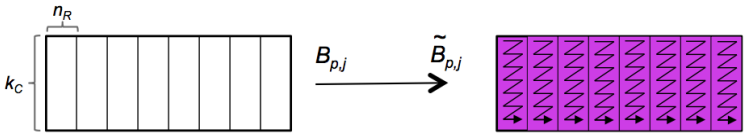
In the next units you will discover how to implement packing and how to then integrate it into the five loops around the micro-kernel.

3.3.3 Implementation: packing row panel $B_{p,j}$

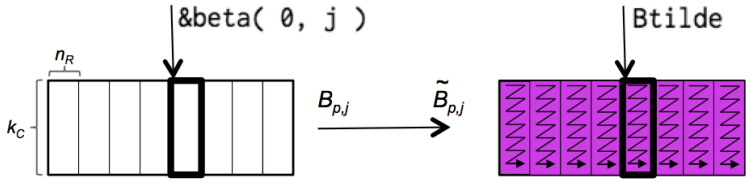


YouTube: https://www.youtube.com/watch?v=0ovhed0gzr4

We briefly discuss the packing of the row panel $B_{p,j}$ into $\widetilde{B}_{p,j}$:



We break the implementation, in Assignments/Week3/C/PackB.c, down into two routines. The first loops over all the panels that need to be packed

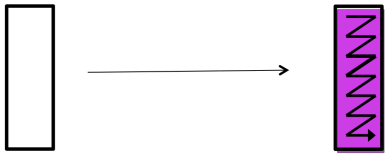


as illustrated in Figure 3.3.2.

```
void PackPanelB_KCxNC( int k, int n, double *B, int ldB, double *Btilde )
/* Pack a k x n panel of B in to a KC x NC buffer.
.
   The block is copied into Btilde a micro-panel at a time. */
{
  for ( int j=0; j<n; j+= NR ){
    int jb = min( NR, n-j );

    PackMicro-PanelB_KCxNR( k, jb, &beta( 0, j ), ldB, Btilde );
    Btilde += k * jb;
  }
}
```

Figure 3.3.2: A reference implementation for packing $B_{p,j}$.

That routine then calls a routine that packs the panel



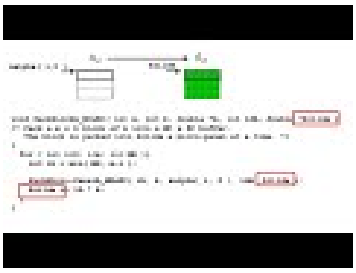
Given in Figure 3.3.3.

```
void PackMicroPanelB_KCxNR( int k, int n, double *B, int ldB,
            double *Btilde )
/* Pack a micro-panel of B into buffer pointed to by Btilde.
   This is an unoptimized implementation for general KC and NR.
   k is assumed to be less then or equal to KC.
   n is assumed to be less then or equal to NR.  */
{
  /* March through B in row-major order, packing into Btilde. */
  if ( n == NR ) {
    /* Full column width micro-panel.*/
    for ( int p=0; p<k; p++ )
      for ( int j=0; j<NR; j++ )
        *Btilde++ = beta( p, j );
  }
  else {
    /* Not a full row size micro-panel. We pad with zeroes.
     To be added */
  }
}
```

Figure 3.3.3: A reference implementation for packing a micro-panel of $B_{p,j}$.

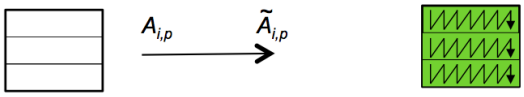
Remark 3.3.4 We emphasize that this is a "quick and dirty" implementation. It is meant to be used for matrices that are sized to be nice multiples of the various blocking parameters. The goal of this course is to study how to implement matrix-matrix multiplication for matrices that are nice multiples of these blocking sizes. Once one fully understands how to optimize that case, one can start from scratch and design an implementation that works for all matrix sizes. Alternatively, upon completing this week one understands the issues well enough to be able to study a high-quality, open source implementation like the one in our BLIS framework [3].

3.3.4 Implementation: packing block $A_{i,p}$

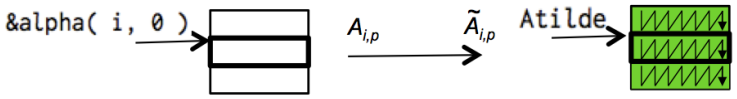


YouTube: https://www.youtube.com/watch?v=GDYA478W-f0

We next discuss the packing of the block $A_{i,p}$ into $\widetilde{A}_{i,p}$:



We break the implementation, in Assignments/Week3/C/PackA.c, down into two routines. The first loops over all the rows that need to be packed



as illustrated in Figure 3.3.5.

```
void PackBlockA_MCxKC( int m, int k, double *A, int ldA, double *Atilde )
/* Pack a  m x k block of A into a MC x KC buffer.   MC is assumed to
    be a multiple of MR.  The block is packed into Atilde a micro-panel
    at a time. If necessary, the last micro-panel is padded with rows
    of zeroes. */
{
  for ( int i=0; i<m; i+= MR ){
    int ib = min( MR, m-i );

    PackMicro-PanelA_MRxKC( ib, k, &alpha( i, 0 ), ldA, Atilde );
    Atilde += ib * k;
  }
}
```

Figure 3.3.5: A reference implementation for packing $A_{i,p}$.

That routine then calls a routine that packs the panel

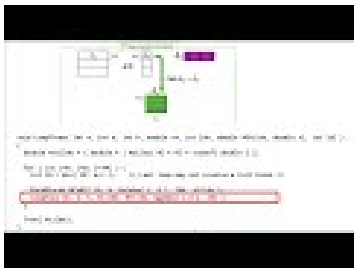
Given in Figure 3.3.6.

```
void PackMicroPanelA_MRxKC( int m, int k, double *A, int ldA, double *Atilde )
/* Pack a micro-panel of A into buffer pointed to by Atilde.
   This is an unoptimized implementation for general MR and KC. */
{
  /* March through A in column-major order, packing into Atilde as we go. */

  if ( m == MR ) {
    /* Full row size micro-panel.*/
    for ( int p=0; p<k; p++ )
      for ( int i=0; i<MR; i++ )
        *Atilde++ = alpha( i, p );
  }
  else {
    /* Not a full row size micro-panel.  We pad with zeroes.  To be  added */
  }
}
```

Figure 3.3.6: A reference implementation for packing a micro-panel of $A_{i,p}$.

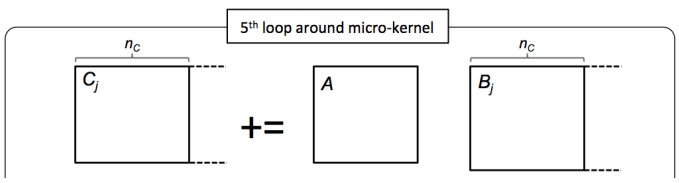
Remark 3.3.7 Again, these routines only work when the sizes are "nice". We leave it as a challenge to generalize all implementations so that matrix-matrix multiplication with arbitrary problem sizes works. To manage the complexity of this, we recommend "padding" the matrices with zeroes as they are being packed. This then keeps the micro-kernel simple.

3.3.5 Implementation: five loops around the micro-kernel, with packing



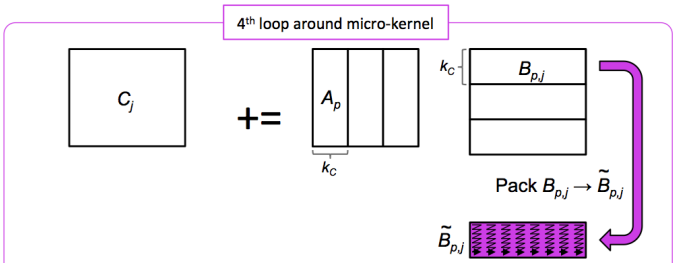
YouTube: https://www.youtube.com/watch?v=Fye-IwGDTYA

Below we illustrate how packing is incorporated into the "Five loops around the micro-kernel".

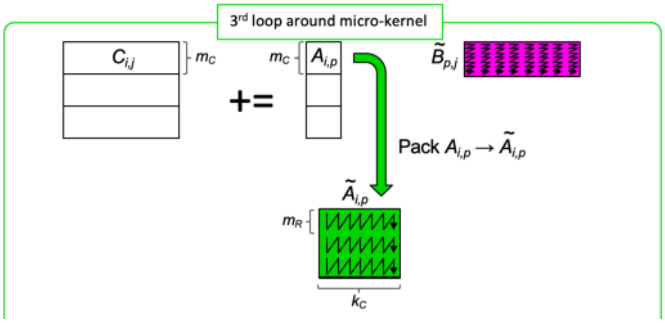



```
void LoopFive( int m, int n, int k, double *A, int ldA,
                   double *B, int ldB, double *C, int ldC )
{
```

```
  for ( int j=0; j<n; j+=NC ) {
    int jb = min( NC, n-j );    /* Last loop may not involve a full block */

    LoopFour( m, jb, k, A, ldA, &beta( 0,j ), ldB, &gamma( 0,j ), ldC );
  }
}
```

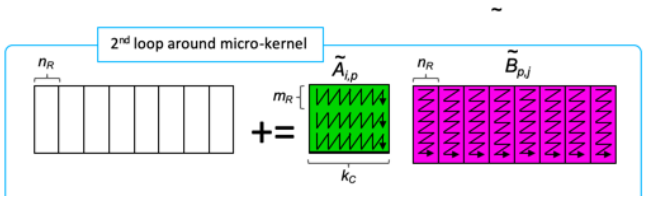



```
void LoopFour( int m, int n, int k, double *A, int ldA, double *B, int ldB,
               double *C, int ldC )
{
  double *Btilde = ( double * ) malloc( KC * NC * sizeof( double ) );

  for ( int p=0; p<k; p+=KC ) {
    int pb = min( KC, k-p );    /* Last loop may not involve a full block */

    PackPanelB_KCxNC( pb, n, &beta( p, 0 ), ldB, Btilde );
    LoopThree( m, n, pb, &alpha( 0, p ), ldA, Btilde, C, ldC );
  }
  free( Btilde);
}
```

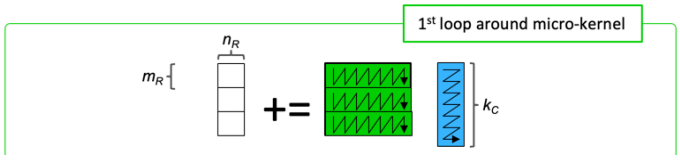



```
void LoopThree( int m, int n, int k, double *A, int ldA, double *Btilde, double *C, in
{
  double *Atilde = ( double * ) malloc( MC * KC * sizeof( double ) );

  for ( int i=0; i<m; i+=MC ) {
    int ib = min( MC, m-i );    /* Last loop may not involve a full block */
```

```
    PackBlockA_MCxKC( ib, k, &alpha( i, 0 ), ldA, Atilde );
    LoopTwo( ib, n, k, Atilde, Btilde, &gamma( i,0 ), ldC );
  }

  free( Atilde);
}
```




```
void LoopTwo( int m, int n, int k, double *Atilde, double *Btilde, double *C, int ldC )
{
  for ( int j=0; j<n; j+=NR ) {
    int jb = min( NR, n-j );

    LoopOne( m, jb, k, Atilde, &Btilde[ j*k ], &gamma( 0,j ), ldC );
  }
```




```
void LoopOne( int m, int n, int k, double *Atilde, double *Micro-PanelB, double *C, int ldC )
{
  for ( int i=0; i<m; i+=MR ) {
    int ib = min( MR, m-i );

    Gemm_MRxNRKernel_Packed( k,&Atilde[ i*k ], Micro-PanelB, &gamma( i,0 ), ldC );
  }
}
```

3.3.6 Micro-kernel with packed data

YouTube: https://www.youtube.com/watch?v=4hd0QnQ6JBo

```
void Gemm_MRxNRKernel_Packed( int k,
                         double *MP_A, double *MP_B, double *C, int ldC )
{
  __m256d gamma_0123_0 = _mm256_loadu_pd( &gamma( 0,0 ) );
  __m256d gamma_0123_1 = _mm256_loadu_pd( &gamma( 0,1 ) );
  __m256d gamma_0123_2 = _mm256_loadu_pd( &gamma( 0,2 ) );
  __m256d gamma_0123_3 = _mm256_loadu_pd( &gamma( 0,3 ) );

  __m256d beta_p_j;

  for ( int p=0; p<k; p++ ){
    /* load alpha( 0:3, p ) */
    __m256d alpha_0123_p = _mm256_loadu_pd( MP_A );

    /* load beta( p, 0 ); update gamma( 0:3, 0 ) */
    beta_p_j = _mm256_broadcast_sd( MP_B );
    gamma_0123_0 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_0 );

    /* load beta( p, 1 ); update gamma( 0:3, 1 ) */
    beta_p_j = _mm256_broadcast_sd( MP_B+1 );
    gamma_0123_1 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_1 );

    /* load beta( p, 2 ); update gamma( 0:3, 2 ) */
    beta_p_j = _mm256_broadcast_sd( MP_B+2 );
    gamma_0123_2 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_2 );

    /* load beta( p, 3 ); update gamma( 0:3, 3 ) */
    beta_p_j = _mm256_broadcast_sd( MP_B+3 );
    gamma_0123_3 = _mm256_fmadd_pd( alpha_0123_p, beta_p_j, gamma_0123_3 );

    MP_A += MR;
    MP_B += NR;
  }

  /* Store the updated results.  This should be done more carefully since
     there may be an incomplete micro-tile. */
  _mm256_storeu_pd( &gamma(0,0), gamma_0123_0 );
  _mm256_storeu_pd( &gamma(0,1), gamma_0123_1 );
  _mm256_storeu_pd( &gamma(0,2), gamma_0123_2 );
  _mm256_storeu_pd( &gamma(0,3), gamma_0123_3 );
}
```

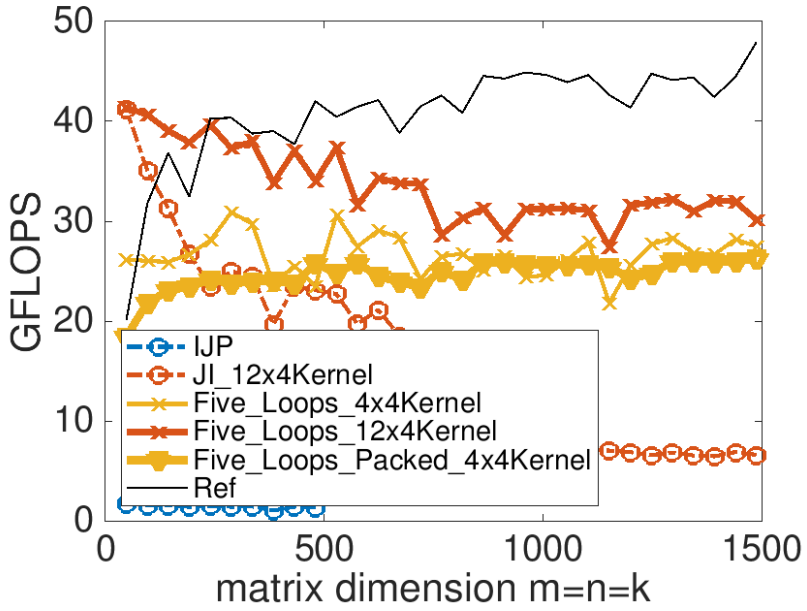
Figure 3.3.8: Blocking for multiple levels of cache, with packing.

How to modify the five loops to incorporate packing was discussed in Unit 3.3.5. A micro-kernel to compute with the packed data when $m_R \times n_R = 4 \times 4$ is now illustrated in Figure 3.3.8.

Homework 3.3.6.1 Examine the files Assignments/Week3/C/Gemm_Five_Loops_Packed_MRxNRKernel.c and Assignments/Week3/C/Gemm_4x4Kernel_Packed.c. Collect performance data with
`make Five_Loops_Packed_4x4Kernel`

and view the resulting performance with Live Script Plot_Five_Loops.mlx.

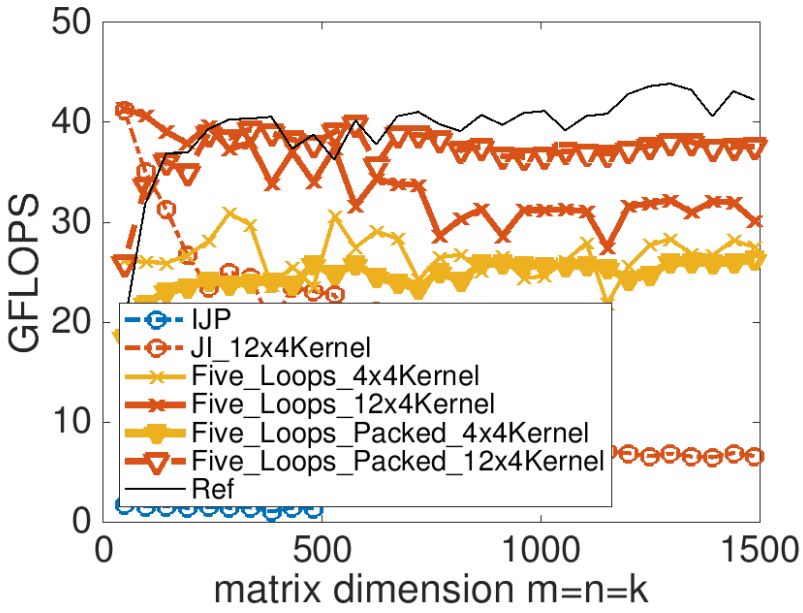
Solution. On Robert's laptop:



Homework 3.3.6.2 Copy the file Gemm_4x4Kernel_Packed.c into file Gemm_12x4Kernel_Packed.c. Modify that file so that it uses $m_R \times n_R = 12 \times 4$. Test the result with
`make Five_Loops_Packed_12x4Kernel`

and view the resulting performance with Live Script Plot_Five_Loops.mlx.

Solution. Assignments/Week3/Answers/Gemm_12x4Kernel_Packed.c
On Robert's laptop:



Now we are getting somewhere!

Homework 3.3.6.3 In Homework 3.2.3.1, you determined the best block sizes MC and KC. Now that you have added packing to the implementation of the five loops around the micro-kernel, these parameters need to be revisited. You can collect data for a range of choices by executing
`make Five_Loops_Packed_?x?Kernel_MCxKC`

where ?x? is your favorite choice for register blocking. View the result with data/Plot_Five_loops.mlx.

Solution.

- Assignments/Week4/Answers/Gemm_Five_Loops_Packed_MRxNRKernel.c

3.4 Further Tricks of the Trade

The course so far has exposed you to the "big ideas" on how to highly optimize matrix-matrix multiplication. You may be able to apply some of these ideas (how to extract parallelism via vector instructions, the importance of reusing data, and the benefits of accessing data with stride one) to your own computations. For some computations, there is no opportunity to reuse data to that same extent, which means they are inherently bandwidth bound.

By now, you may also have discovered that this kind of programming is just not your cup of soup. What we hope you will then have realized is that there are high-performance libraries out there, and that it is important to cast your computations in terms of calls to those libraries so that you benefit from the expertise of others.

Some of you now feel "the need for speed." You enjoy understanding how software and hardware interact and you have found a passion for optimizing

computation. In this section, we give a laundry list of further optimizations that can be tried by you and others like you. We merely mention the possibilities and leave it to you to research the details and translate these into even faster code. If you are taking this course as an edX MOOC, you can discuss the result with others on the discussion forum.

3.4.1 Alignment (easy and worthwhile)

The vector intrinsic routines that load and/or broadcast vector registers are faster when the data from which one loads is aligned.

Conveniently, loads of elements of A and B are from buffers into which the data was packed. By creating those buffers to be aligned, we can ensure that all these loads are aligned. Intel's intrinsic library has a special memory allocation and deallocation routines specifically for this purpose: _mm_malloc and _mm_free. (align should be chosen to equal the length of a cache line, in bytes: 64.)

3.4.2 Avoiding repeated memory allocations (may become important in Week 4)

In the final implementation that incorporates packing, at various levels workspace is allocated and deallocated multiple times. This can be avoided by allocating workspace immediately upon entering LoopFive (or in MyGemm). Alternatively, the space could be "statically allocated" as a global array of appropriate size. In "the real world" this is a bit dangerous: If an application calls multiple instances of matrix-matrix multiplication in parallel (e.g., using OpenMP about which we will learn next week), then these different instances may end up overwriting each other's data as they pack, leading to something called a "race condition." But for the sake of this course, it is a reasonable first step.

3.4.3 Play with the block sizes (highly recommended)

As mentioned, there are many choices for $m_R \times n_R$. In our discussions, we focused on 12×4. That different choices for $m_R \times n_R$ yield better performance, now that packing has been added into the implementation.

Once you have determined the best $m_R \times n_R$ you may want to go back and redo Homework 3.3.6.3 to determine the best m_C and k_C. Then, collect final performance data once you have updated the Makefile with the best choices.

3.4.4 Broadcasting elements of A and loading elements of B (tricky and maybe not that worthwhile)

So far, we have focused on choices for $m_R \times n_R$ where m_R is a multiple of four. Before packing was added, this was because one loads into registers from contiguous memory and for this reason loading from A, four elements at a time, while broadcasting elements from B was natural because of how data was mapped to memory using column-major order. After packing was introduced, one could also contemplate loading from B and broadcasting elements of A,

which then also makes kernels like 6×8 possible. The problem is that this would mean performing a vector instruction with elements that are in rows of C, and hence when loading elements of C one would need to perform a transpose operation, and also when storing the elements back to memory.

There are reasons why a 6×8 kernel edges out even a 8×6 kernel. On the surface, it may seem like this would require a careful reengineering of the five loops as well as the micro-kernel. However, there are tricks that can be played to make it much simpler than it seems.

Intuitively, if you store matrices by columns, leave the data as it is so stored, but then work with the data as if it is stored by rows, this is like computing with the transposes of the matrices. Now, mathematically, if

$$C := AB + C$$

then

$$C^T := (AB + C)^T = (AB)^T + C^T = B^T A^T + C^T.$$

If you think about it: transposing a matrix is equivalent to viewing the matrix, when stored in column-major order, as being stored in row-major order.

Consider the simplest implementation for computing $C := AB + C$ from Week 1:

```
#define alpha( i,j ) A[ (j)*ldA + i ]
#define beta( i,j )  B[ (j)*ldB + i ]
#define gamma( i,j ) C[ (j)*ldC + i ]

void MyGemm( int m, int n, int k, double *A, int ldA,
                     double *B, int ldB, double *C, int ldC )
{
  for ( int i=0; i<m; i++ )
    for ( int j=0; j<n; j++ )
      for ( int p=0; p<k; p++ )
        gamma( i,j ) += alpha( i,p ) * beta( p,j );
}
```

From this, a code that instead computes $C^T := B^T A^T + C^T$, with matrices still stored in column-major order, is given by

```
#define alpha( i,j ) A[ (j)*ldA + i ]
#define beta( i,j )  B[ (j)*ldB + i ]
#define gamma( i,j ) C[ (j)*ldC + i ]

void MyGemm( int m, int n, int k, double *A, int ldA,
                     double *B, int ldB, double *C, int ldC )
{
  for ( int i=0; i<m; i++ )
    for ( int j=0; j<n; j++ )
      for ( int p=0; p<k; p++ )
        gamma( j,i ) += alpha( p,i ) * beta( j,p );
}
```

Notice how the roles of m and n are swapped and how working with the transpose replaces i,j with j,i, etc. However, we could have been clever: Instead of replacing i,j with j,i, etc., we could have instead swapped the order of these in the macro definitions: #define alpha(i,j) becomes #define alpha(j,i):

```
#define alpha( j,i ) A[ (j)*ldA + i ]
#define beta( j,i )  B[ (j)*ldB + i ]
#define gamma( j,i ) C[ (j)*ldC + i ]

void MyGemm( int m, int n, int k, double *A, int ldA,
                     double *B, int ldB, double *C, int ldC )
{
  for ( int i=0; i<m; i++ )
    for ( int j=0; j<n; j++ )
      for ( int p=0; p<k; p++ )
        gamma( i,j ) += alpha( i,p ) * beta( p,j );
}
```

Next, we can get right back to our original triple-nested loop by swapping the roles of A and B:

```
#define alpha( j,i ) A[ (j)*ldA + i ]
#define beta( j,i )  B[ (j)*ldB + i ]
#define gamma( j,i ) C[ (j)*ldC + i ]

void MyGemm( int m, int n, int k, double *B, int ldB,
                     double *A, int ldA, double *C, int ldC )
{
  for ( int i=0; i<m; i++ )
    for ( int j=0; j<n; j++ )
      for ( int p=0; p<k; p++ )
        gamma( i,j ) += alpha( i,p ) * beta( p,j );
}
```

The point is: Computing $C := AB + C$ with these matrices stored by columns is the same as computing $C := BA + C$ viewing the data as if stored by rows, making appropriate adjustments for the sizes of the matrix. With this insight, any matrix-matrix multiplication implementation we have encountered can be transformed into one that views the data as if stored by rows by redefining the macros that translate indexing into the arrays into how the matrices are stored, a swapping of the roles of A and B, and an adjustment to the sizes of the matrix.

Because we implement the packing with the macros, they actually can be used as is. What is left is to modify the inner kernel to use $m_R \times n_R = 6 \times 8$.

3.4.5 Loop unrolling (easy enough; see what happens)

There is some overhead that comes from having a loop in the micro-kernel. That overhead is in part due to the cost of updating the loop index p and the pointers to the buffers, MP_A and MP_B. It is also possible that "unrolling"

the loop will allow the compiler to rearrange more instructions in the loop and/or the hardware to better perform out-of-order computation because each iteration of the loop involves a branch instruction, where the branch depends on whether the loop is done. Branches get in the way of compiler optimizations and out-of-order computation by the CPU.

What is loop unrolling? In the case of the micro-kernel, unrolling the loop indexed by p by a factor two means that each iteration of that loop updates the micro-tile of C twice instead of once. In other words, each iteration of the unrolled loop performs two iterations of the original loop, and updates p+=2 instead of p++. Unrolling by a larger factor is the natural extension of this idea. Obviously, a loop that is unrolled requires some care in case the number of iterations was not a nice multiple of the unrolling factor. You may have noticed that our performance experiments always use problem sizes that are multiples of 48. This means that if you use a reasonable unrolling factor that divides into 48, you are probably going to be OK.

Similarly, one can unroll the loops in the packing routines.

See what happens!

Homework 3.4.5.1 Modify your favorite implementation so that it unrolls the loop in the micro-kernel with different unrolling factors.

(Note: I have not implemented this myself yet...)

3.4.6 Prefetching (tricky; seems to confuse the compiler...)

Elements of $\widetilde{A}$ in theory remain in the L2 cache. If you implemented the 6×8 kernel, you are using 15 out of 16 vector registers and the hardware almost surely prefetches the next element of $\widetilde{A}$ into the L1 cache while computing with the current one. One can go one step further by using the last unused vector register as well, so that you use two vector registers to load/broadcast two elements of $\widetilde{A}$. The CPU of most modern cores will execute "out of order" which means that at the hardware level instructions are rescheduled so that a "stall" (a hickup in the execution because, for example, data is not available) is covered by other instructions. It is highly likely that this will result in the current computation overlapping with the next element from $\widetilde{A}$ being prefetched (load/broadcast) into a vector register.

A second reason to prefetch is to overcome the latency to main memory for bringing in the next micro-tile of C. The Intel instruction set (and vector intrinsic library) includes instructions to prefetch data into an indicated level of cache. This is accomplished by calling the instruction

```
void _mm_prefetch(char *p, int i)
```

- The argument *p gives the address of the byte (and corresponding cache line) to be prefetched.
- The value i is the hint that indicates which level to prefetch to.
  - _MM_HINT_T0: prefetch data into all levels of the cache hierarchy.

- _MM_HINT_T1: prefetch data into level 2 cache and higher.
- _MM_HINT_T2: prefetch data into level 3 cache and higher, or an implementation-specific choice.

These are actually hints that a compiler can choose to ignore. In theory, with this a next micro-tile of C can be moved into a more favorable memory layer while a previous micro-tile is being updated. In theory, the same mechanism can be used to bring a next micro-panel of $\widetilde{B}$, which itself is meant to reside in the L3 cache, into the L1 cache during computation with a previous such micro-panel. In practice, we did not manage to make this work. It may work better to use equivalent instructions with in-lined assembly code.

For additional information, you may want to consult Intel's Intrinsics Guide.

3.4.7 Using in-lined assembly code

Even more control over where what happens can be attained by using in-lined assembly code. This will allow prefetching and how registers are used to be made more explicit. (The compiler often changes the order of the operations even when intrinsics are used in C.) Details go beyond this course. You may want to look at the BLIS micro-kernel for the Haswell achitecture, implemented with in-lined assembly code.

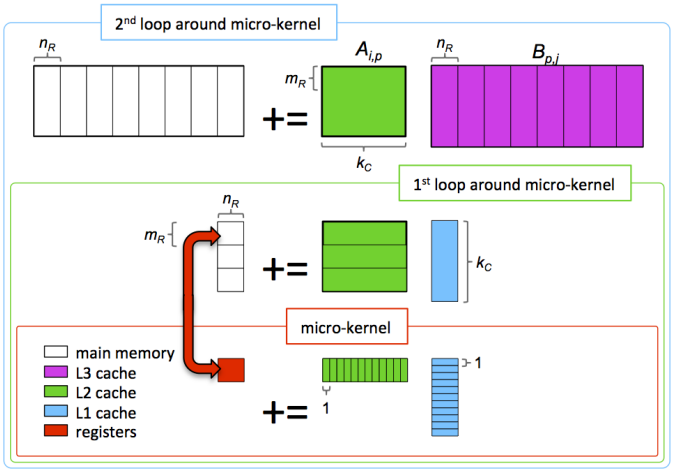
3.5 Enrichments

3.5.1 Goto's algorithm and BLIS

The overall approach described in Week 2 and Week 3 was first suggested by Kazushige Goto and published in

- [9] Kazushige Goto and Robert van de Geijn, On reducing TLB misses in matrix multiplication, Technical Report TR02-55, Department of Computer Sciences, UT-Austin, Nov. 2002.
- [10] Kazushige Goto and Robert van de Geijn, Anatomy of High-Performance Matrix Multiplication, ACM Transactions on Mathematical Software, Vol. 34, No. 3: Article 12, May 2008.

We refer to it as Goto's algorithm or the GotoBLAS algorithm (since he incorporated it in his implementation of the BLAS by the name GotoBLAS). His innovations included staging the loops so that a block of A remains in the L2 cache. He assembly-coded all the computation performed by the two loops around the micro-kernel



which he called the "inner-kernel". His accomplishments at the time were impressive enough that the New York Times did an article on him in 2005: "Writing the Fastest Code, by Hand, for Fun: A Human Computer Keeps Speeding Up Chips."

The very specific packing by micro-panels that we discuss in this week was also incorporated in Goto's implementations but the idea should be attributed to Greg Henry.

The BLIS library refactored Goto's algorithm into the five loops around the micro-kernel with which you are now very familiar, as described in a sequence of papers:

- [32] Field G. Van Zee and Robert A. van de Geijn, BLIS: A Framework for Rapidly Instantiating BLAS Functionality, ACM Journal on Mathematical Software, Vol. 41, No. 3, June 2015.
- [31] Field G. Van Zee, Tyler Smith, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John Gunnels, Tze Meng Low, Bryan Marker, Lee Killough, and Robert A. van de Geijn, The BLIS Framework: Experiments in Portability, ACM Journal on Mathematical Software, Vol. 42, No. 2, June 2016.
- [27] Field G. Van Zee and Tyler M. Smith, Implementing High-performance Complex Matrix Multiplication via the 3M and 4M Methods, ACM Transactions on Mathematical Software, Vol. 44, No. 1, pp. 7:1-7:36, July 2017.
- [30] Field G. Van Zee, Implementing High-Performance Complex Matrix Multiplication via the 1m Method, ACM Journal on Mathematical Software, in review.

While Goto already generalized his approach to accommodate a large set of special cases of matrix-matrix multiplication, known as the level-3 BLAS [5], as described in

- [11] Kazushige Goto and Robert van de Geijn, High-performance implementation of the level-3 BLAS, ACM Transactions on Mathematical Software, Vol. 35, No. 1: Article 4, July 2008.

His implementations required a considerable amount of assembly code to be written for each of these special cases. The refactoring of matrix-matrix multiplication in BLIS allows the micro-kernel to be reused across these special cases [32], thus greatly reducing the effort required to implement all of this functionality, at the expense of a moderate reduction in performance.

3.5.2 How to choose the blocking parameters

As you optimized your various implementations of matrix-matrix multiplication, you may have wondered how to optimally pick the various block sizes (MR,NR,MC, NC, and KC). Some believe that the solution is to explore the space of possibilities, timing each choice and picking the one that attains the best performance [33]. This is known as **autotuning**.

It has been shown that the block sizes for BLIS can be determined through analysis instead, as discussed in

- [18] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti, Analytical Modeling Is Enough for High-Performance {BLIS}, ACM Journal on Mathematical Software, Vol. 43, No. 2, Aug. 2016.

3.5.3 Alternatives to Goto's algorithm

In Unit 2.5.1, theoretical lower bounds on data movement incurred by matrix-matrix multiplication were discussed. You may want to go back and read [23], mentioned in that unit, in which it is discussed how Goto's algorithm fits into the picture.

In a more recent paper,

- [24] Tyler M. Smith and Robert A. van de Geijn, The MOMMS Family of Matrix Multiplication Algorithms, arXiv, 2019.

a family of practical matrix-matrix multiplication algorithms is discussed, of which Goto's algorithm is but one member. Analytical and empirical results suggest that for future CPUs, for which the ratio between the cost of a flop and the cost of moving data between the memory layers becomes even worse, other algorithms may become superior.

3.5.4 Practical implementation of Strassen's algorithm

Achieving a high performing, practical implementation of Strassen's algorithm, which was discussed in Unit 2.5.2, is nontrivial:

- As the number of levels of Strassen increases, the extra overhead associated with the additions becomes nontrivial.
- Now that you understand the cost of moving data between memory layers, you can also appreciate that there is cost related to moving data associated with the extra additions, if one is not careful.
- Accommodating matrices with sizes that are not a power of two (or a multiple of a power of two) becomes a can of worms.

Imagine trying to deal with the complexity of this recursive algorithm as you try to apply what you have learned so far in the course...

Until recently, the approach was to only perform a few levels of Strassen's algorithm, and to call a high-performance conventional matrix-matrix multiplication for the multiplications at the leaves of the recursion.

- If one performs one level of Strassen, this would reduce the number of flops required by $1 - 7/8 = 1/8 = 0.125$.
- If one performs two levels of Strassen, this would reduce the number of flops required by $1 - (7/8)^2 \approx 0.234$.

This yields some potential for improved performance. It comes at the expense of additional workspace due to the many intermediate matrices that need to be computed, which reduces the largest problem that can fit in memory.

In the recent paper

- [14] Jianyu Huang, Tyler Smith, Greg Henry, and Robert van de Geijn, Strassen's Algorithm Reloaded, International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16), 2016.

it was shown that the additions and subtractions that appear in Unit 2.5.2 can be incorporated into the packing that is inherent in the optimizations of matrix-matrix multiplication that you mastered so far in the course! This observation makes Strassen's algorithm much more practical than previously thought. Those interested in the details can read that paper.

If that paper interests you, you may also want to read

- [12] Jianyu Huang, Leslie Rice, Devin A. Matthews, Robert A. van de Geijn, Generating Families of Practical Fast Matrix Multiplication Algorithms, in Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS17), Orlando, FL, May 29-June 2, 2017.which discusses a family of Strassen-like algorithms.

The code discussed in that paper can be found at `https://github.com/flame/fmm-gen`.

3.6 Wrap Up

3.6.1 Additional exercises

The following are pretty challenging exercises. They stretch you to really think through the details. Or you may want to move on to Week 4, and return to these if you have extra energy in the end.

Homework 3.6.1.1 You will have noticed that we only time certain problem sizes when we execute our performance experiments. The reason is that the implementations of MyGemm do not handle "edge cases" well: when the problem size is not a nice multiple of the blocking sizes that appear in the micro-kernel: m_R, n_R, and k_C.

SInce our final, highest-performing implementations pack panels of B and

blocks of A, much of the edge case problems can be handled by "padding" micro-panels with zeroes when the remaining micro-panel is not a "full" one. Also, one can compute a full micro-tile of C and then add it to a partial such submatrix of C, if a micro-tile that is encountered is not a full $m_R \times n_R$ submatrix.

Reimplement MyGemm using these tricks so that it computes correctly for all sizes m, n, and k.

Homework 3.6.1.2 Implement the practical Strassen algorithm discussed in Unit 3.5.4.

3.6.2 Summary

3.6.2.1 The week in pictures

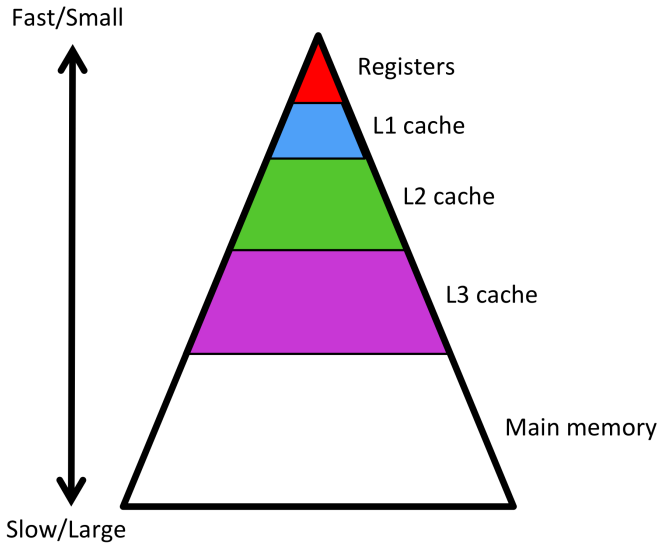


Figure 3.6.1: A typical memory hierarchy, with registers, caches, and main memory.

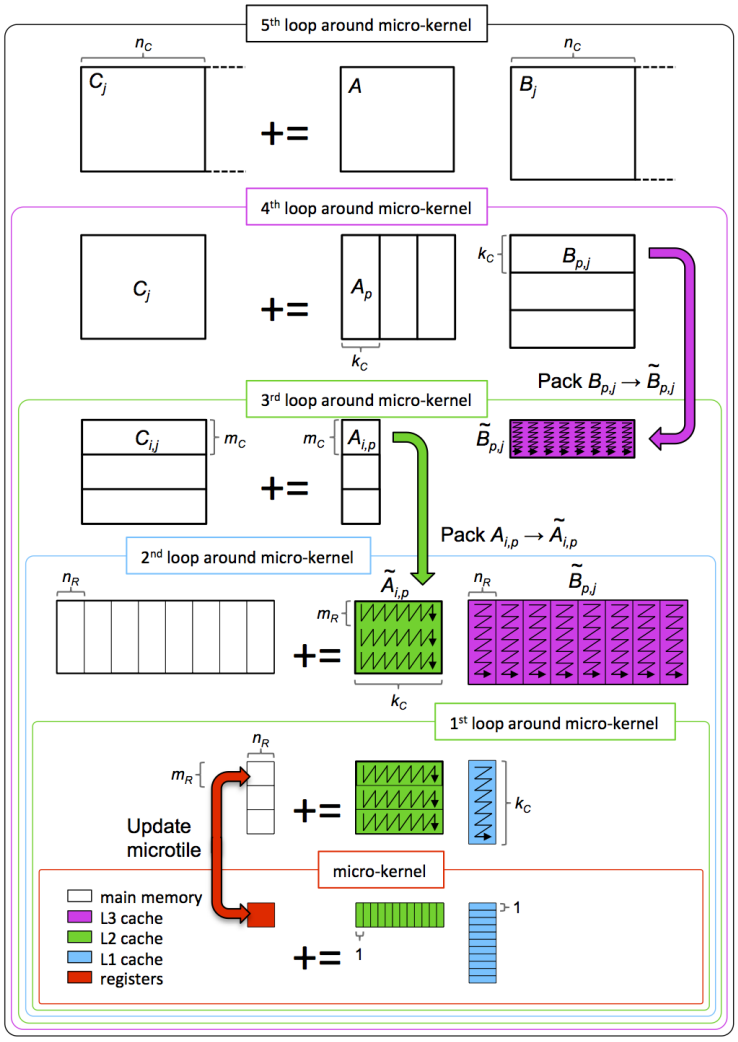


Figure 3.6.2: The five loops, with packing.

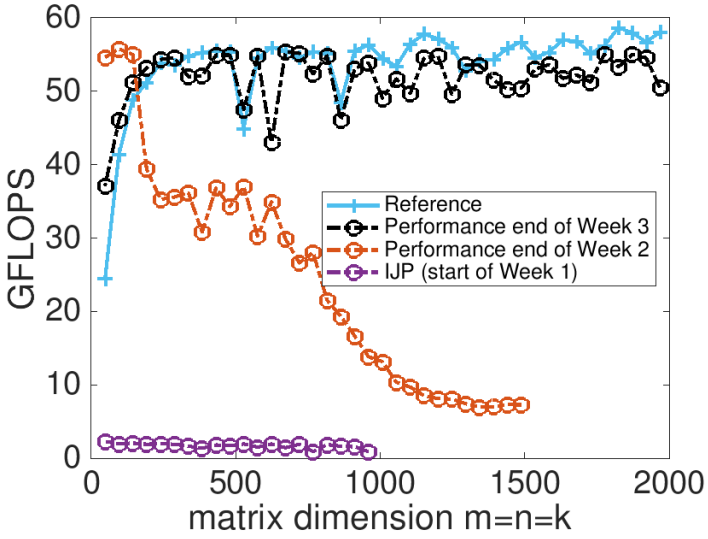


Figure 3.6.3: Performance on Robert's laptop at the end of Week 3.

Week 4

Multithreaded Parallelism

Only Weeks 0-2 have been released on edX. In order for our beta testers to comment on the materials for Week 4, the notes for this week are available to all. These notes are very much still under construction... Proceed at your own risk!

4.1 Opening Remarks

4.1.1 Launch

In this launch, you get a first exposure to parallelizing implementations of matrix-matrix multiplication with OpenMP. By examining the resulting performance data in different ways, you gain a glimpse into how to interpret that data.

Recall how the IJP ordering can be explained by partitioning matrices: Starting with $C := AB + C$, partitioning C and A by rows yields

$$\left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right) := \left(\begin{array}{c} \widetilde{a}_0^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T \end{array}\right) B + \left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right) = \left(\begin{array}{c} \widetilde{a}_0^T B + \widetilde{c}_0^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T B + \widetilde{c}_{m-1}^T \end{array}\right).$$

What we recognize is that the update of each row of C is completely independent and hence their computation can happen in parallel.

Now, it would be nice to simply tell the compiler "parallelize this loop, assigning the iterations to cores in some balanced fashion." For this, we use a standardized interface, OpenMP, to which you will be introduced in this week.

For now, all you need to know is:

- You need to include the header file omp.h

  ```
  #include "omp.h"
  ```

  at the top of a routine that uses OpenMP compiler directives and/or calls to OpenMP routines and functions.

- You need to insert the line

```
#pragma omp parallel for
```

before the loop the iterations of which can be executed in parallel.

- You need to set the environment variable OMP_NUM_THREADS to the number of threads to be used in parallel

```
export OMP_NUM_THREADS=4
```

before executing the program.

The compiler blissfully does the rest.

Remark 4.1.1 In this week, we get serious about collecting performance data. You will want to

- Make sure a laptop you are using is plugged into the wall, since running on battery can do funny things to your performance.
- Close down browsers and other processes that are running on your machine. I noticed that Apple's Activity Monitor can degrade performance a few percent.
- Learners running virtual machines will want to assign enough virtual CPUs.

Homework 4.1.1.1 In Assignments/Week4/C/ execute

```
export OMP_NUM_THREADS=1
make IJP
```

to collect fresh performance data for the IJP loop ordering.

Next, copy Gemm_IJP.c to Gemm_MT_IJP.c. (MT for Multithreaded). Include the header file omp.h at the top and insert the directive discussed above before the loop indexed with i. Compile and execute with the commands

```
export OMP_NUM_THREADS=4
make MT_IJP
```

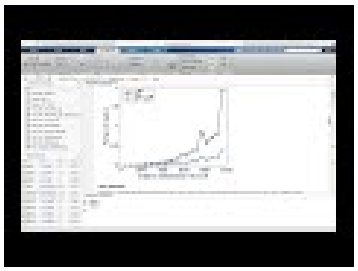
ensuring only up to four cores are used by setting OMP_NUM_THREADS=4. Examine the resulting performance with the Live Script data/Plot_MT_Launch.mlx.

You may want to vary the number of threads you use depending on how many cores your processor has. You will then want to modify that number in the Live Script as well.

Solution.

YouTube: https://www.youtube.com/watch?v=1M76YuKorlI

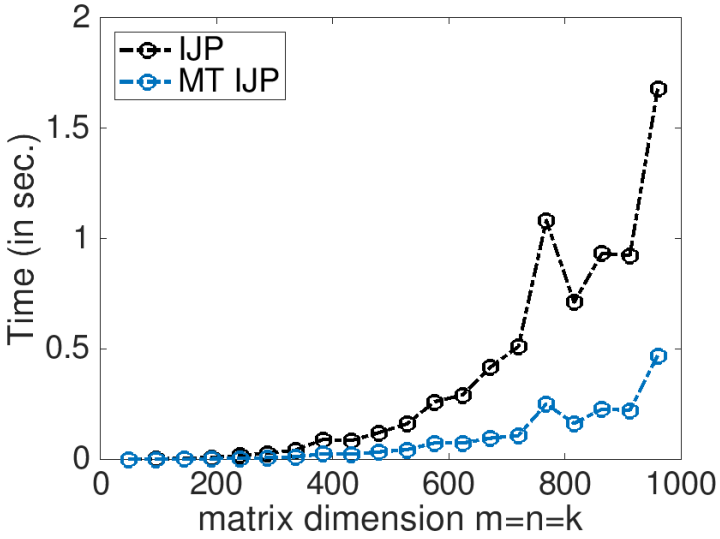
- Gemm_MT_IJP.c



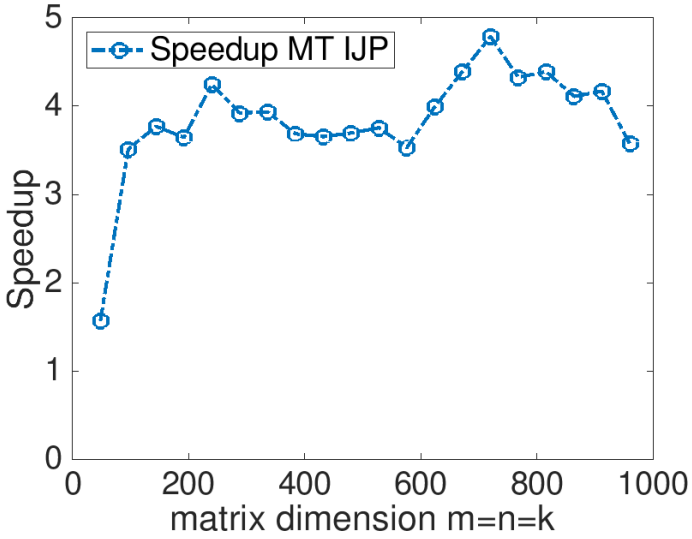
YouTube: https://www.youtube.com/watch?v=DXYDhNBGUjA

On Robert's laptop (using 4 threads):

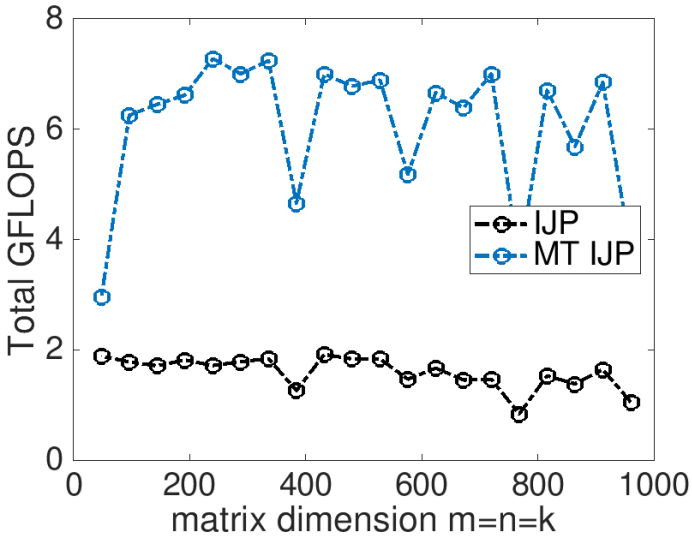
Time:



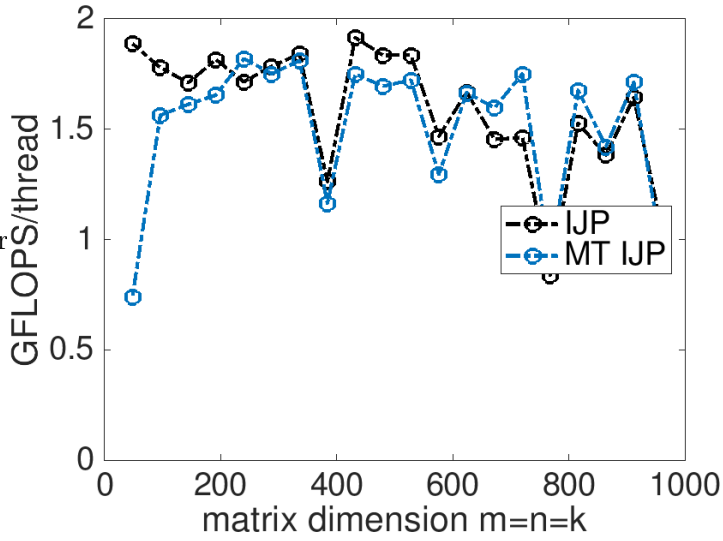
Speedup



Total GFLOPS



GFLOPS per thread



Homework 4.1.1.2 In Week 1, you discovered that the JPI ordering of loops yields the best performance of all (simple) loop orderings. Can the outer-most loop be parallelized much like the outer-most loop of the IJP ordering? Justify your answer. If you believe it can be parallelized, in Assignments/Week4/C/ execute

```
make JPI
```

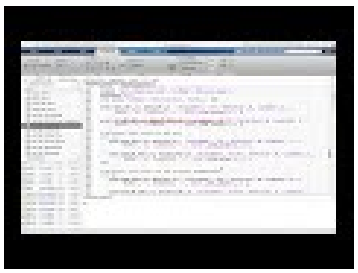
to collect fresh performance data for the JPI loop ordering.

Next, copy Gemm_JPI.c to Gemm_MT_JPI.c. Include the header file omp.h at the top and insert the directive discussed above before the loop indexed with j. Compile and execute with the commands

```
export OMP_NUM_THREADS=4
make MT_JPI
```

Examine the resulting performance with the Live Script data/Plot_MT_Launch.mlx by changing (0) to (1) (in four places).

Solution.



YouTube: https://www.youtube.com/watch?v=415ZdTOt8bs

The loop ordering JPI also allows for convenient parallelization of the outermost loop. Partition C and B by columns. Then

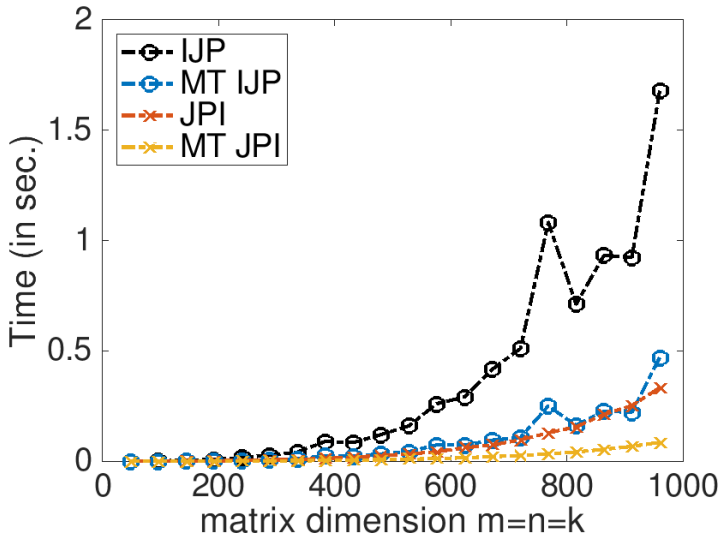
$$\begin{array}{rcl} \left(\begin{array}{c|c|c} c_0 & \cdots & c_{n-1} \end{array} \right) & := & A \left(\begin{array}{c|c|c} b_0 & \cdots & b_{n-1} \end{array} \right) + \left(\begin{array}{c|c|c} c_0 & \cdots & c_{n-1} \end{array} \right) \\ & = & \left(\begin{array}{c|c|c} Ab_0 + c_0 & \cdots & Ab_{n-1} + c_{n-1} \end{array} \right). \end{array}$$

This illustrates that the columns of C can be updated in parallel.

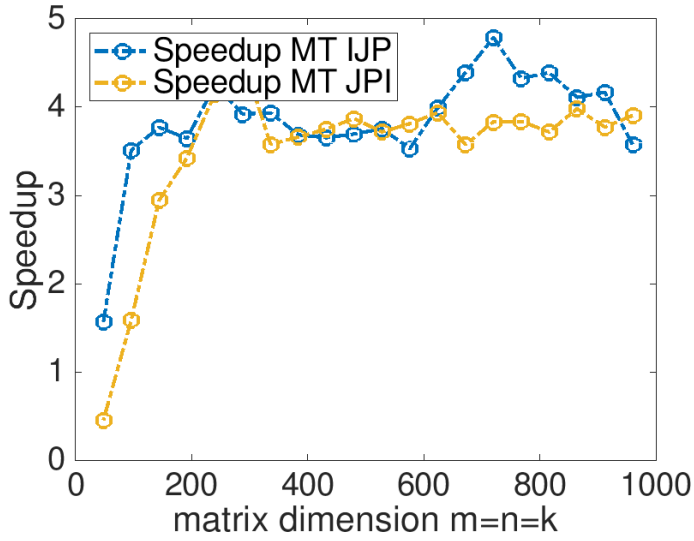
- Gemm_MT_JPI.c

On Robert's laptop (using 4 threads):

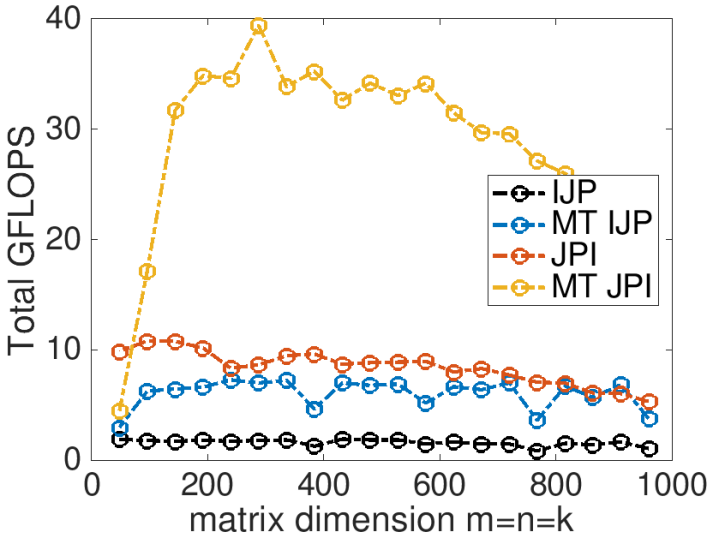
Time:



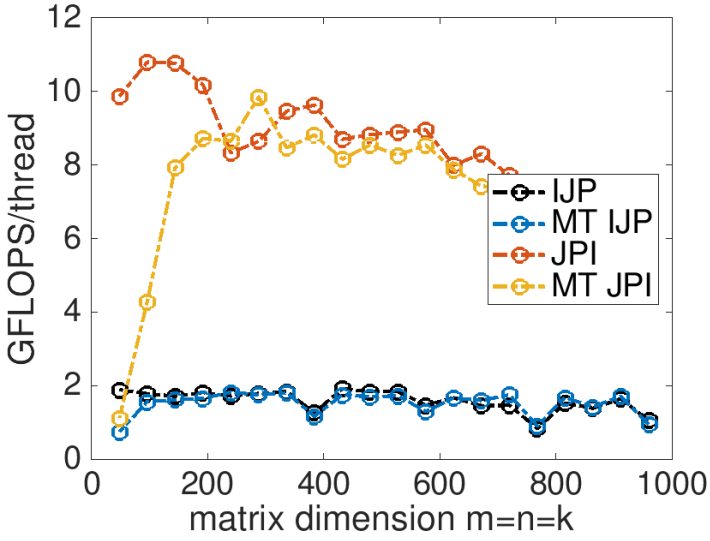
Speedup



Total GFLOPS

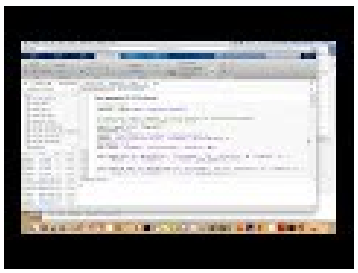


GFLOPS per thread



Homework 4.1.1.3 With Live Script data/Plot_MT_Launch.mlx, modify Plot_MT_Launch.mlx, by changing (0) to (1) (in four places), so that the results for the reference implementation are also displayed. Which multithreaded implementation yields the best speedup? Which multithreaded implementation yields the best performance?

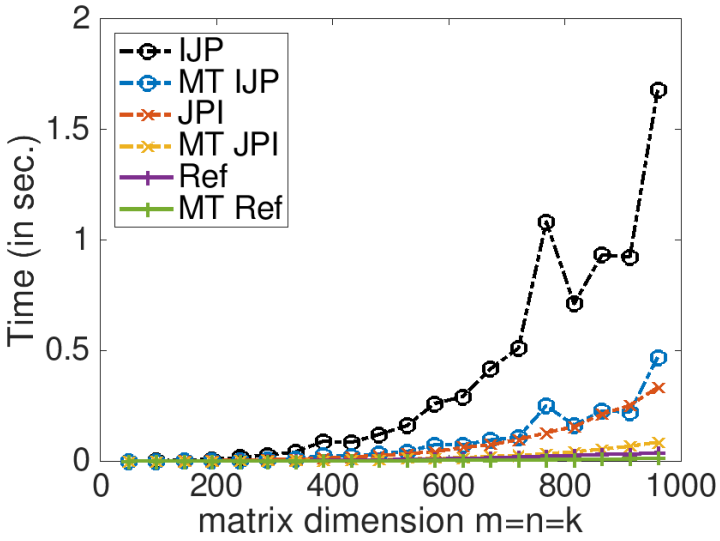
Solution.



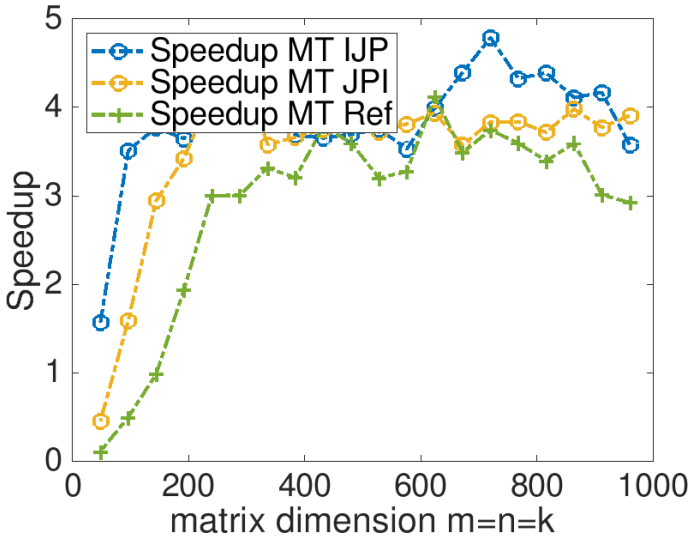
YouTube: `https://www.youtube.com/watch?v=rV8AKjvx4fg`

On Robert's laptop (using 4 threads):

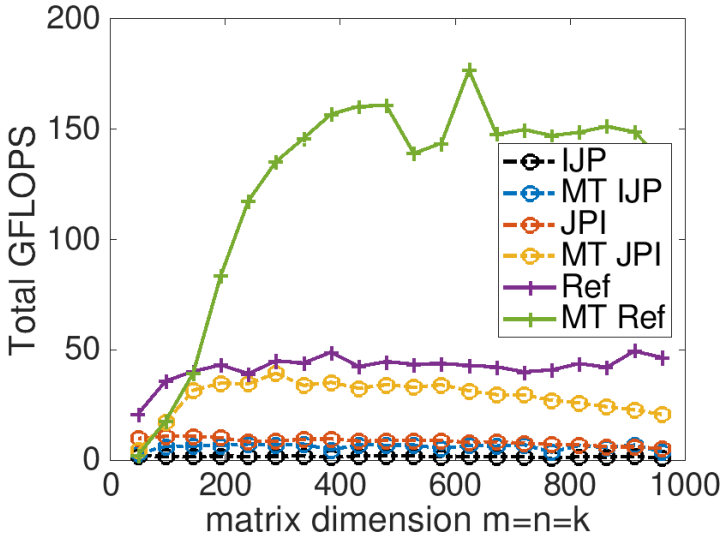
Time:



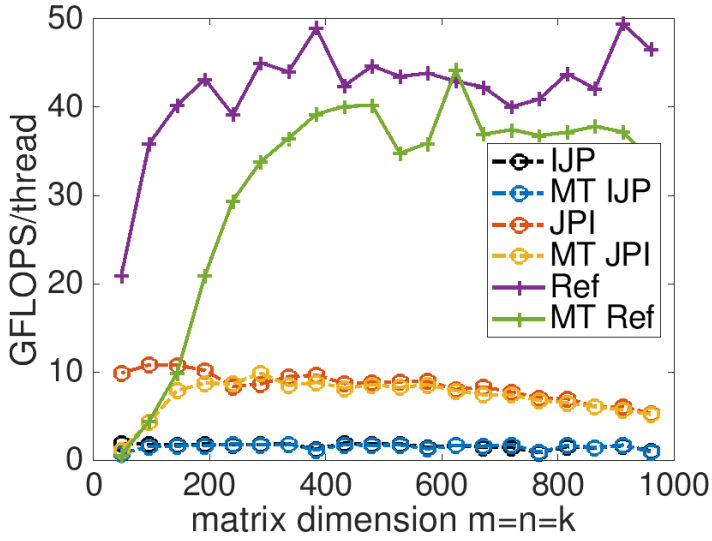
Speedup



Total GFLOPS



GFLOPS per thread



Remark 4.1.2 In later graphs, you will notice a higher peak GFLOPS rate for the reference implementation. Up to this point in the course, I had not fully realized the considerable impact having other applications open on my laptop has on the performance experiments. we get very careful with this starting in Section 4.3.

Learn more:

- For a general discussion of pragma directives, you may want to visit https://www.cprogramming.com/reference/preprocessor/pragma.html

4.1.2 Outline Week 4

4.1.3 What you will learn

In this week, we discover how to parallelize matrix-matrix multiplication among multiple cores of a processor.

Upon completion of this week, we will be able to

- Exploit multiple cores by multithreading your implementation.
- Direct the compiler to parallelize code sections with OpenMP.

- Parallelize the different loops and interpret the resulting performance.
- Experience when loops can be more easily parallelized and when more care must be taken.
- Apply the concepts of speedup and efficiency to implementations of matrix-matrix multiplication.
- Analyze limitations on parallel efficiency due to Ahmdahl's law.

The enrichments introduce us to

- The casting of other linear algebra operations in terms of matrix-matrix multiplication.
- The benefits of having a family of algorithms for a specific linear algebra operation and where to learn how to systematically derive such a family.
- Operations that resemble matrix-matrix multiplication that are encountered in Machine Learning, allowing the techniques to be extended.
- Parallelizing matrix-matrix multiplication for distributed memory architectures.
- Applying the learned techniques to the implementation of matrix-matrix multiplication on GPUs.

4.2 OpenMP

4.2.1 Of cores and threads

YouTube: https://www.youtube.com/watch?v=2V4HCXCPm0s

From Wikipedia: "A multi-core processor is a single computing component with two or more independent processing units called cores, which read and execute program instructions."

A thread is a thread of execution.

- It is an stream of program instructions and associated data. All these instructions may execute on a single core, multiple cores, or they may move from core to core.
- Multiple threads can execute on different cores or on the same core.

In other words:

- Cores are hardware components that can compute simultaneously.
- Threads are streams of execution that can compute simultaneously.

What will give you your solution faster than using one core? Using multiple cores!

4.2.2 Basics

OpenMP is a standardized API (Application Programming Interface) for creating multiple threads of execution from a single program. For our purposes, you need to know very little:

- There is a header file to include in the file(s):

  ```
  #include <omp.h>
  ```

- Directives to the compiler (pragmas):

  ```
  #pragma omp parallel
  #pragma omp parallel for
  ```

- A library of routines that can, for example, be used to inquire about the execution environment:

  ```
  omp_get_max_threads()
  omp_get_num_threads()
  omp_get_thread_num()
  ```

- Environment parameters that can be set before executing the program:

  ```
  export OMP_NUM_THREADS=4
  ```

  If you want to see what the value of this environment parameter is, execute

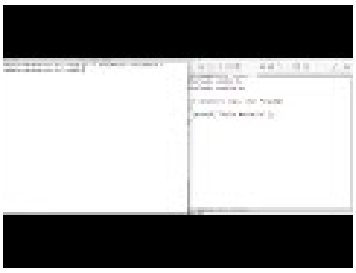
  ```
  echo $OMP_NUM_THREADS
  ```

Learn more:

- Search for "OpenMP" on Wikipedia.
- Tim Mattson's (Intel) "Introduction to OpenMP" (2013) on YouTube.
  - Slides: Intro_To_OpenMP_Mattson.pdf
  - Exercise files: Mattson_OMP_exercises.zip

We suggest you investigate OpenMP more after completing this week.

4.2.3 Hello World!



YouTube: https://www.youtube.com/watch?v=AEIYiu0wO5M

We will illustrate some of the basics of OpenMP via the old standby, the "Hello World!" program:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  printf( "Hello World!\n" );
}
```

Homework 4.2.3.1 In Week4/C/ compile HelloWorld.c with the command
`gcc -o HelloWorld.x HelloWorld.c`

and execute the resulting executable with

```
export OMP_NUM_THREADS=4
./HelloWorld.x
```

Solution. The output is

```
Hello World!
```

even though we indicated four threads are available for the execution of the program.

Homework 4.2.3.2 Copy the file HelloWorld.c to HelloWorld1.c. Modify it to add the OpenMP header file:
`#include "omp.h"`

at the top of the file. Compile it with the command

```
gcc -o HelloWorld1.x HelloWorld1.c
```

and execute it with

```
export OMP_NUM_THREADS=4
     ./HelloWorld1.x
```

Next, recompile and execute with

```
gcc -o HelloWorld1.x HelloWorld1.c -fopenmp
export OMP_NUM_THREADS=4
./HelloWorld1.x
```

Pay attention to the -fopenmp, which links the OpenMP library. What do you notice?

```
gcc -o HelloWorld1.x HelloWorld1.c -fopenmp
export OMP_NUM_THREADS=4
./HelloWorld1.x
```

(You don't need to export OMP_NUM_THREADS=4 every time you execute. We expose it so that you know exactly how many threads are available.)

Solution.

YouTube: https://www.youtube.com/watch?v=EcaLVnK_aws

- Assignments/Week4/Answers/HelloWorld1.c

In all cases, the output is

```
Hello World!
```

None of what you have tried so far resulted in any parallel execution because an OpenMP program uses a "fork and join" model: Initially, there is just one thread of execution. Multiple threads are deployed when the program reaches a parallel region, initiated by

```
#pragma omp parallel
{
  <command>
}
```

At that point, multiple threads are "forked" (initiated), each of which then performs the command <command> given in the parallel section. The parallel section here is the section of the code immediately after the #pragma directive bracketed by the "{" and "}" which C views as a single command (that may be composed of multiple commands within that region). Notice that the "{" and "}" are not necessary if the parallel region consists of a single command. At the end of the region, the threads are synchronized and "join" back into a single thread.

Homework 4.2.3.3 Copy the file HelloWorld1.c to HelloWorld2.c. Before the printf statement, insert

```
#pragma omp parallel
```

Compile and execute:

```
gcc -o HelloWorld2.x HelloWorld2.c -fopenmp
export OMP_NUM_THREADS=4
./HelloWorld2.x
```

What do you notice?

Solution.

YouTube: https://www.youtube.com/watch?v=7Sr8jfzGHtw

- Assignments/Week4/Answers/HelloWorld2.c

The output now should be

```
Hello World!
Hello World!
Hello World!
Hello World!
```

You are now running identical programs on four threads, each of which is printing out an identical message. Execution starts with a single thread that forks four threads, each of which printed a copy of the message. Obviously, this isn't very interesting since they don't collaborate to make computation that was previously performed by one thread faster.

Next, we introduce three routines with which we can extract information about the environment in which the program executes and information about a specific thread of execution:

- omp_get_max_threads() returns the maximum number of threads that are available for computation. It equals the number assigned to OMP_NUM_THREADS before executing a program.
- omp_get_num_threads() equals the number of threads in the current team: The total number of threads that are available may be broken up into teams that perform separate tasks.
- omp_get_thread_num() returns the index that uniquely identifies the thread that calls this function, among the threads in the current team. This index ranges from 0 to (omp_get_num_threads()-1). In other words, the indexing of the threads starts at zero.

In all our examples, omp_get_num_threads() equals omp_get_max_threads() in a parallel section.

Homework 4.2.3.4 Copy the file HelloWorld2.c to HelloWorld3.c. Modify the body of the main routine to

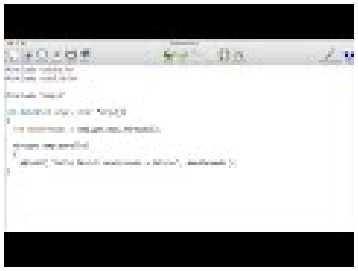
```
int maxthreads = omp_get_max_threads();

#pragma omp parallel
{
  int nthreads = omp_get_num_threads();
  int tid = omp_get_thread_num();

  printf( "Hello World! from %d of %d max_threads = %d \n\n",
                                                tid, nthreads, maxthreads );
}
```

Compile it and execute it:

```
export OMP_NUM_THREADS=4
gcc -o HelloWorld3.x HelloWorld3.c -fopenmp
./HelloWorld3.x
```

What do you notice?

Solution.



YouTube: https://www.youtube.com/watch?v=RsASfLwn4KM

- Assignments/Week4/Answers/HelloWorld3.c

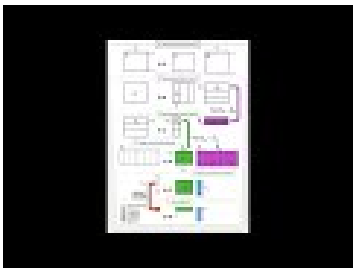
The output:

```
Hello World! from 0 of 4 max_threads = 4

Hello World! from 1 of 4 max_threads = 4

Hello World! from 3 of 4 max_threads = 4

Hello World! from 2 of 4 max_threads = 4
```

In the last exercise, there are four threads available for execution (since OMP_NUM_THREADS equals 4). In the parallel section, each thread assigns its index (rank) to its private variable tid. Each thread then prints out its copy of Hello World! Notice that the order of these printfs may not be in sequential order. This very simple example demonstrates how the work performed by a specific thread is determined by its index within a team.

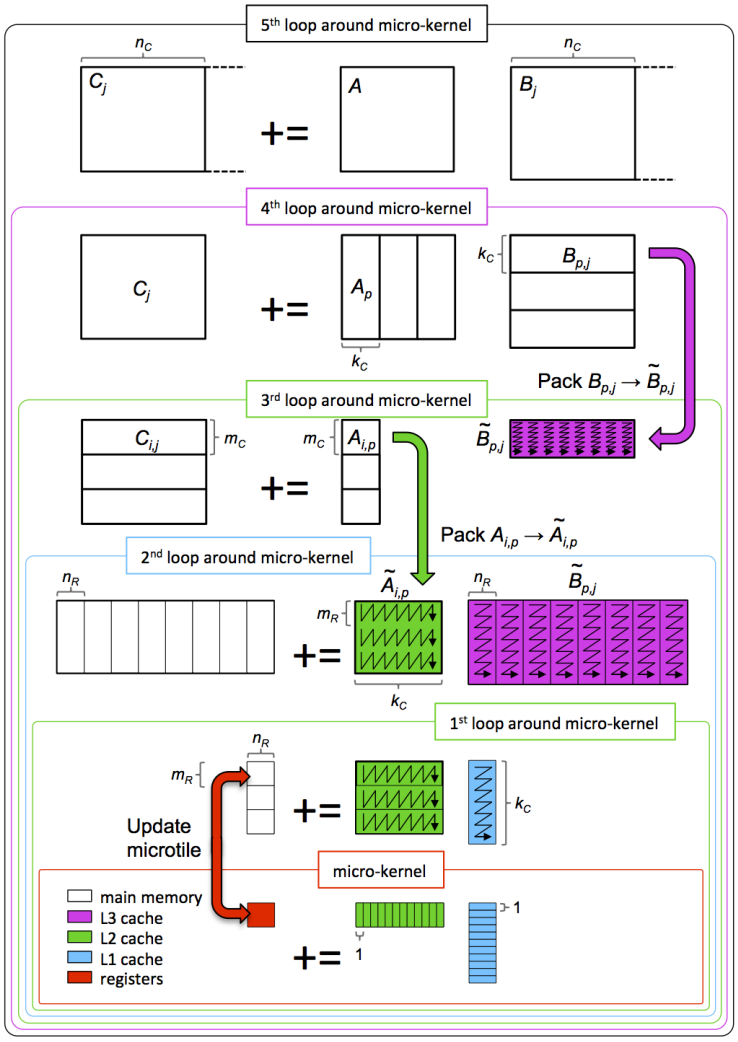
4.3 Multithreading Matrix Multiplication

4.3.1 Lots of loops to parallelize



YouTube: https://www.youtube.com/watch?v=2lLDVS7fGTA

Let's again consider the five loops around the micro-kernel:



What we notice is that there are lots of loops that can be parallelized. In this section we examine how parallelizing each individually impacts parallel performance.

Homework 4.3.1.1 In directory Assignments/Week4/C, execute

```
export OMP_NUM_THREADS=1
make Five_Loops_Packed_8x6Kernel
```

to collect fresh performance data. In the Makefile, MC and KC have been set to values that yield reasonable performance in our experience. Transfer the resulting output file (in subdirectory data) to Matlab Online Matlab Online and test it by running

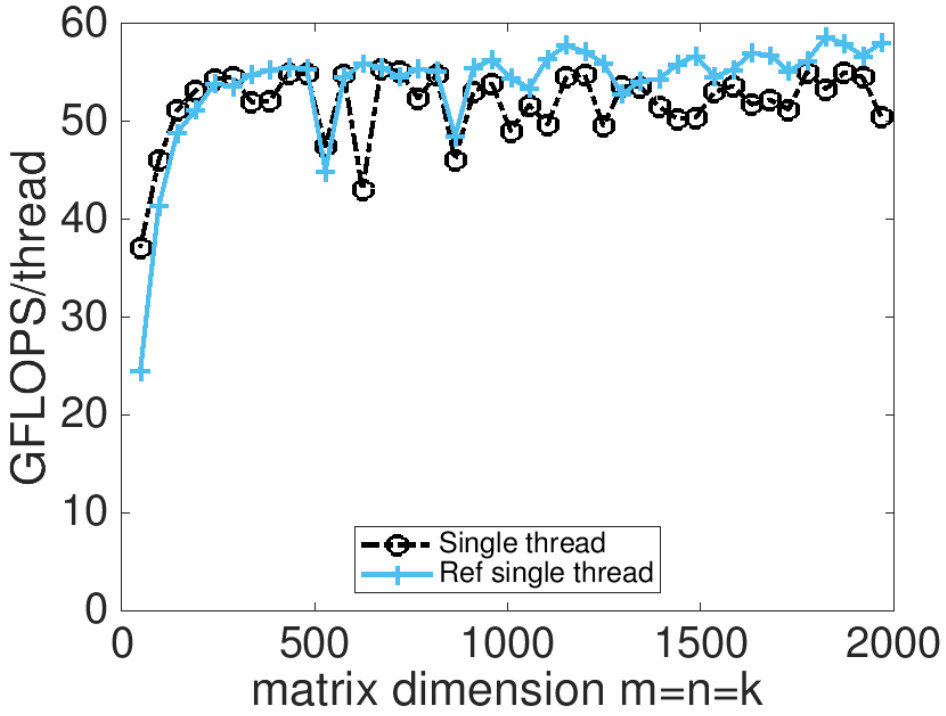
```
Plot_MT_Performance_8x6.mlx
```

If you decide also to test a 12x4 kernel or others, upload their data and run the corresponding Plot_MT_Performance_??x??.mlx to see the results. (If the .mlx file for your choice of $m_R \times n_R$ does not exist, you may want to copy one of the existing .mlx files and do a global search and replace.)

Solution.

YouTube: `https://www.youtube.com/watch?v=W7l7mxkEgg0`

On Robert's laptop:



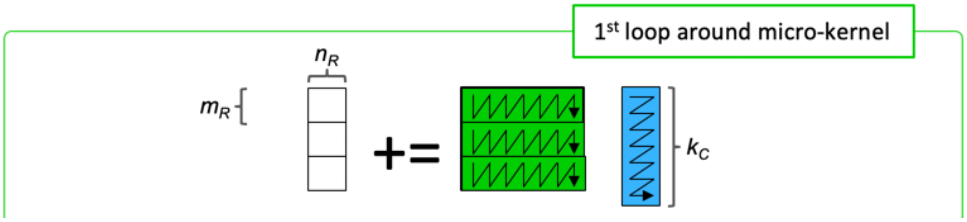
This reminds us: we are doing really well on a single core!

Remark 4.3.1 Here and in future homework, you can substitute 12x4 for 8x6 if you prefer that size of micro-tile.

Remark 4.3.2 It is at this point in my in-class course that I emphasize that one should be careful not to pronounce "parallelize" as "paralyze." You will notice that sometimes if you naively parallelize your code, you actually end up paralyzing it...

4.3.2 Parallelizing the first loop around the micro-kernel

Let us start by considering how to parallelize the first loop around the micro-kernel:



The situation here is very similar to that considered in Unit 4.1.1 when parallelizing the IJP loop ordering. There, we observed that if C and A are partitioned by rows, the matrix-matrix multiplication can be described by

$$\left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right) := \left(\begin{array}{c} \widetilde{a}_0^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T \end{array}\right) B + \left(\begin{array}{c} \widetilde{c}_0^T \\ \hline \vdots \\ \hline \widetilde{c}_{m-1}^T \end{array}\right) = \left(\begin{array}{c} \widetilde{a}_0^T B + \widetilde{c}_0^T \\ \hline \vdots \\ \hline \widetilde{a}_{m-1}^T B + \widetilde{c}_{m-1}^T \end{array}\right).$$

We then observed that the update of each row of C could proceed in parallel.

The matrix-matrix multiplication with the block of A and micro-panels of C and B performed by the first loop around the micro-kernel instead partitions the block of A into row (micro-)panels and the micro-panel of C into micro-tiles.

$$\left(\begin{array}{c} C_0 \\ \hline \vdots \\ \hline C_{M-1} \end{array}\right) := \left(\begin{array}{c} A_0 \\ \hline \vdots \\ \hline A_{M-1} \end{array}\right) B + \left(\begin{array}{c} C_0 \\ \hline \vdots \\ \hline C_{M-1} \end{array}\right) = \left(\begin{array}{c} A_0 B + C_0 \\ \hline \vdots \\ \hline A_{M-1} B + C_{M-1} \end{array}\right).$$

The bottom line: the updates of these micro-tiles can happen in parallel.

Homework 4.3.2.1 In directory Assignments/Week4/C,

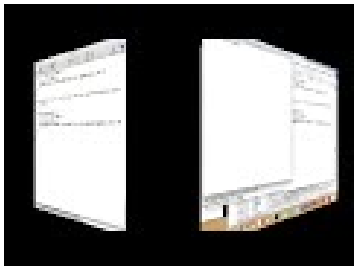
- Copy Gemm_Five_Loops_Packed_MRxNRKernel.c into Gemm_MT_Loop1_MRxNRKern
- Modify it so that the first loop around the micro-kernel is parallelized.
- Execute it with

```
export OMP_NUM_THREADS=4
make MT_Loop1_8x6Kernel
```

- Be sure to check if you got the right answer!

- View the resulting performance with data/Plot_MT_performance_8x6.mlx, uncommenting the appropriate sections.

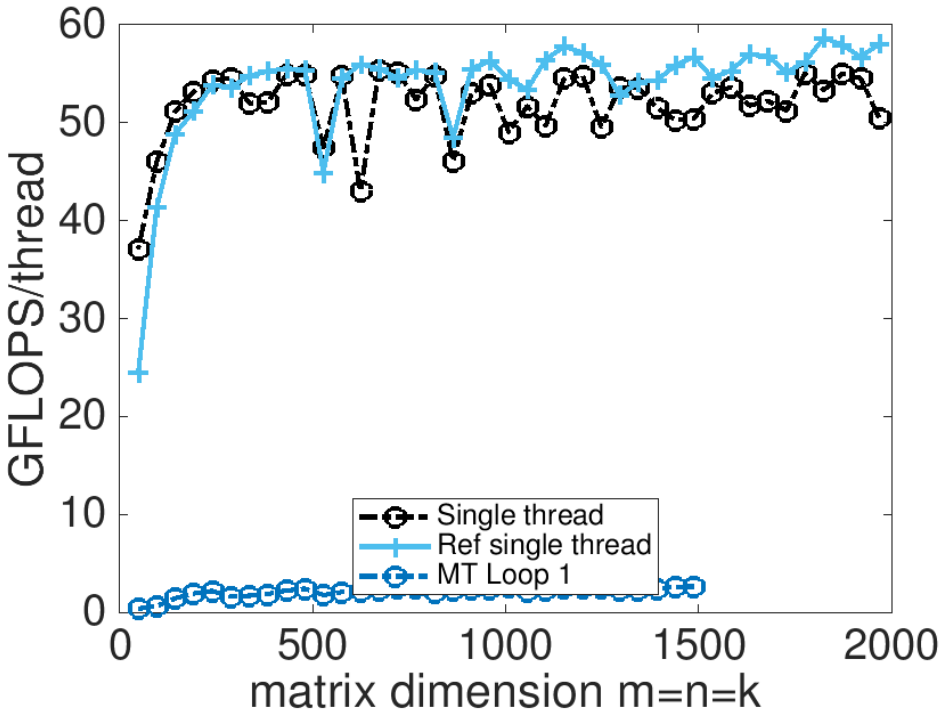
Solution.



YouTube: https://www.youtube.com/watch?v=WQEumfrQ6j4

- Assignments/Week4/Answers/Gemm_MT_Loop1_MRxNRKernel.c

On Robert's laptop (using 4 threads):



Parallelizing the first loop around the micro-kernel yields poor performance. A number of issues get in the way:

- Each task (the execution of a micro-kernel) that is performed by a thread is relatively small and as a result the overhead of starting a parallel section (spawning threads and synchronizing upon their completion) is nontrivial.
- In our experiments, we chose MC=72 and MR=8 and hence there are at most $72/8 = 9$ iterations that can be performed by the threads. This leads to load imbalance unless the number of threads being used divides 9 evenly. It also means that no more than 9 threads can be usefully

employed. The bottom line: there is limited parallelism to be found in the loop.

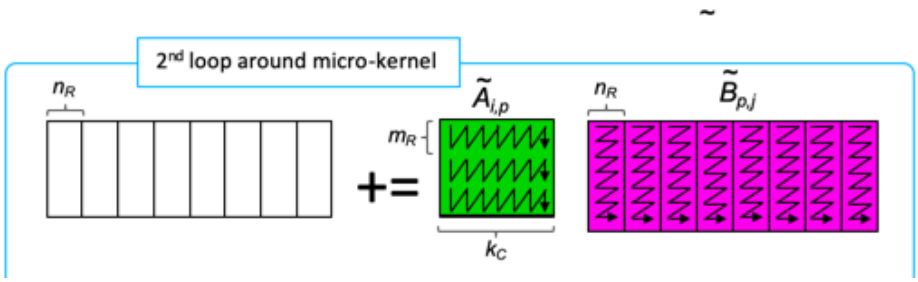
This has a secondary effect: Each micro-panel of B is reused relatively few times by a thread, and hence the cost of bringing it into a core's L1 cache is not amortized well.

- Each core only uses a few of the micro-panels of A and hence only a fraction of the core's L2 cache is used (if each core has its own L2 cache).

The bottom line: the first loop around the micro-kernel is not a good candidate for parallelization.

4.3.3 Parallelizing the second loop around the micro-kernel

Let us next consider how to parallelize the second loop around the micro-kernel:

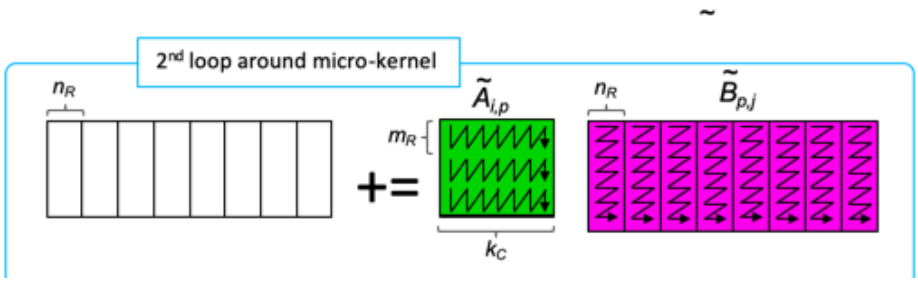


This time, the situation is very similar to that considered in Unit 4.1.1 when parallelizing the JPI loop ordering. There, in Homework 4.1.1.2 we observed that if C and B are partitioned by columns, the matrix-matrix multiplication can be described by

$$\begin{array}{rcl} \left(\begin{array}{c|c|c} c_0 & \cdots & c_{n-1} \end{array} \right) & := & A \left(\begin{array}{c|c|c} b_0 & \cdots & b_{n-1} \end{array} \right) + \left(\begin{array}{c|c|c} c_0 & \cdots & c_{n-1} \end{array} \right) \\ & = & \left(\begin{array}{c|c|c} Ab_0 + c_0 & \cdots & Ab_{n-1} + c_{n-1} \end{array} \right). \end{array}$$

We then observed that the update of each column of C can proceed in parallel.

The matrix-matrix multiplication



performed by the second loop around the micro-kernel instead partitions the row panel of C and the row panel of B into micro-panels.

$$\begin{array}{rcl} \left(\begin{array}{c|c|c} C_0 & \cdots & C_{N-1} \end{array} \right) & := & A \left(\begin{array}{c|c|c} B_0 & \cdots & B_{N-1} \end{array} \right) + \left(\begin{array}{c|c|c} C_0 & \cdots & C_{N-1} \end{array} \right) \\ & = & \left(\begin{array}{c|c|c} AB_0 + C_0 & \cdots & AB_{N-1} + C_{N-1} \end{array} \right). \end{array}$$

The bottom line: the updates of the micro-panels of C can happen in parallel.

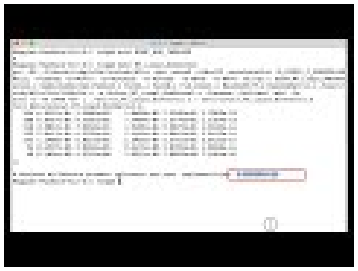
Homework 4.3.3.1 In directory Assignments/Week4/C,

- Copy Gemm_MT_Loop1_MRxNRKernel.c. into Gemm_MT_Loop2_MRxNRKernel.c.
- Modify it so that only the second loop around the micro-kernel is parallelized.
- Execute it with

```
export OMP_NUM_THREADS=4
make MT_Loop2_8x6Kernel
```

- Be sure to check if you got the right answer!
- View the resulting performance with data/Plot_MT_performance_8x6.mlx, changing 0 to 1 for the appropriate section.

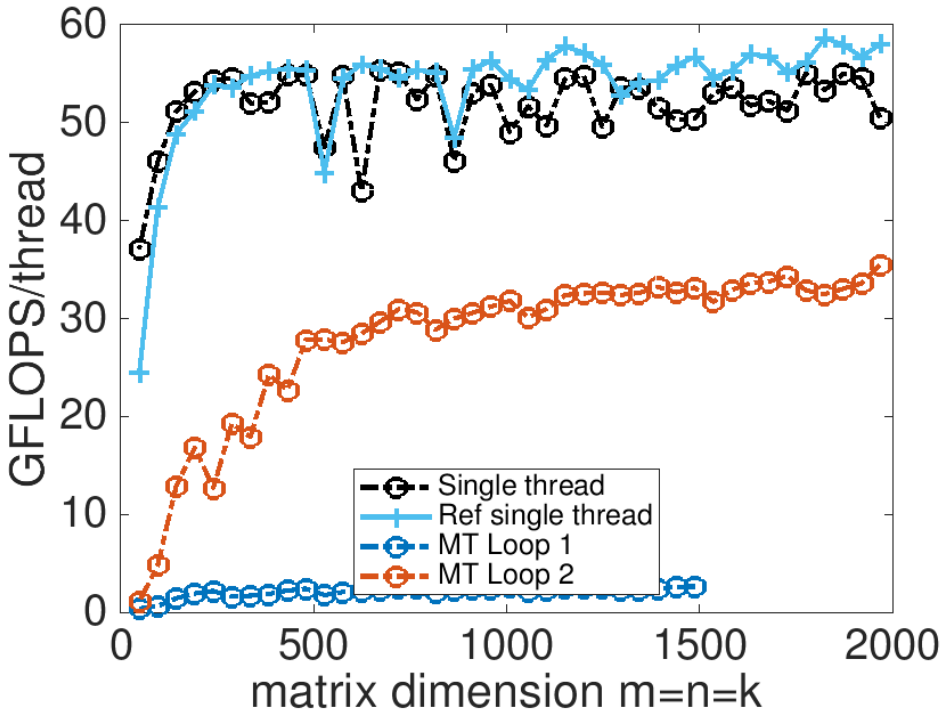
Solution.



YouTube: https://www.youtube.com/watch?v=_v6OY4GLkHY

- Assignments/Week4/Answers/Gemm_MT_Loop2_MRxNRKernel.c

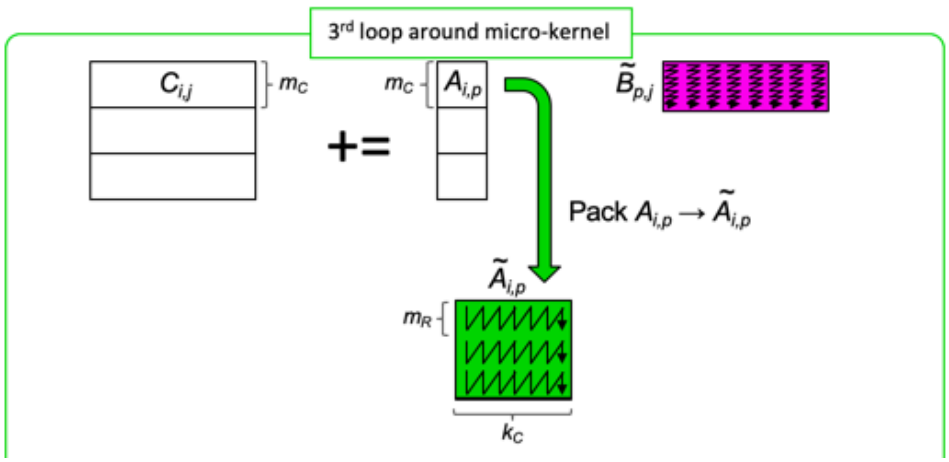
On Robert's laptop (using 4 threads):



Parallelizing the second loop appears to work very well. The granuarity of the computation in each iteration is larger. The ratio between NC and NR is typically large: in the makefile I set NC=2016 and NR=6, so that there are $2016/6 = 336$ tasks that can be scheduled to the threads. The only issue is that all cores load their L2 cache with the same block of A, which is a waste of a valuable resource (the aggregate size of the L2 caches).

4.3.4 Parallelizing the third loop around the micro-kernel

Moving right along, consider how to parallelize the third loop around the micro-kernel:



This reminds us of Unit 4.1.1 (parallelizing the IJP loop) and Unit 4.3.2

(paralellizing the first loop around the micro-kernel),

$$\left(\begin{array}{c} C_0 \\ \hline \vdots \\ \hline C_{M-1} \end{array}\right) := \left(\begin{array}{c} A_0 \\ \hline \vdots \\ \hline A_{M-1} \end{array}\right) B + \left(\begin{array}{c} C_0 \\ \hline \vdots \\ \hline C_{M-1} \end{array}\right) = \left(\begin{array}{c} A_0 B + C_0 \\ \hline \vdots \\ \hline A_{M-1} B + C_{M-1} \end{array}\right),$$

except that now the sizes of the row panels of C and blocsk of A are larger. The bottom line: the updates of the row-panels of C can happen in parallel.

Homework 4.3.4.1 In directory Assignments/Week4/C,

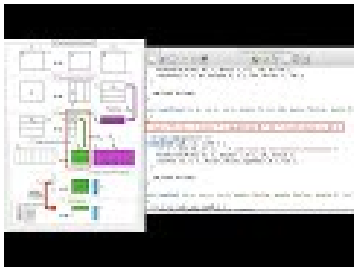
- Copy Gemm_MT_Loop2_MRxNRKernel.c. into Gemm_MT_Loop3_MRxNRKernel.c.
- Modify it so that only the third loop around the micro-kernel is parallelized.
- Execute it with

  ```
  export OMP_NUM_THREADS=4
  make MT_Loop3_8x6Kernel
  ```

- Be sure to check if you got the right answer! Parallelizing this loop is a bit trickier... When you get frustrated, look at the hint. And when you get really frustrated, watch the video in the solution.
- View the resulting performance with data/Plot_MT_performance_8x6.mlx, uncommenting the appropriate sections.

Hint. Parallelizing the third loop is a bit trickier. Likely, you started by inserting the #pragma statement and got the wrong answer. The problem lies with the fact that all threads end up packing a different block of A into the same buffer $\widetilde{A}$. How do you overcome this?

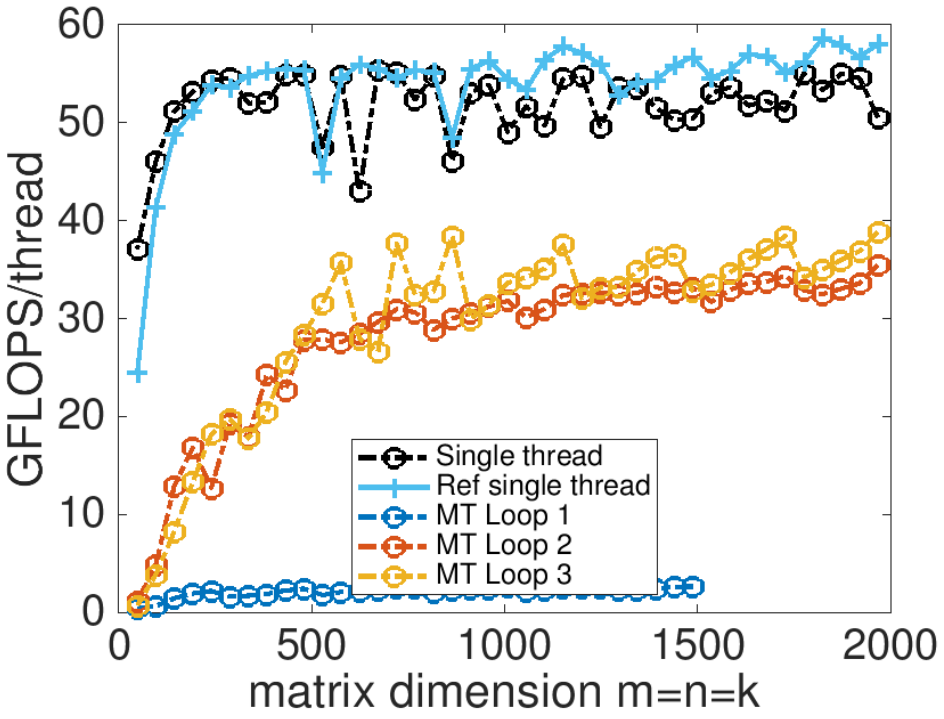
Solution.

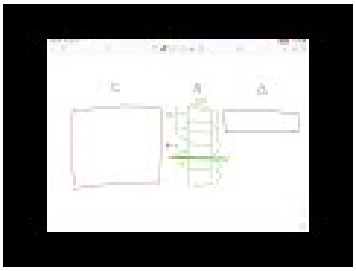


YouTube: https://www.youtube.com/watch?v=ijN0ORHroRo

- Assignments/Week4/Answers/Gemm_MT_Loop3_MRxNRKernel_simple.c

On Robert's laptop (using 4 threads):





YouTube: `https://www.youtube.com/watch?v=Mwv1RpEoU4Q`

So what is the zigzagging in the performance graph all about? This has to do with the fact that m/MC is relatively small. If $m = 650$, $MC = 72$ (which is what it is set to in the makefile), and we use 4 threads, then $m/MC = 9.03$ and hence two threads are assigned the computation related to two blocks of A, one thread is assigned three such blocks, and one thead is assigned a little more than two blocks. This causes load imbalance, which is the reason for the zigzagging in the curve.

So, you need to come up with a way so that most computation is performed using full blocks of A (with MC rows each) and the remainder is evenly distributed amongst the threads.

Homework 4.3.4.2

- Copy Gemm_MT_Loop3_MRxNRKernel.c. into Gemm_MT_Loop3_MRxNRKernel_simp to back up the simple implementation. Now go back to Gemm_MT_Loop3_MRxNRKernel.c and modify it so as to smooth the performance, inspired by the last video.

- Execute it with

```
export OMP_NUM_THREADS=4
```

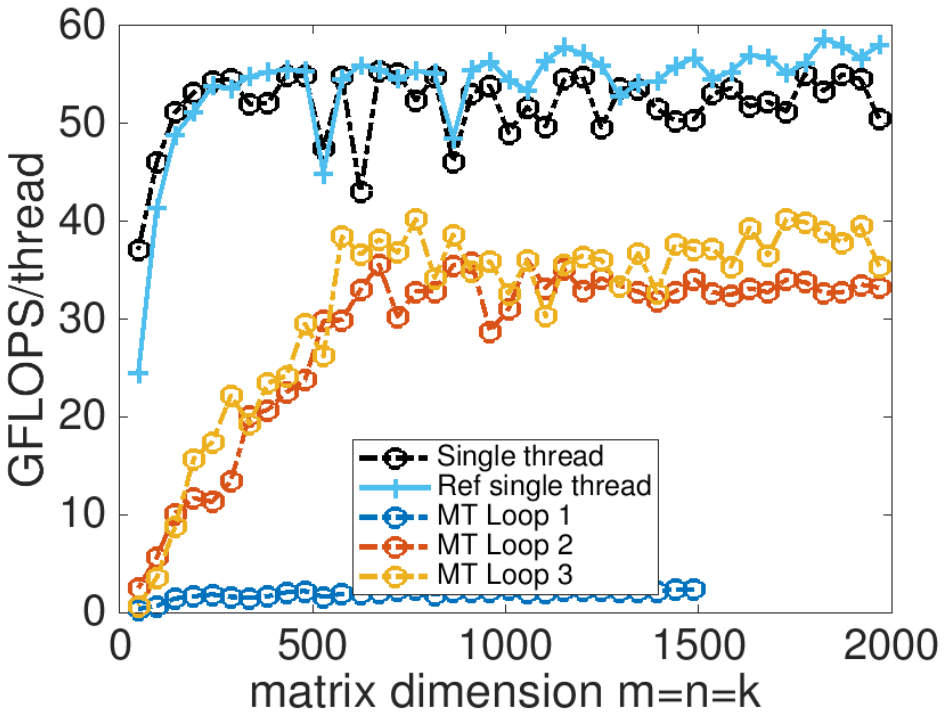
```
make MT_Loop3_8x6Kernel
```

- Be sure to check if you got the right answer!
- View the resulting performance with data/Plot_MT_performance_8x6.mlx.
- You may want to store the new version, once it works, in Gemm_MT_Loop3_MRxNRKernel_smooth.c

Solution.

- Assignments/Week4/Answers/Gemm_MT_Loop3_MRxNRKernel.c

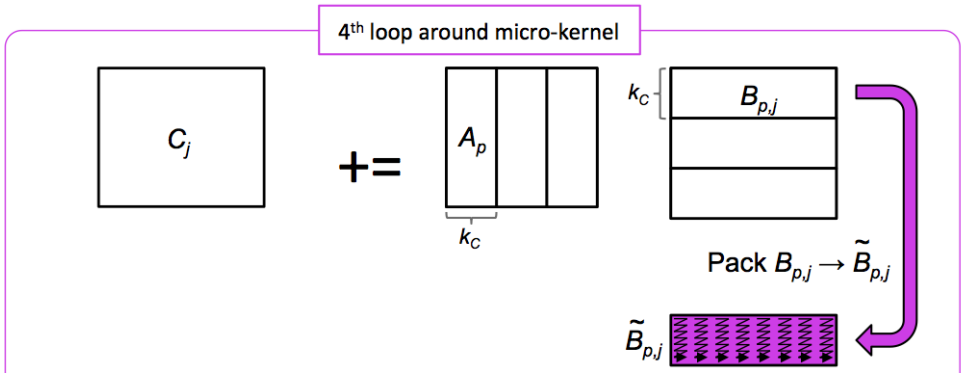
On Robert's laptop (using 4 threads):



What you notice is that the performance is much smoother. Each thread now fills most of its own L2 cache with a block of A. They share the same row panel of B (packed into $\widetilde{B}$). Notice that the packing of that panel of B is performed by a single thread. We'll get back to that in Section 4.4.

4.3.5 Parallelizing the fourth loop around the micro-kernel

Next, consider how to parallelize the fourth loop around the micro-kernel:



We can describe this picture by partitioning A into column panels and B into row panels:

$$C := \left(A_0 \mid \cdots \mid A_{m-1} \right) \left(\begin{array}{c} B_0 \\ \hline \vdots \\ \hline B_{m-1} \end{array} \right) + C = A_0 B_0 + \cdots + A_{K-1} B_{K-1} + C.$$

Given, for example, four threads, the thread with id 0 can compute

$$A_0 B_0 + A_4 B_4 + \cdots + C$$

while the thread with id 1 computes

$$A_1 B_1 + A_5 B_5 + \cdots +$$

and so forth.

Homework 4.3.5.1 In directory Assignments/Week4/C,

- Copy Gemm_MT_Loop3_MRxNRKernel.c. into Gemm_MT_Loop4_MRxNRKernel.c.
- Modify it so that only the fourth loop around the micro-kernel is parallelized.
- Execute it with

  ```
  export OMP_NUM_THREADS=4
  make MT_Loop4_8x6Kernel
  ```

- Be sure to check if you got the right answer! Parallelizing this loop is a bit trickier... Even trickier than parallelizing the third loop around the micro-kernel... Don't spend too much time on it before looking at the hint.
- View the resulting performance with data/Plot_MT_performance_8x6.mlx, uncommenting the appropriate sections.

Hint. Initially you may think that parallelizing this is a matter of changing how the temporary space $\widetilde{B}$ is allocated and used. But then you find that even that doesn't give you the right answer... The problem is that all iterations update the same part of C and this creates something called a "race" condition.

To update C, you read a micro-tile of C into registers, you update it, and you write it out. What if in the mean time another thread reads the same micro-tile of C? It would be reading the old data...

The fact is that Robert has yet to find the courage to do this homework himself... Perhaps you too should skip it.

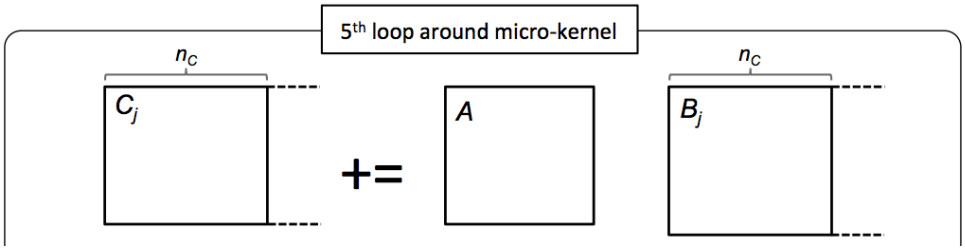
Solution. I have no solution to share at this moment...

The problem is that in the end all these contributions need to be added for the final result. Since all threads update the same array in which C is stored, then you never know if a given thread starts updating an old copy of a micro-tile of C while another thread is still updating it. This is known as a race condition.

This doesn't mean it can't be done, nor that it isn't worth pursuing. Think of a matrix-matrix multiplication where m and n are relatively small and k is large. Then this fourth loop around the micro-kernel has more potential for parallelism than the other loops. It is just more complicated and hence should wait until you learn more about OpenMP (in some other course).

4.3.6 Parallelizing the fifth loop around the micro-kernel

Finally, let us consider how to parallelize the fifth loop around the micro-kernel:



This time, the situation is very similar to those considered in Unit 4.1.1 (parallelizing the JPI loop ordering) and Unit 4.3.3 (parallelizing the second loop around the micro-kernel). Here we again partition

$$\begin{array}{rcl} \left(\begin{array}{c|c|c} C_0 & \cdots & C_{N-1} \end{array} \right) & := & A \left(\begin{array}{c|c|c} B_0 & \cdots & B_{N-1} \end{array} \right) + \left(\begin{array}{c|c|c} C_0 & \cdots & C_{N-1} \end{array} \right) \\ & = & \left(\begin{array}{c|c|c} AB_0 + C_0 & \cdots & AB_{N-1} + C_{N-1} \end{array} \right). \end{array}$$

and observe that the updates of the submatrices of C can happen in parallel.

Homework 4.3.6.1 In directory Assignments/Week4/C,

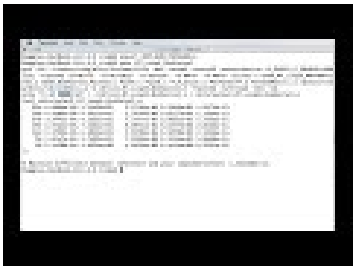
- Copy Gemm_MT_Loop2_MRxNRKernel.c. into Gemm_MT_Loop5_MRxNRKernel.c.
- Modify it so that only the fifth loop around the micro-kernel is parallelized.
- Execute it with

```
export OMP_NUM_THREADS=4
make MT_Loop5_8x6Kernel
```

- Be sure to check if you got the right answer!

- View the resulting performance with data/Plot_MT_performance_8x6.mlx, uncommenting the appropriate sections.

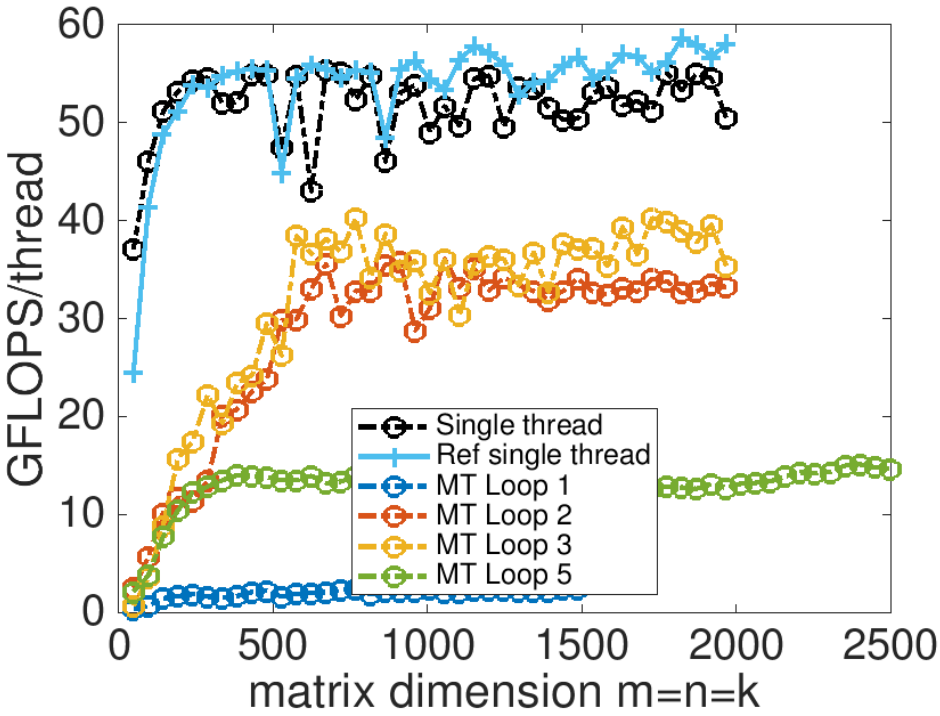
Solution.

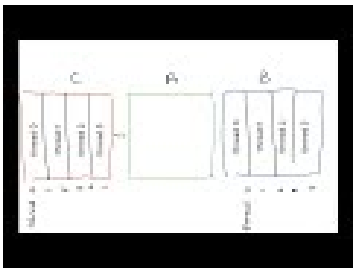


YouTube: https://www.youtube.com/watch?v=U9GqqOInHSA

- Assignments/Week4/Answers/Gemm_MT_Loop5x_MRxNRKernel.c

On Robert's laptop (using 4 threads):





YouTube: https://www.youtube.com/watch?v=Pbt3525NixI

The zigzagging in the performance graph is now so severe that you can't

even see it yet by the time you test matrices with $m = n = k = 2000$. This has to do with the fact that n/NC is now very small. If $n = 2000$, $NC = 2016$ (which is what it is set to in the makefile), and we use 4 threads, then $n/NC \approx 1$ and hence only one thread is assigned any computation.

So, you need to come up with a way so that most computation is performed using full blocks of C (with NC columns each) and the remainder is evenly distributed amongst the threads.

Homework 4.3.6.2

- Copy Gemm_MT_Loop5_MRxNRKernel.c. into Gemm_MT_Loop5_MRxNRKernel_simple.c to back up the simple implementation. Now go back to Gemm_MT_Loop5_MRxNRKernel.c and modify it so as to smooth the performance, inspired by the last video.
- Execute it with

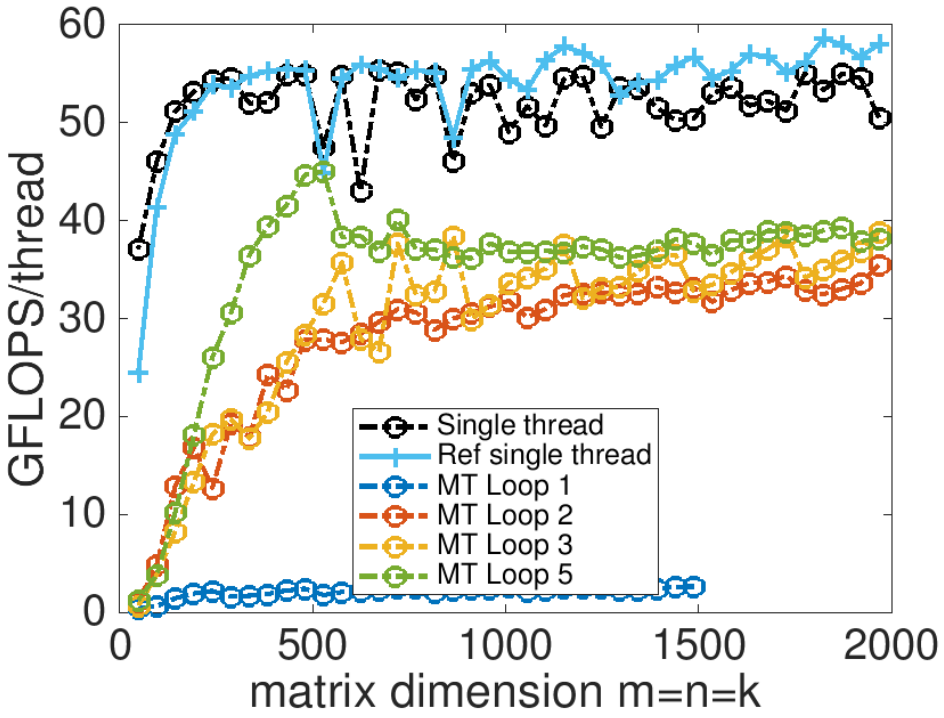
```
export OMP_NUM_THREADS=4
make MT_Loop5_8x6Kernel
```

- Be sure to check if you got the right answer!
- View the resulting performance with data/Plot_MT_performance_8x6.mlx.

Solution.

- Assignments/Week4/Answers/Gemm_MT_Loop5_MRxNRKernel_smooth.c

On Robert's laptop (using 4 threads):

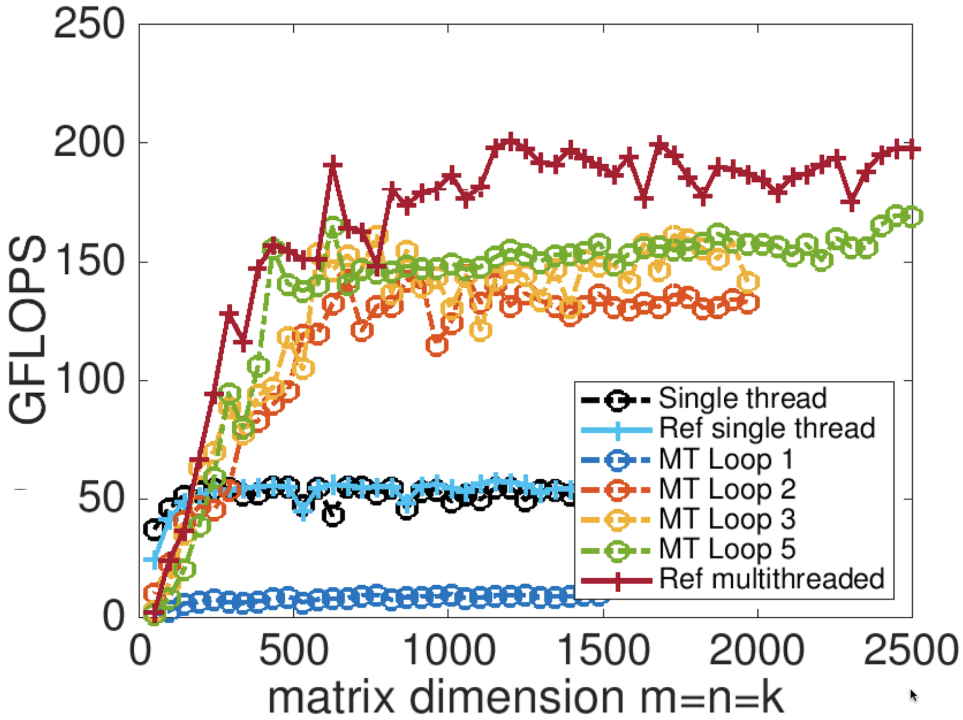


What we notice is that parallelizing the fifth loop gives good performance

when using four threads. However, once one uses a lot of threads, the part of the row panel of B assigned to each thread becomes small, which means that the any overhead associated with packing and/or bringing a block of A into the L2 cache may not be amortized over enough computation.

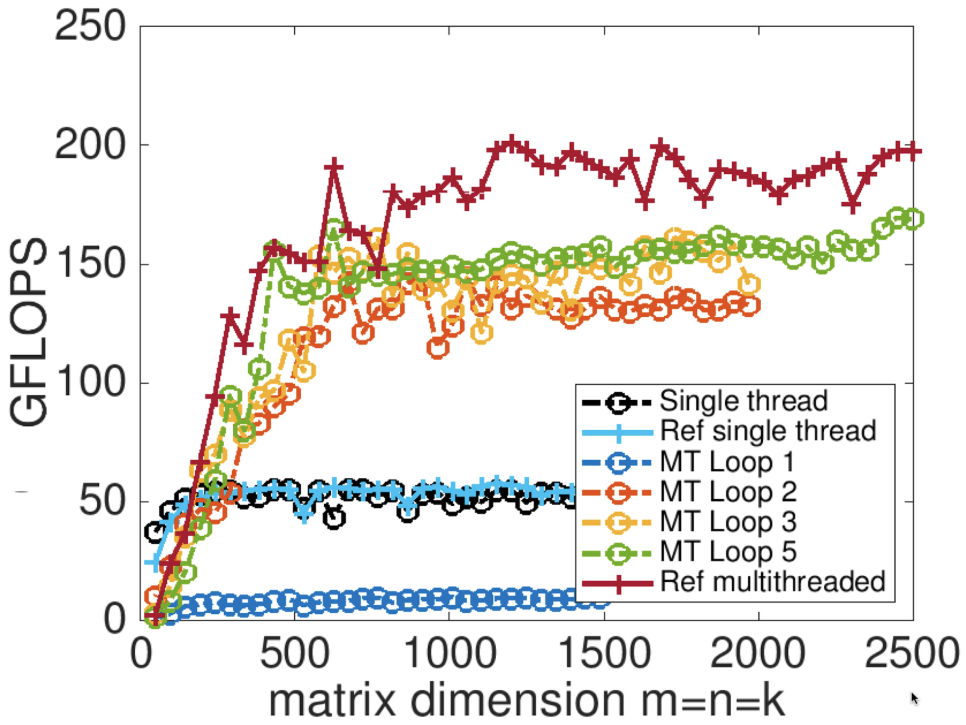
4.3.7 Discussion

If we also show the performance on Robert's laptop of the reference implementation when using multiple threads (4 in this graph) we get



The difference in performance between the multithreaded version of the reference implementation and the performance of, for example, the version of our implementation that parallelizes the fifth loop around the micro-kernel may be due to the fact that for our implementation runs "hotter" in the sense that it exercises the cores to the point where the architecture "clocks down" (reduces the clock speed to reduce power consumption). It will be interesting to see what happens on your computer. It requires further investigation that goes beyond the scope of this course.

For now, let's be satisfied with the excellent performance improvements we have achieved. Here is a graph that shows GFLOPS along the y-axis instead of GFLOPS/thread. Remember: we started by only getting a few GFLOPS back in Week 1, so we have improved performance by two orders of magnitude!



You can generate a similar graph for your own implementations with the Live Script in Plot_MT_Aggregate_GFLOPS_8x6.mlx.

4.4 Parallelizing More

4.4.1 Speedup, efficiency, etc.

4.4.1.1 Sequential time

We denote the sequential time for an operation by

$$T(n),$$

where n is a parameter that captures the size of the problem being solved. For example, if we only time problems with square matrices, then n would be the size of the matrices. $T(n)$ is equal the time required to compute the operation on a single core. If we consider all three sizes of the matrices, then n might be a vector with three entries.

A question is "Which sequential time?" Should we use the time by a specific implementation? Should we use the time for the fastest implementation? That's a good question! Usually it is obvious which to use from context.

4.4.1.2 Parallel time

The parallel time for an operation and a given implementation using t threads is given by

$$T_t(n).$$

This allows us to define the sequential time for a specific implementation by $T_1(n)$. Itis the sequential time of the parallel implementation when using only one thread.

4.4.1.3 Speedup

In the launch we already encountered the concept of speedup. There, we used it more informally.

The speedup for an operation and a given implementation using t threads is given by

$$S_t(n) = T(n)/T_t(n).$$

It measures how much speedup is attained by using t threads.

Often, the ratio

$$T_1(n)/T_t(n)$$

is used to report speedup (as we did in Unit 4.1.1). We now understand that this may give an optimistic result: If the sequential execution of the given implementation yields poor performance, it is still possible to attain very good speedup. The fair thing is to use $T(n)$ of the best implementation when computing speedup.

Remark 4.4.1 Intuitively, it would seem that

$$S_t(n) = T(n)/T_t(n) \leq t.$$

when you have at least t cores. In other words, that speedup is less than or equal to the number of threads/cores at your disposal. After all, it would seem that $T(n) \leq tT_t(n)$ since one could create a sequential implementation from the parallel implementation by mapping all computations performed by the threads to one thread, and this would remove the overhead of coordinating between the threads.

Let's use a simple example to show that speedup can be greater than t: Generally speaking, when you make a bed with two people, it takes less than half the time than when you do it by yourself. Why? Because usually two people have two sets of hands and the "workers" can therefore be on both sides of the bed simultaneously. Overhead incurrred when you make the bed by yourself (e.g., walking from one side to the other) is avoided. The same is true when using t threads on t cores: The task to be completed now has t sets of registers, t L1 caches, etc.

Attaining better than a factor t speedup with t threads and cores is called super-linear speedup. It should be viewed as the exception rather than the rule.

4.4.1.4 Efficiency

The efficiency for an operation and a given implementation using t threads is given by

$$E_t(n) = S_t(n)/t.$$

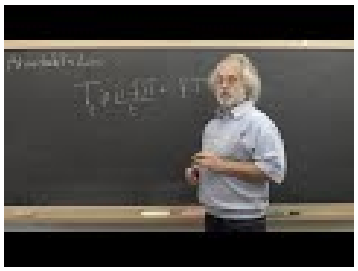
It measures how efficiently the t threads are being used.

4.4.1.5 GFLOPS/thread

We like to report GFLOPS (billions of floating point operations per second). Notice that this gives us an easy way of judging efficiency: If we know what the theoretical peak of a core is, in terms of GFLOPS, then we can easily judge how efficiently we are using the processor relative to the theoretical peak. How do we compute the theoretical peak? We can look up the number of cycles a core can perform per second and the number of floating point operations it can perform per cycle. This can then be converted to a peak rate of computation for a single core (in billions of floating point operations per second). If there are multiple cores, we just multiply this peak by the number of cores.

Remark 4.4.2 For current processors, it is almost always the case that as more threads are used, the frequency (cycles/second) of the cores is "clocked down" to keep the processor from overheating. This, obviously, complicates matters...

4.4.2 Ahmdahl's law



YouTube: https://www.youtube.com/watch?v=pT_f4ngiAl0

Ahmdahl's law is a simple observation about the limits on parallel speedup and efficiency. Consider an operation that requires sequential time

$$T$$

for computation. What if fraction f of this time is not parallelized (or cannot be parallelized), and the other fraction $(1 - f)$ is parallelized with t threads. We can then write

$$T = \underbrace{(1-f)T}_{\text{part that is parallelized}} + \underbrace{fT}_{\text{part that is not parallelized}}$$

and, assuming super-linear speedup is not expected, the total execution time with t threads, T_t, obeys the inequality

$$T_t \geq \frac{(1-f)T}{t} + fT \geq fT.$$

This, in turn, means that the speedup is bounded by

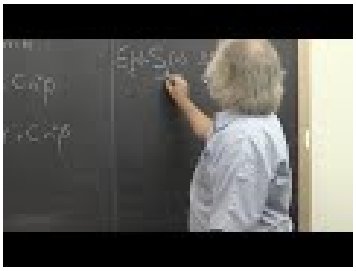
$$S_t = \frac{T}{T_t} \leq \frac{T}{fT} = \frac{1}{f}.$$

What we notice is that the maximal speedup is bounded by the inverse of the fraction of the execution time that is not parallelized. If, for example, 1/5 the

code is not optimized, then no matter how many threads you use, you can at best compute the result $1/(1/5) = 5$ times faster. This means that one must think about parallelizing all parts of a code.

Remark 4.4.3 Ahmdahl's law says that if one does not parallelize a part of the sequential code in which fraction f time is spent, then the speedup attained regardless of how many threads are used is bounded by $1/f$.

The point is: so far, we have focused on parallelizing the computational part of matrix-matrix multiplication. We should also parallelize the packing, since it requires a nontrivial part of the total execution time.



YouTube: https://www.youtube.com/watch?v=EKnJEoGEo6g

For parallelizing matrix-matrix multiplication, there is some hope. In our situation, the execution time is a function of the problem size, which we can model by the computation time plus overhead. A high-performance implementation may have a cost function that looks something like

$$T(n) = 2n^3\gamma + Cn^2\beta,$$

where γ is the time required for executing a floating point operation, β is some constant related to the time required to move a floating point number, and C is some other constant. (Notice that we have not done a thorough analysis of the cost of the final implementation in Week 3, so this formula is for illustrative purposes.) If now the computation is parallelized perfectly among the threads and, in a pessimistic situation, the time related to moving data is not at all, then we get that

$$T_t(n) = \frac{2n^3\gamma}{t} + Cn^2\beta.$$

so that the speedup is given by

$$S_t(n) = \frac{2n^3\gamma + Cn^2\beta}{\frac{2n^3\gamma}{t} + Cn^2\beta}$$

and the efficiency is given by

$$E_t(n) = \frac{S_t(n)}{t} = \frac{2n^3\gamma + Cn^2\beta}{2n^3\gamma + tCn^2\beta} = \frac{2n^3\gamma}{2n^3\gamma + tCn^2\beta} + \frac{Cn^2\beta}{2n^3\gamma + tCn^2\beta}$$

$$= \frac{1}{1 + \frac{tCn^2\beta}{2n^3\gamma}} + \frac{1}{\frac{2n^3\gamma}{Cn^2\beta} + t} = \frac{1}{1 + \left(\frac{tC\beta}{2\gamma}\right)\frac{1}{n}} + \frac{1}{\left(\frac{2\gamma}{C\beta}\right)n + t}.$$

Now, if t is also fixed, then the expressions in parentheses are constants and, as n gets large, the first term converges to 1 while the second term converges to 0. So, as the problem size gets large, the efficiency starts approaching 100%.

What this means is that whether all or part of the overhead can also be parallelized affects how fast we start seeing high efficiency rather than whether we eventually attain high efficiency for a large enough problem. Of course, there is a limit to how large of a problem we can store and/or how large of a problem we actually want to compute.

4.4.3 Parallelizing the packing

Since not parallelizing part of the computation can translate into a slow ramping up of performance, it is worthwhile to consider what else perhaps needs to be parallelized. Packing can contribute significantly to the overhead (although we have not analyzed this, nor measured it), and hence is worth a look. We are going to look at each loop in our "five loops around the micro-kernel" and reason through whether parallelizing the packing of the block of A and/or the packing of the row panel of B should be considered.

4.4.3.1 Loop two and parallelizing the packing

Let's start by considering the case where the second loop around the micro-kernel has been parallelized. Notice that all packing happens before this loop is reached. This means that, unless one explicitly parallelizes the packing of the block of A and/or the packing of the row panel of B, these components of the computation are performed by a single thread.

Homework 4.4.3.1

- Copy PackA.c into MT_PackA.c. Change this file so that packing is parallelized.
- Execute it with

  ```
  export OMP_NUM_THREADS=4
  make MT_Loop2_MT_PackA_8x6Kernel
  ```

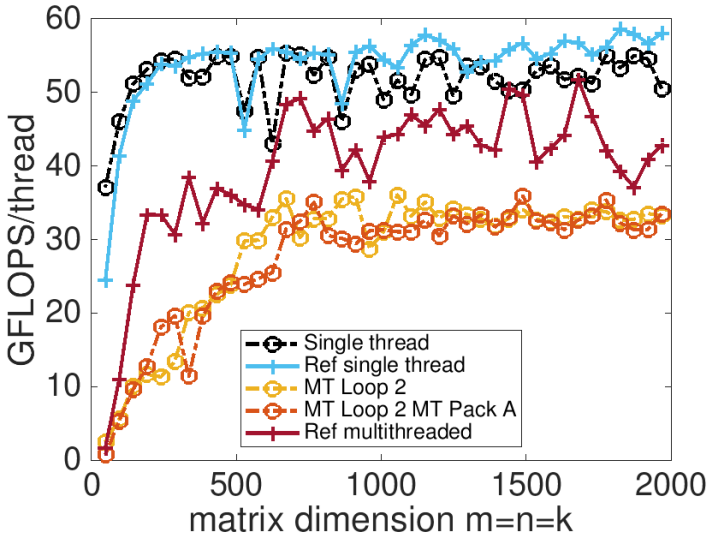
- Be sure to check if you got the right answer! Parallelizing the packing, the way PackA.c is written, is a bit tricky.
- View the resulting performance with data/Plot_MT_Loop2_MT_Pack_8x6.mlx.

Solution.

YouTube: https://www.youtube.com/watch?v=xAopdjhGXQA

- Assignments/Week4/Answers/MT_PackA.c

On Robert's laptop (using 4 threads), the performance is not noticeably changed:



Homework 4.4.3.2

- Copy PackB.c into MT_PackB.c. Change this file so that packing is parallelized.
- Execute it with

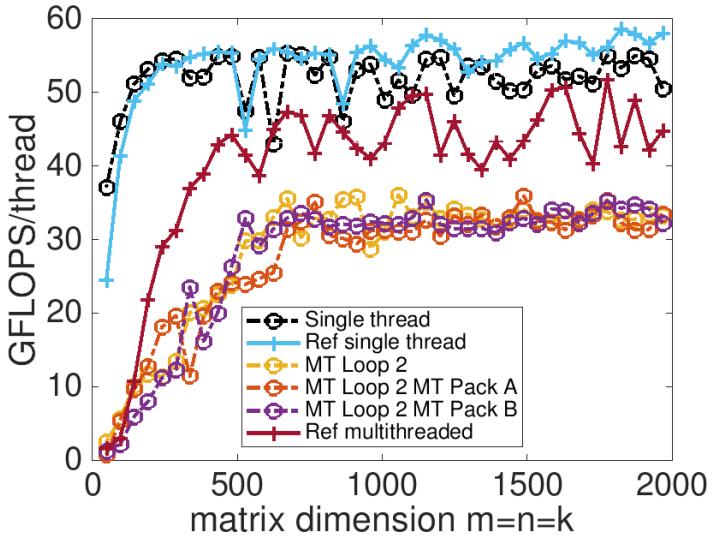
```
export OMP_NUM_THREADS=4
make MT_Loop2_MT_PackB_8x6Kernel
```

- Be sure to check if you got the right answer! Again, parallelizing the packing, the way PackB.c is written, is a bit tricky.
- View the resulting performance with data/Plot_MT_Loop2_MT_Pack_8x6.mlx.

Solution.

- Assignments/Week4/Answers/MT_PackB.c

On Robert's laptop (using 4 threads), the performance is again not noticeably changed:



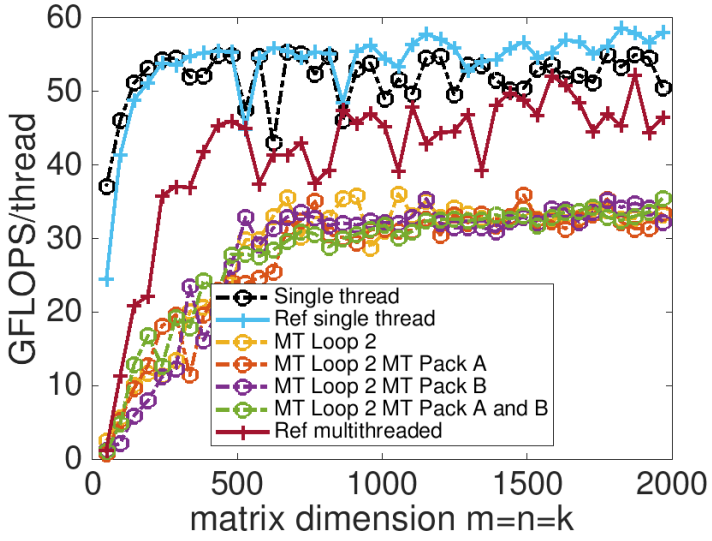
Homework 4.4.3.3 Now that you have parallelized both the packing of the block of A and the packing of the row panel of B, you are set to check if doing both shows a benefit.

- Execute

```
export OMP_NUM_THREADS=4
make MT_Loop2_MT_PackAB_8x6Kernel
```

- Be sure to check if you got the right answer!
- View the resulting performance with data/Plot_MT_Loop2_MT_Pack_8x6.mlx.

Solution. On Robert's laptop (using 4 threads), the performance is still not noticeably changed:



Why don't we see an improvement? Packing is a memory intensive task. Depending on how much bandwidth there is between cores and memory, pack-

ing with a single core (thread) may already saturate that bandwidth. In that case, parallelizing the operation so that multiple cores are employed does not actually speed the process. It appears that on Robert's laptop, the bandwidth is indeed saturated. Those with access to beefier processors with more bandwidth to memory may see some benefit from parallelizing the packing, especially when utilizing more cores.

4.4.3.2 Loop three and parallelizing the packing

Next, consider the case where the third loop around the micro-kernel has been parallelized. Now, each thread packs a different block of A and hence there is no point in parallelizing the packing of that block. The packing of the row panel of B happens before the third loop around the micro-kernel is reached, and hence one can consider parallelizing that packing.

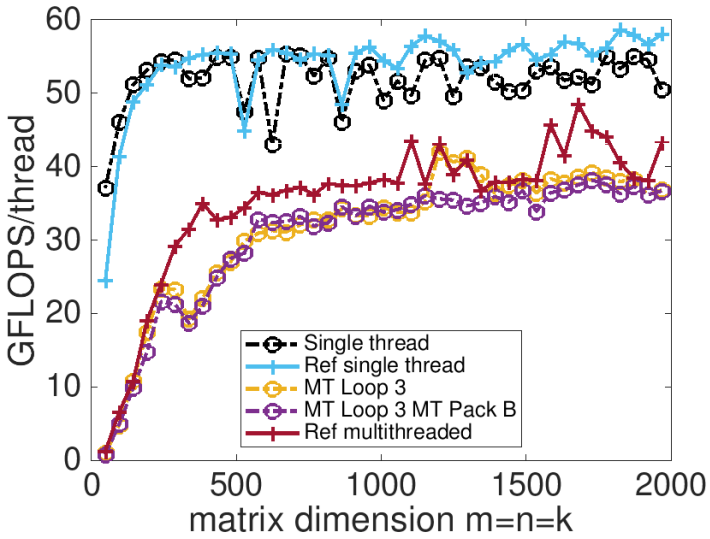
Homework 4.4.3.4 You already parallelized the packing of the row panel of B in Homework 4.4.3.2

- Execute

```
export OMP_NUM_THREADS=4
make MT_Loop3_MT_PackB_8x6Kernel
```

- Be sure to check if you got the right answer!
- View the resulting performance with data/Plot_MT_Loop3_MT_Pack_8x6.mlx.

Solution. On Robert's laptop (using 4 threads), the performance is again not noticeably changed:



4.4.3.3 Loop five and parallelizing the packing

Next, consider the case where the fifth loop around the micro-kernel has been parallelized. Now, each thread packs a different row panel of B and hence there

is no point in parallelizing the packing of that block. As each thread executes subsequent loops (loops four through one around the micro-kernel), they pack blocks of A redundantly, since each allocates its own space for the packed block. It should be possible to have them collaborate on packing a block, but that would require considerable synchronization between loops... Details are beyond this course. If you know a bit about OpenMP, you may want to try this idea.

4.4.4 Parallelizing multiple loops

In Section 4.3, we only considered parallelizing one loop at a time. When one has a processor with many cores, and hence has to use many threads, it may become beneficial to parallelize multiple loops. The reason is that there is only so much parallelism to be had in any one of the m, n, or k sizes. At some point, computing with matrices that are small in some dimension starts affecting the ability to amortize the cost of moving data.

OpenMP allows for "nested parallelism." Thus, one possibility would be to read up on how OpenMP allows one to control how many threads to use in a parallel section, and to then use that to achieve parallelism from multiple loops. Another possibility is to create a parallel section (with #pragma omp parallel rather than #pragma omp parallel for) when one first enters the "five loops around the micro-kernel," and to then explicitly control which thread does what work at appropriate points in the nested loops.

In [25]

- Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee, Anatomy of High-Performance Many-Threaded Matrix Multiplication, in the proceedings of the 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014), 2014.

the parallelization of various loops is discussed, as is the need to parallelize muiltiple loops when targeting "many-core" architectures like the Intel Xeon Phi (KNC) and IBM PowerPC A2 processors. The Intel Xeon Phi has 60 cores, each of which has to execute four "hyperthreads" in order to achieve the best performance. Thus, the implementation has to exploit 240 threads...

4.5 Enrichments

4.5.1 Casting computation in terms of matrix-matrix multiplication

A key characteristic of matrix-matrix multiplication that allows it to achieve high performance is that it performs $O(n^3)$ computations with $O(n^2)$ data (if $m = n = k$). This means that if n is large, it may be possible to amortize the cost of moving a data item between memory and faster memory over $O(n)$ computations.

There are many computations that also have this property. Some of these are minor variations on matrix-matrix multiplications, like those that are part

of the level-3 BLAS mentioned in Unit 1.5.1. Others are higher level operations with matrices, like the LU factorization of an $n \times n$ matrix we discuss later in this unit. For each of these, one could in principle apply techniques you experienced in this course. But you can imagine that this could get really tedious.

In the 1980s, it was observed that many such operations can be formulated so that most computations are cast in terms of a matrix-matrix multiplication [7] [15] [1]. This meant that software libraries with broad functionality could be created without the burden of going through all the steps to optimize each routine at a low level.

Let us illustrate how this works for the LU factorization, which, given an $n \times n$ matrix A computes a unit lower triangular L and upper triangular matrix U such that $A = LU$. If we partition

$$A \rightarrow \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, L \rightarrow \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix}, \text{ and } U \rightarrow \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix},$$

where A_{11}, L_{11}, and U_{11} are $b \times b$, then $A = LU$ means that

$$\begin{array}{rcl} \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} & = & \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} \\ & = & \begin{pmatrix} L_{11}U_{11} & L_{11}U_{11} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{pmatrix}. \end{array}$$

If one manipulates this, one finds that

$$\begin{array}{rcl} A_{11} & = & L_{11}U_{11} \\ A_{12} & = & L_{11}U_{12} \\ A_{21} & = & L_{21}U_{11} \\ A_{22} - L_{21}U_{12} & = & L_{22}U_{22}. \end{array}$$

which then leads to the steps

- Compute the LU factoriation of the smaller matrix $A_{11} \rightarrow L_{11}U_{11}$.

  Upon completion, we know unit lower triangular L_{11} and upper triangular matrix U_{11}.

- Solve $L_{11}U_{12} \rightarrow A_{12}$ for U_{12}. This is known as a "triangular solve with multiple right-hand sides," which is an operation supported by the level-3 BLAS.

  Upon completion, we know U_{12}.

- Solve $L_{21}U_{11} \rightarrow A_{21}$ for L_{12}. This is a different case of "triangular solve with multiple right-hand sides," also supported by the level-3 BLAS.

  Upon completion, we know L_{21}.

- Update $A_{22} := A_{22} - L_{21}U_{12}$.

- Since before this update $A_{22} - L_{21}U_{12} = L_{22}U_{22}$, this process can now continue with $A = A_{22}$.

These steps, are presented as an algorithm in Figure 4.5.1, using the "FLAME" notation developed as part of our research and used in our other two MOOCs (Linear Algebra: Foundations to Frontiers [21] and LAFF-On Programming for Correctness [20]). This leaves matrix A overwritten with the unit lower triangular matrix L below the diagonal (with the ones on the diagonal implicit) and U on and above the diagonal.

The important insight is that if A is $n \times n$, then factoring A_{11} requires approximately $\frac{2}{3}b^3$ flops, computing U_{12} and L_{21} each require approximately $b^2(n-b)$ flops, and updating $A_{22} := A_{22} - L_{21}U_{12}$ requires approximately $2b(n-b)^2$ flops. If $b \ll n$ then most computation is in the update $A_{22} := A_{22} - L_{21}U_{12}$, which we recognize as a matrix-matrix multiplication (except that the result of the matrix-matrix multiplication is subtracted rather than added to the matrix). Thus, most of the computation is cast in terms of a matrix-matrix multiplication. If it is fast, then the overall computation is fast once n is large.

If you don't know what an LU factorization is, you will want to have a look at Week 6 of our MOOC titled "Linear Algebra: Foundations to Frontiers [21]. There, it is shown that (Gaussian) elimination, which allows one to solve systems of linear equations, is equivalent to LU factorization.

$A = \text{LU\_blk\_var5}(A)$

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$$

A_{TL} is 0×0

while $n(A_{BR}) > 0$

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|cc} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{array}\right)$$

$A_{11} \rightarrow L_{11}U_{11}$, overwriting A_{11} with L_{11} and U_{11}

Solve $L_{11}U_{12} = A_{12}$, overwriting A_{12} with U_{12}

Solve $L_{21}U_{11} = A_{21}$, overwriting A_{21} with L_{21}

$A_{22} := A_{22} - L_{21}U_{21}$ (matrix-matrix multiplication)

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{cc|c} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

Figure 4.5.1: Blocked LU factorization.

Algorithms like the one illustrated here for LU factorization are referred to as blocked algorithms. Unblocked algorithms are blocked algorithms for which the block size has been chosen to equal 1. Typically, for the smaller subproblem (the factorization of A_{11} in our example), an unblocked algorithm is used. A large number of unblocked and blocked algorithms for different linear algebra operations are discussed in the notes that we are preparing for our graduate level course "Advanced Linear Algebra: Foundations to Frontiers," [19], which

will be offered as part of the UT-Austin Online Masters in Computer Science. We intend to offer it as an auditable course on edX as well.

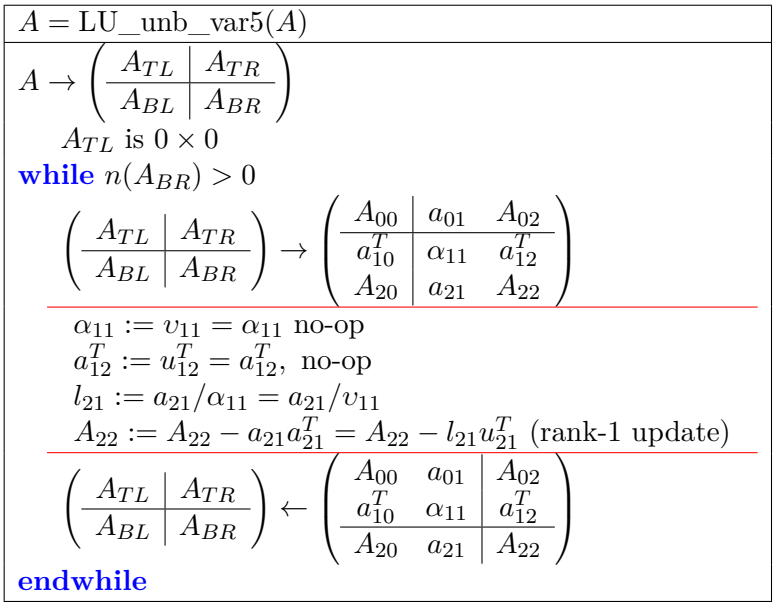


Figure 4.5.2: Unblocked LU factorization.

A question you may ask yourself is "What block size, b, should be used in the algorithm (when n is large)?"

4.5.2 Family values

For most dense linear algebra operations, like LU factorization, there are multiple unblocked and corresponding blocked algorithms. Usually, each of the blocked algorithms casts most computation in terms of an operation that is a matrix-matrix multiplication, or a variant thereof. For example, there are five blocked algorithms (variants) for computing the LU factorization, captured in Figure 4.5.3.

$A = \text{LU\_blk}(A)$

$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$

A_{TL} is 0×0

while $n(A_{BR}) > 0$

$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|cc} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{array}\right)$

Variant 1:	Variant 2:	Variant 3:
$A_{01} := L_{00}^{-1} A_{01}$	$A_{10} := A_{10} U_{00}^{-1}$	$A_{10} := A_{10} U_{00}^{-1}$
$A_{10} := A_{10} U_{00}^{-1}$	$A_{11} := A_{11} - A_{10} A_{01}$	$A_{11} := A_{11} - A_{10} A_{01}$
$A_{11} := A_{11} - A_{10} A_{01}$	$A_{11} := LU(A_{11})$	$A_{11} := LU(A_{11})$
$A_{11} := LU(A_{11})$	$A_{12} := A_{12} - A_{10} A_{02}$	$A_{21} := A_{21} - A_{20} A_{01}$
	$A_{12} := L_{11}^{-1} A_{12}$	$A_{21} := A_{21} U_{11}^{-1}$

Variant 4:	Variant 5:
$A_{11} := A_{11} - A_{10} A_{01}$	$A_{11} := LU(A_{11})$
$A_{12} := A_{12} - A_{10} A_{02}$	$A_{12} := L_{11}^{-1} A_{12}$
$A_{12} := L_{11}^{-1} A_{12}$	$A_{21} := A_{21} U_{11}^{-1}$
$A_{21} := A_{21} - A_{20} A_{01}$	$A_{22} := A_{22} - A_{21} A_{12}$
$A_{21} := A_{21} U_{11}^{-1}$	

$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{cc|c} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$

endwhile

Figure 4.5.3: Five blocked LU factorizations. L_{ii} and U_{ii} denote the unit lower triangular matrix and upper triangular matrix stored in A_{ii}, respectively. $L_{ii}^{-1} A_{ij}$ is shorthand for solving the triangular system with right-hand sides $L_{ii} U_{ij} = A_{ij}$ and $A_{ij} U_{jj}^{-1}$ is shorthand for solving the triangular system with right-hand sides $L_{ij} U_{jj} = A_{ij}$.

It turns out that different variants are best used under different circumstances.

- Variant 5, also known as Classical Gaussian Elimination and the algorithm in Figure 4.5.1, fits well with the matrix-matrix multiplication that we developed in this course, because it casts most computation in terms of a matrix-multiplication with large matrix sizes m and n, and small matrix size k (which equals b): $A_{22} := A_{22} - A_{21} A_{12}$.
- Variant 3, also known as the left-looking variant, can be easily checkpointed (a technique that allows the computation to be restarted if, for example, a computer crashes partially into a long calculation).
- Variants 3, 4, and 5 can all be modified to incorporate "partial pivoting" which improves numerical stability (a topic covered in courses on numerical linear algebra).

Further details go beyond the scope of this course. But we will soon have a MOOC for that: Advanced Linear Algebra: Foundations to Frontiers! [19].

What we are trying to give you a flavor of is the fact that it pays to have a family of algorithms at one's disposal so that the best algorithm for a situation can be chosen. The question then becomes "How do we find all algorithms in the family?" We have a MOOC for that too: LAFF-On Programming for Correctness [20].

4.5.3 Matrix-matrix multiplication for Machine Learning

Matrix-matrix multiplication turns out to be an operation that is frequently employed by algorithms in machine learning. In this unit, we discuss the all-nearest-neighbor problem. Knowing how to optimize matrix-matrix multiplication allows one to optimize a practical algorithm for computing it.

The k-nearest-neighbor problem (KNN) takes as input m points in $\mathbb{R}^n$, $\{x_j\}_{j=0}^{m-1}$, and a reference point, x, and computes the k nearest neighbors of x among the m points. The all-nearest-neighbor (ANN) problem computes the k nearest neighbors of each of the points x_j.

The trick to computing ANN is to observe that we need to compute the distances between all points x_i and x_j, given by $\|x_i - x_j\|_2$. But,

$$\|x_i - x_j\|_2^2 = (x_i - x_j)^T(x_i - x_j) = x_i^T x_i - 2x_i^T x_j + x_j^T x_j.$$

So, if one creates the matrix

$$X = \left(\begin{array}{c|c|c|c} x_0 & x_1 & \cdots & x_{m-1} \end{array} \right)$$

and computes

$$C = X^T X = \left(\begin{array}{c|c|c|c} x_0^T x_0 & \star & \cdots & \star \\ \hline x_1^T x_0 & x_1^T x_1 & \cdots & \star \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline x_{m-1}^T x_0 & x_{m-1}^T x_1 & \cdots & x_{m-1}^T x_{m-1} \end{array} \right).$$

Hence, if the lower triangular part of C is computed, then

$$\|x_i - x_j\|_2^2 = \gamma_{i,i} - 2\gamma_{i,j} + \gamma_{j,j}.$$

By sorting this information, the nearest neighbors for each x_i can be found.

There are three problems with this:

- Only the lower triangular part of C needs to be computed. This operation is known as a symmetric rank-k update. (Here the k refers to the number of rows in X rather than the k in k-nearest-neighbors.) How Goto's algorithm can be modified to compute the symmetric rank-k update is discussed in [11].

- Typically $m \gg n$ and hence the intermediate matrix C takes a very large amount of space to store an intermediate result.

- Sorting the resulting information in order to determine the k nearest neighbors for each point means reading and writing data multipliple times from and to memory.

You can read up on how to achieve a high performance implementation for solving the all-nearest-neighbor problem by exploiting what we know about implementing matrix-matrix multiplication in [34]:

- Chenhan D. Yu, Jianyu Huang, Woody Austin, Bo Xiao, George Biros. Performance Optimization for the K-Nearest Neighbors Kernel on x86 Architectures, proceedings of SC'15, 2015.

4.5.4 Matrix-matrix multiplication on GPUs

Matrix-matrix multiplications are often offloaded to GPU accellerators due to their cost-effective performance achieved with low power consumption. While we don't target GPUs with the implementations explored in this course, the basic principles that you now know how to apply when programming CPUs also apply to the implementation of matrix-matrix multiplication on GPUs.

The recent CUDA Templates for Linear Algebra Subroutines (CUTLASS) [16] from NVIDIA expose how to achieve high-performance matrix-matrix multiplication on NVIDIA GPUs, coded in C++.

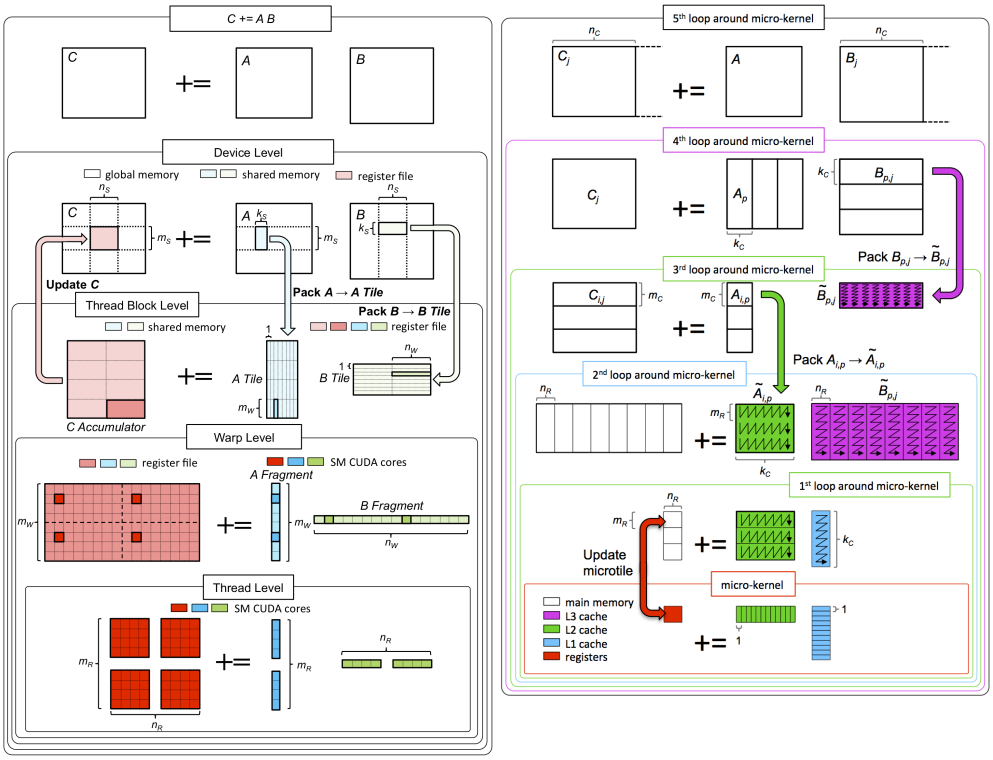


Figure 4.5.4: Left: picture that captures the nested computation as implemented in CUTLASS. Right: the picture that captures the BLIS "Five Loops around the micro-kernel, with packing."

The explanation in [16] is captured in Figure 4.5.4, taken from [13]

- Jianyu Huang, Chenhan D. Yu, and Robert A. van de Geijn, Strassen's Algorithm Reloaded on GPUs, ACM Transactions on Mathematics Software, in review.

where it is compared to the by now familiar picture of the five loops around the micro-kernel. The main focus of the paper is the practical implementation on GPUs of Strassen's algorithm from [14], discussed in Unit 3.5.4.

4.5.5 Matrix-matrix multiplication on distributed memory architectures

We had intended to also cover the implementation of matrix-matrix multiplication on distributed memory architectures. We struggled with how to give learners access to a distributed memory computer and for this reason decided to not yet tackle this topic. Here we point you to some papers that will fill the void for now. These papers were written to target a broad audience, much like the materials in this course. Thus, you should already be well-equipped to study the distributed memory implementation of matrix-matrix multiplication on your own.

Think of distributed memory architectures as a collection of processors, each with their own cores, caches, and main memory, that can collaboratively solve problems by sharing data via the explicit sending and receiving of messages via a communication network. Such computers used to be called "multi-computers" to capture that each "node" in the system has its own processor and memory hierarchy.

Practical implementations of matrix-matrix multiplication on multi-computers are variations on the Scalable Universal Matrix-Multiplication Algorithm (SUMMA) [29]. While that paper should be easily understood upon completing this course, a systematic treatment of the subject that yields a large family of algorithms is given in [22]:

- Martin D. Schatz, Robert A. van de Geijn, and Jack Poulson, Parallel Matrix Multiplication: A Systematic Journey, SIAM Journal on Scientific Computing, Volume 38, Issue 6, 2016.

which should also be easily understood by learners in this course.

Much like it is important to understand how data moves between the layers in the memory hierarchy of a single processor, it is important to understand how to share data between the memories of a multi-computer. Although the above mentioned papers give a brief overview of the data movements that are encounted when implementing matrix-matrix multiplication, orchestrated as "collective communication", it helps to look at this topic in-depth. For this we recommend the paper [4]

- Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn, Collective communication: theory, practice, and experience, Concurrency and Computation: Practice and Experience, Volume 19, Number 13, 2007.

In the future, we may turn these materials into yet another Massive Open Online Course. Until then, enjoy the reading.

4.5.6 High-Performance Computing Beyond Matrix-Matrix Multiplication

In this course, we took one example, and used that example to illustrate various issues encountered when trying to achieve high performance. The course is an inch wide and a mile deep!

For materials that treat the subject of high performance computing more broadly, you may find [8] interesting:

- Victor Eijkhout, Introduction to High-Performance Scientific Computing.

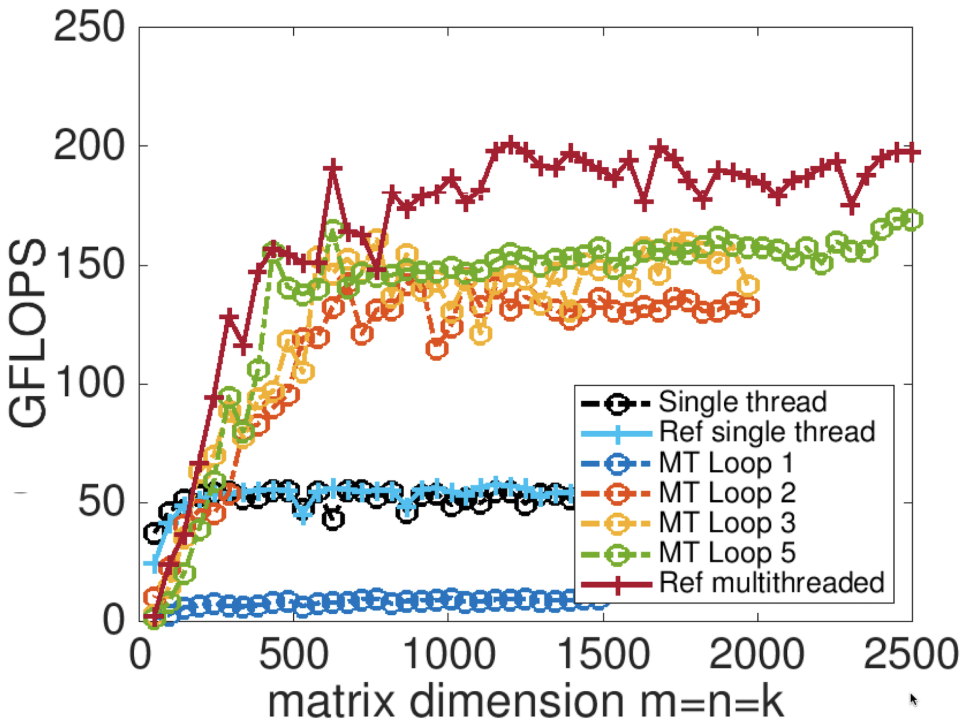
4.6 Wrap Up

4.6.1 Additional exercises

You may want to dive deeper into our BLAS-like Library Instantiation Software (BLIS) [3], discussed in Unit 1.5.2, which instantiates many of the fundamental insights you have encountered in this course in a high quality, high performing library. That library implements all of the BLAS functionality discussed in Unit 1.5.1, and more.

4.6.2 Summary

4.6.2.1 The week in pictures



4.6.2.2 OpenMP basics

OpenMP is a standardized API (Application Programming Interface) for creating multiple threads of execution from a single program. For our purposes, you need to know very little:

- There is a header file to include in the file(s):

  ```
  #include <omp.h>
  ```

- Directives to the compiler (pragmas):

  ```
  #pragma omp parallel
  #pragma omp parallel for
  ```

- A library of routines that can, for example, be used to inquire about the execution environment:

  ```
  omp_get_max_threads()
  omp_get_num_threads()
  omp_get_thread_num()
  ```

- Environment parameters that can be set before executing the program:

  ```
  export OMP_NUM_THREADS=4
  ```

  If you want to see what the value of this environment parameter is, execute

  ```
  echo $OMP_NUM_THREADS
  ```

Appendix A

Appendix B

GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE. The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS. This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute

the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the

Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING. You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY. If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque

copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS. You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties — for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS. You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all

of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS. You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS. A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION. Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in

the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION. You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE. The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING. "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A

public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents. To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with... Texts." line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

References

[1] Ed Anderson, Zhaojun Bai, James Demmel, Jack J. Dongarra, Jeremy DuCroz, Ann Greenbaum, Sven Hammarling, Alan E. McKenney, Susan Ostrouchov, and Danny Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, 1992.

[2] Jeff Bilmes, Krste Asanovc, Chee-whye Chin, Jim Demmel, *Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology*, International Conference on Supercomputing, July 1997.

[3] *BLAS-like Library Instantiation Software Framework*, GitHub repository.

[4] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn, *Collective communication: theory, practice, and experience*, Concurrency and Computation: Practice and Experience, Volume 19, Number 13, 2007. If you don't have access, you may want to read the advanced draft Ernie Chan, Marcel Heimlich, Avijit Purkayastha, and Robert van de Geijn. Collective Communication: Theory, Practice, and Experience, FLAME Working Note #22. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-06-44. September 26, 2006.

[5] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff, *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol. 16, No. 1, pp. 1-17, March 1990.

[6] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson, *An Extended Set of {FORTRAN} Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, Vol. 14, No. 1, pp. 1-17, March 1988.

[7] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM, Philadelphia, PA, 1991.

[8] Victor Eijkhout, *Introduction to High-Performance Scientific Computing,* lulu.com. http://pages.tacc.utexas.edu/~eijkhout/istc/istc.html

[9] Kazushige Goto and Robert van de Geijn, *On reducing TLB misses in matrix multiplication*, Technical Report TR02-55, Department of Computer Sciences, UT-Austin, Nov. 2002.

[10] Kazushige Goto and Robert van de Geijn, *Anatomy of High-Performance Matrix Multiplication*, ACM Transactions on Mathematical Software, Vol. 34, No. 3: Article 12, May 2008.

[11] Kazushige Goto and Robert van de Geijn, *High-performance implementation of the level-3 BLAS*, ACM Transactions on Mathematical Software, Vol. 35, No. 1: Article 4, July 2008.

[12] Jianyu Huang, Leslie Rice, Devin A. Matthews, Robert A. van de Geijn, *Generating Families of Practical Fast Matrix Multiplication Algorithms*, in Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS17), Orlando, FL, May 29-June 2, 2017.

[13] Jianyu Huang, Chenhan D. Yu, and Robert A. van de Geijn, *Strassen's Algorithm Reloaded on GPUs*, ACM Transactions on Mathematics Software, in review.

[14] Jianyu Huang, Tyler Smith, Greg Henry, and Robert van de Geijn, *Strassen's Algorithm Reloaded*, International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16), 2016.

[15] Bo Kågström, Per Ling, and Charles Van Loan, *GEMM-based Level 3 BLAS: High Performance Model Implementations and Performance Evaluation Benchmark*, ACM Transactions on Mathematical Software, Vol. 24, No. 3: pp. 268-302, 1998.

[16] Andrew Kerr, Duane Merrill, Julien Demouth and John Tran, *CUDA Templates for Linear Algebra Subroutines (CUTLASS)*, NVIDIA Developer Blog, 2017. https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/.

[17] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, *Basic Linear Algebra Subprograms for Fortran Usage*, ACM Transactions on Mathematical Software, Vol. 5, No. 3, pp. 308-323, Sept. 1979.

[18] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti, *Analytical Modeling Is Enough for High-Performance {BLIS}*, ACM Journal on Mathematical Software, Vol. 43, No. 2, Aug. 2016. PDF of draft

[19] Margaret E. Myers and Robert A. van de Geijn, *Advanced Linear Algebra: Foundations to Frontiers*, ulaff.net, in preparation.

[20] Margaret E. Myers and Robert A. van de Geijn, *LAFF-On Programming for Correctness*, ulaff.net, 2017.

[21] Margaret E. Myers and Robert A. van de Geijn, *Linear Algebra: Foundations to Frontiers - Notes to LAFF With*, ulaff.net, 2014.

[22] Martin D. Schatz, Robert A. van de Geijn, and Jack Poulson, *Parallel Matrix Multiplication: A Systematic Journey*, SIAM Journal on Scientific Computing, Volume 38, Issue 6, 2016.

[23] Tyler Michael Smith, Bradley Lowery, Julien Langou, Robert A. van de Geijn, *A Tight I/O Lower Bound for Matrix Multiplication*, arxiv.org:1702.02017v2, 2019. (Submitted to ACM Transactions on Mathematical Software.)

[24] Tyler Smith and Robert van de Geijn, *The MOMMS Family of Matrix*

Multiplication Algorithms, arxiv.org:1904.05717 , 2019.

[25] Tyler M. Smith, Robert A. van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee, *Anatomy of High-Performance Many-Threaded Matrix Multiplication*, proceedings of the 28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014), 2014.

[26] Volker Strassen, *Gaussian Elimination is not Optimal*, Numer. Math. 13, p. 354-356, 1969

[27] Field G. Van Zee and Tyler M. Smith, *Implementing High-performance Complex Matrix Multiplication via the 3M and 4M Methods*, ACM Transactions on Mathematical Software, Vol. 44, No. 1, pp. 7:1-7:36, July 2017.

[28] Robert van de Geijn and Kazushige Goto, *BLAS (Basic Linear Algebra Subprograms)*, Encyclopedia of Parallel Computing, Part 2, pp. 157-164, 2011. If you don't have access, you may want to read an advanced draft.

[29] Robert van de Geijn and Jerrell Watts, *SUMMA: Scalable Universal Matrix Multiplication Algorithm*, Concurrency: Practice and Experience, Volume 9, Number 4, 1997.

[30] Field G. Van Zee, *Implementing High-Performance Complex Matrix Multiplication via the 1m Method*, ACM Journal on Mathematical Software, in review.

[31] Field G. Van Zee, Tyler Smith, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John Gunnels, Tze Meng Low, Bryan Marker, Lee Killough, and Robert A. van de Geijn, *The BLIS Framework: Experiments in Portability*, ACM Journal on Mathematical Software, Vol. 42, No. 2, June 2016. You can access this article for free by visiting the Science of High-Performance Computing group webpage and clicking on the title of Journal Article 39.

[32] Field G. Van Zee and Robert A. van de Geijn, *BLIS: A Framework for Rapidly Instantiating BLAS Functionality*, ACM Journal on Mathematical Software, Vol. 41, No. 3, June 2015. You can access this article for free by visiting the Science of High-Performance Computing group webpage and clicking on the title of Journal Article 39.

[33] Richard C. Whaley, Antoine Petitet, and Jack J. Dongarra, *Automated Empirical Optimization of Software and the ATLAS Project*, Parallel Computing, 27 (1–2): 3–35, 2001.

[34] Chenhan D. Yu, Jianyu Huang, Woody Austin, Bo Xiao, George Biros, *Performance Optimization for the K-Nearest Neighbors Kernel on x86 Architectures*, proceedings of SC'15, 2015. [PDF]

Index

Colophon

This book was authored in MathBook XML.