Opportunities for Parallelism in Matrix Multiplication

FLAME Working Note #71

Tyler M. Smith*   Robert A. van de Geijn*   Mikhail Smelyanskiy†   Jeff Hammond‡
Field G. Van Zee*

October 23, 2013
Revised: January 10, 2014

Abstract

BLIS is a new framework for rapid instantiation of the BLAS. We describe how BLIS extends the "GotoBLAS approach" to implementing matrix multiplication (GEMM). While GEMM was previously implemented as three loops around an inner kernel, BLIS exposes two additional loops within that inner kernel, casting the computation in terms of the BLIS micro-kernel so that porting GEMM becomes a matter of customizing this micro-kernel for a given architecture. We discuss how this facilitates a finer level of parallelism that greatly simplifies the multithreading of GEMM as well as additional opportunities for parallelizing multiple loops. Specifically, we show that with the advent of many-core architectures such as the IBM PowerPC A2 processor (used by Blue Gene/Q) and the Intel Xeon Phi processor, parallelizing both within and around the inner kernel, as the BLIS approach supports, is not only convenient, but also necessary for scalability. The resulting implementations deliver what we believe to be the best open source performance for these architectures, achieving both impressive performance and excellent scalability.

1 Introduction

High-performance implementation of matrix-matrix multiplication (GEMM) is both of great practical importance, since many computations in scientific computing can be cast in terms of this operation, and of pedagogical importance, since it is often used to illustrate how to attain high performance on a novel architecture. A few of the many noteworthy papers from the past include Agarwal et al. [1] (an early paper that showed how an implementation in a high level language—Fortran—can attain high performance), Bilmer et al. [3] (which introduced auto-tuning and code generation using the C programming language), Whaley and Dongarra [14] (which productized the ideas behind PHiPAC), and Goto and van de Geijn [6] (which described what is currently accepted to be the most effective approach to implementation, which we will call the GotoBLAS approach).

Very recently, we introduced the BLAS-like Library Instantiation Software (BLIS) [13] which can be viewed as a systematic reimplementation of the GotoBLAS, but with a number of key insights that greatly reduce the effort for the library developer. The primary innovation is the insight that the inner kernel—the smallest unit of computation within the GotoBLAS GEMM implementation—can be further simplified into two loops around a micro-kernel. This means that the library developer needs only implement and optimize

---

*Department of Computer Science and Institute for Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX 78712. {tms,rvdg,field}@cs.utexas.edu.

†Parallel Computing Lab, Intel Corporation, Santa Clara, CA 95054. mikhail.smelyanskiy@intel.com.

‡Leadership Computing Facility, Argonne National Laboratory, Lemont, IL 60439. jhammond@anl.gov.

a routine[1] that implements the computation of $C := AB + C$ where C is a small submatrix that fits in the registers of a target architecture. In a second paper [12], we reported experiences regarding portability and performance on a large number of current processors. Most of that paper is dedicated to implementation and performance on a single core. A brief demonstration of how BLIS also supports parallelism was also included in that paper, but with few details.

The present paper describes in detail the opportunities for parallelism exposed by the BLIS implementation of GEMM. It focuses specifically on how this supports high performance and scalability when targeting many-core architectures that require more threads than cores if near-peak performance is to be attained. Two architectures are examined: the PowerPC A2 processor with 16 cores that underlies IBM's Blue Gene/Q supercomputer, which supports four-way hyperthreading for a total of 64 threads; and the Intel Xeon Phi processor with 60 cores[2] and also supports four-way hyperthreading for a total of 240 threads. It is shown that excellent performance and scalability can be achieved specifically because of the extra parallelism that is exposed by the BLIS approach within the inner kernel employed by the GotoBLAS approach.

It is also shown that when many threads are employed it is necessary to parallelize both in the m and the n dimensions. This builds upon Marker et al. [10], which we believe was the first paper to look at 2D work decomposition for GEMM on multithreaded architectures. The paper also builds upon work that describe the vendor implementations for the PowerPC A2 [7] and the Xeon Phi [9]. BLIS wraps many of those insights up in a cleaner framework so that exploration of the algorithmic design space is, in our experience, simplified. We show performance to be competitive relative to that of Intel's Math Kernel Library (MKL) and IBM's Engineering and Scientific Subroutine Library (ESSL)[3].

2 BLIS

In our discussions in this paper, we focus on the special case $C := AB + C$, where A, B, and C are $m \times k$, $k \times n$, and $m \times n$, respectively.[4] It helps to be familiar with the GotoBLAS approach to implementing GEMM, as described in [6]. We will briefly review the BLIS approach for a single core implementation in this section, with the aid of Figure 1.

Our description starts with the outer-most loop, indexed by j_c. This loop partitions C and B into (wide) column panels. Next, A and the current column panel of B are partitioned into column panels and row panels, respectively, so that the current column panel of C (of width n_c) is updated as a sequence of rank-k updates (with $k = k_c$), indexed by p_c. At this point, the GotoBLAS approach packs the current row panel of B into a contiguous buffer, $\widetilde{B}$. If there is an L3 cache, the computation is arranged to try to keep $\widetilde{B}$ in the L3 cache. The primary reason for the outer-most loop, indexed by j_c, is to limit the amount of workspace required for $\widetilde{B}$, with a secondary reason to allow $\widetilde{B}$ to remain in the L3 cache.

Now, the current panel of A is partitioned into blocks, indexed by i_c, that are packed into a contiguous buffer, $\widetilde{A}$. The block is sized to occupy a substantial part of the L2 cache, leaving enough space to ensure that other data does not evict the block. The GotoBLAS approach then implements the "block-panel" multiplication of $\widetilde{A}\widetilde{B}$ as its inner kernel, making this the basic unit of computation. It is here that the BLIS approach continues to mimic the GotoBLAS approach, except that it explicitly exposes two additional loops. In BLIS, these loops are coded portably in C, whereas in GotoBLAS they are hidden within the

---

[1] This micro-kernel routine is usually written in assembly code, but may also be expressed in C with vector intrinsics.

[2] In theory, 61 cores can be used for computation. In practice, 60 cores are usually employed.

[3] We do not compare to OpenBLAS [15] or ATLAS as neither has an implementation for either architecture, to our knowledge.

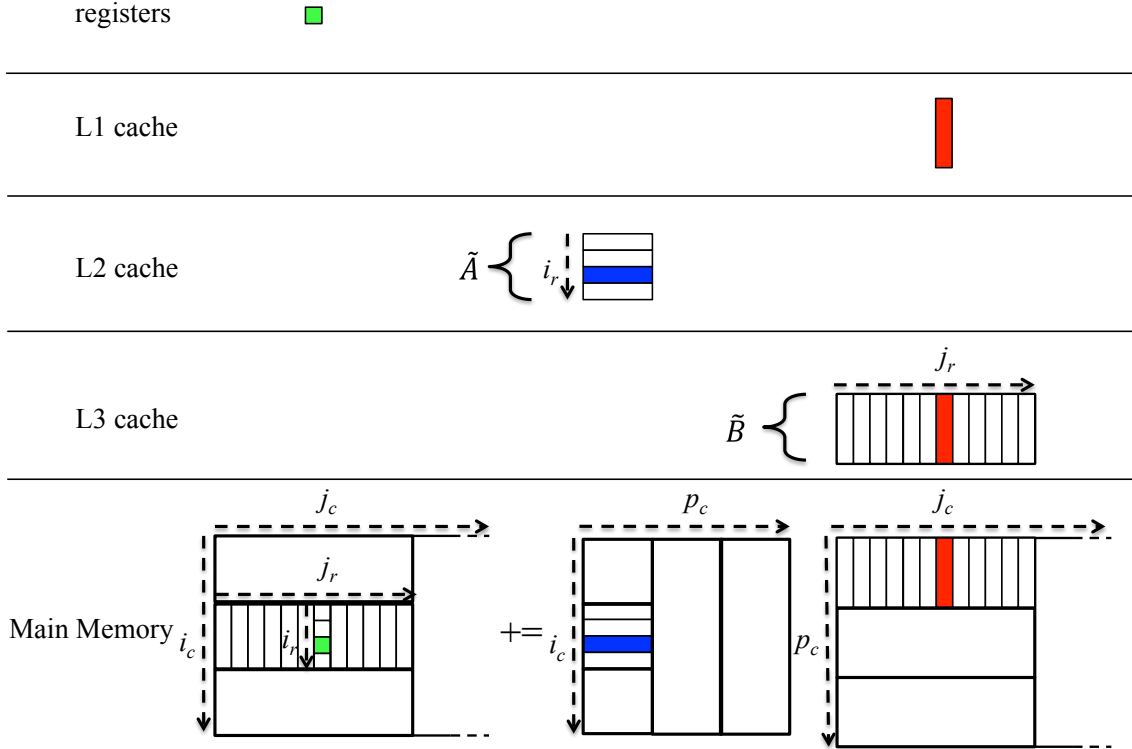[4] We will also write this operation as $C \mathrel{+}= AB$.

Figure 1: Illustration of which parts of the memory hierarchy each block of A and B reside in during the execution of the micro-kernel.

implementation of the inner kernel (which is oftentimes assembly-coded).

At this point, we have $\widetilde{A}$ in the L2 cache and $\widetilde{B}$ in the L3 cache (or main memory). The next loop, indexed by j_r, now partitions $\widetilde{B}$ into column "slivers" (micro-panels) of width n_r. At a typical point of the computation, one such sliver is in the L1 cache, being multiplied by $\widetilde{A}$. Panel $\widetilde{B}$ was packed in such a way that this sliver is stored contiguously, one row (of width n_r) at a time. Finally, the inner-most loop, indexed by i_r, partitions $\widetilde{A}$ into row slivers of height m_r. Block $\widetilde{A}$ was packed in such a way that this sliver is stored contiguously, one column (of height m_r) at a time. The BLIS micro-kernel then multiplies the current sliver of $\widetilde{A}$ by the current sliver of $\widetilde{B}$ to update the corresponding $m_r \times n_r$ block of C. This micro-kernel performs a sequence of rank-1 updates (outer products) with columns from the sliver of $\widetilde{A}$ and rows from the sliver of $\widetilde{B}$.

A typical point in the compution is now captured by Figure 1. A $m_r \times n_r$ block of C is in the registers. A $k_c \times n_r$ sliver of $\widetilde{B}$ is in the L1 cache. The $m_r \times k_c$ sliver of $\widetilde{A}$ is streamed from the L2 cache. And so forth. The key takeaway here is that the layering described in this section can be captured by the five nested loops around the micro-kernel in Figure 2.
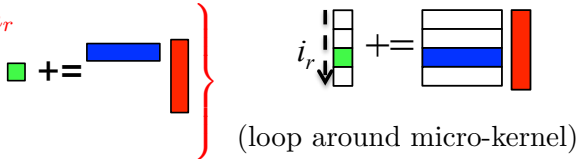
3 Opportunities for Parallelism

We have now set the stage to discuss opportunities for parallelism and when those opportunites may be advantageous. There are two key insights in this section:

- In GotoBLAS, the inner kernel is the basic unit of computation and no parallelization is incorporated

for $j_c = 0, \ldots, n-1$ **in steps of** n_c
  for $p_c = 0, \ldots, k-1$ **in steps of** k_c
    for $i_c = 0, \ldots, m-1$ **in steps of** m_c
      for $j_r = 0, \ldots, n_c-1$ **in steps of** n_r
        for $i_r = 0, \ldots, m_c-1$ **in steps of** m_r

$$C(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1) \overset{+}{=} \cdots$$

        endfor
      endfor
    endfor
  endfor
endfor

(loop around micro-kernel)
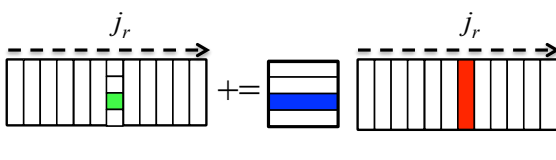
for $j_c = 0, \ldots, n-1$ **in steps of** n_c
  for $p_c = 0, \ldots, k-1$ **in steps of** k_c
    for $i_c = 0, \ldots, m-1$ **in steps of** m_c

      for $j_r = 0, \ldots, n_c-1$ **in steps of** n_r
        for $i_r = 0, \ldots, m_c-1$ **in steps of** m_r
        $C(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1) \overset{+}{=} \cdots$
        endfor
      endfor

    endfor
  endfor
endfor

for $j_c = 0, \ldots, n-1$ **in steps of** n_c
  for $p_c = 0, \ldots, k-1$ **in steps of** k_c

    for $i_c = 0, \ldots, m-1$ **in steps of** m_c
      for $j_r = 0, \ldots, n_c-1$ **in steps of** n_r
        for $i_r = 0, \ldots, m_c-1$ **in steps of** m_r
        $C(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1) \overset{+}{=} \cdots$
        endfor
      endfor
    endfor
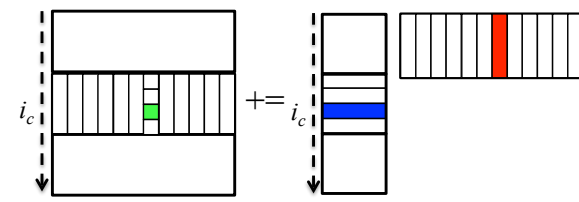
  endfor
endfor

Figure 2: Illustration of the three inner-most loops. The loops idexed by i_r and j_r are the loops that were hiddent inside the GotoBLAS inner kernel.

within that inner kernel[5]. The BLIS framework exposes two loops within that inner kernel, thus

---

[5]It is, of course, possible that more recent implementations by Goto deviate from this. However, these implementations are proprietary.

exposing two extra opportunities for parallelism, for a total of five.

- It is important to use a given memory layer wisely. This gives guidance as to *which* loop should be parallelized.

3.1 Parallelism within the micro-kernel

The micro-kernel is typically implemented as a sequence of rank-1 updates of the $m_r \times n_r$ block of C that is accumulated in the registers. Introducing parallelism over the loop around these rank-1 updates is ill-advised for three reasons: (1) the unit of computation is small, making the overhead considerable, (2) the different threads would accumulate contributions to the block of C, requiring a reduction across threads that is typically costly, and (3) each thread does less computation for each update of the $m_r \times n_r$ block of C, so the amortization of the cost of the update is reduced.

This merely means that parallelizing the loop around the rank-1 updates is not advisable. One could envision carefully parallezing the micro-kernel in other ways for a core that requires hyperthreading in order to attain peak performance. But that kind of parallelism can be described as some combination of parallelizing the first and second loop around the micro-kernel. We will revisit this topic later on.

The key for this paper is that the micro-kernel is a basic unit of computation for BLIS. We focus on how to get parallelism without having to touch that basic unit of computation. If the implementor chooses to introduce a level of parallelism in the micro-kernel, that parallelism is orthogonal to the addition parallelism introduced in the loops round the micro-kernel.

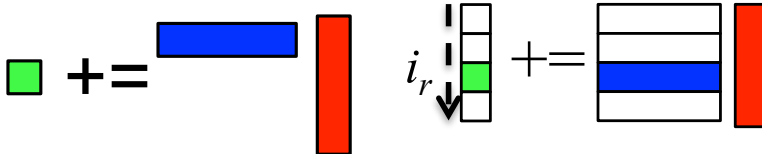## 3.2 Parallelizing the first loop around the micro-kernel (indexed by i_r).

Figure 3: Left: the micro-kernel. Right: the first loop around the micro-kernel.

Let us consider the first of the three loops in Figure 2. If one parallelizes the first loop around the micro-kernel (indexed by i_r), different instances of the micro-kernel are assigned to different theads. Our objective is to optimally use fast memory resources. In this case, the different threads share the same sliver of $\widetilde{B}$, which resides in the L1 cache.

Notice that regardless of the size of the matrices on which we operate, this loop has a fixed number of iterations, $\lceil \frac{m_c}{m_r} \rceil$, since it loops over m_c in steps of m_r. Thus, the amount of parallelism that can be extracted from this loop is quite limited. Additionally, a sliver of $\widetilde{B}$ is brought from the L3 cache into the L1 cache and then used during each iteration of this loop. When parallelized, less time is spent in this loop and thus the cost of bringing that sliver of $\widetilde{B}$ into the L1 cache is amortized over less computation. Notice that the cost of bringing $\widetilde{B}$ into the L1 cache may be overlapped by computation, so it may be completely or partially hidden. In this case, there is a minimum amount of computation required to hide the cost of bringing $\widetilde{B}$ into the L1 cache. Thus, parallelizing is acceptable only when this loop has a large number of iterations. These two factors mean that this loop should be parallelized only when the ratio of m_c to m_r is large. Unfortunately, this is not usually the case, as m_c is usually on the order of a few hundred elements.

3.3 Parallelizing the second loop around the micro-kernel (indexed by j_r).
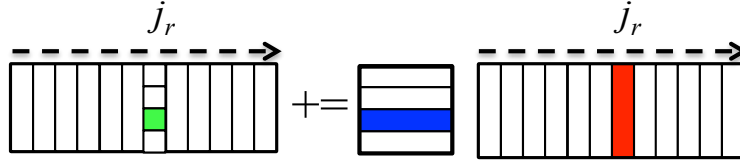

Figure 4: The second loop around the micro-kernel.

Now consider the second of the loops in Figure 2. If one parallelizes the second loop around the micro-kernel (indexed by j_r), each thread will be assigned a different sliver of $\widetilde{B}$, which resides in the L1 cache, and they will all share the same block of $\widetilde{A}$, which resides in the L2 cache. Then, each thread will multiply the block of $\widetilde{A}$ with its own sliver of $\widetilde{B}$.

Similar to the first loop around the micro-kernel, this loop has a fixed number of iterations, as it iterates over n_c in steps of n_r. The time spent in this loop amortizes the cost of packing the block of $\widetilde{A}$ from main memory into the L2 cache. Thus, for similar reasons as the first loop around the micro-kernel, this loop should be parallelized only if the ratio of n_c to n_r is large. Fortunately, this is almost always the case, as n_c is typically on the order of several thousand elements.

Consider the case where this loop is parallelized and each thread shares a single L2 cache. Here, one block $\widetilde{A}$ will be moved into the L2 cache, and there will be several slivers of $\widetilde{B}$ which also require space in the cache. Thus, it is possible that either $\widetilde{A}$ or the slivers of $\widetilde{B}$ will have to be resized so that all fit into the cache simultaneously. However, slivers of $\widetilde{B}$ are small compared to the size of the L2 cache, so this will likely not be an issue.

Now consider the case where the L2 cache is not shared, and this loop over n_c is parallelized. Each thread will pack part of $\widetilde{A}$, and then use the entire block of $\widetilde{A}$ for its local computation. In the serial case of GEMM, the process of packing of $\widetilde{A}$ moves it into a single L2 cache. In contrast, parallelizing this loop results in various parts of $\widetilde{A}$ being placed into *different* L2 caches. This is due to the fact that the packing of $\widetilde{A}$ is parallelized. Within the parallelized packing routine, each thread will pack a different part of $\widetilde{A}$, and so that part of $\widetilde{A}$ will end up in that thread's private L2 cache. A cache coherency protocol must then be relied upon to guarantee that the pieces of $\widetilde{A}$ are duplicated across the L2 caches, as needed. This occurs during and may be overlapped with computation. Because this results in extra memory movements, and because this relies on cache coherency, this may or may not be desireable, depending on the cost of duplication among the caches.

3.4 Parallelizing the third loop around the inner-kernel (indexed by i_c).

Next, consider the third loop around the micro-kernel at the bottom of Figure 2. If one parallelizes this first loop around what we call the macro-kernel (indexed by i_c), which corresponds to Goto's inner kernel, each thread will be assigned a different block of $\widetilde{A}$, which resides in the L2 cache, and they will all share the same row panel of $\widetilde{B}$, which resides in the L3 cache or main memory. Then each thread will multiply its own block of $\widetilde{A}$ with the shared row panel of $\widetilde{B}$.

Unlike the inner-most two loops around the micro-kernel, the number of iterations of this loop is not limited by the blocking sizes; rather, the number of iterations of this loop depends on the size of m. Notice that when m is less than the product of m_c and the degree of parallelization of the loop, then blocks of $\widetilde{A}$ will be smaller than optimal and performance will suffer.
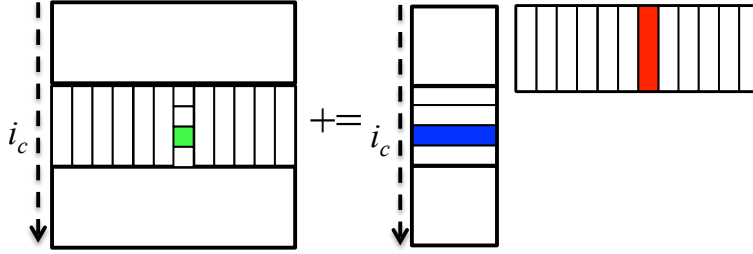
Figure 5: The third loop around the micro-kernel (first loop around Goto's inner kernel).

Now consider the case where there is a single, shared L2 cache. If this loop is parallelized, there must be multiple blocks of $\widetilde{A}$ in this cache. Thus, the size of each $\widetilde{A}$ must be reduced in size by a factor equal to the degree of parallelization of this loop. The size of $\widetilde{A}$ is $m_c \times k_c$, so either or both of these may be reduced. Notice that if we choose to reduce m_c, parallelizing this loop is equivalent to parallelizing the first loop around the micro-kernel. If instead each thread has its own L2 cache, each block of $\widetilde{A}$ resides in its own cache, and thus it would not need to be resized.

Now consider the case where there are multiple L3 caches. If this loop is parallelized, each thread will pack a different part of the row panel of $\widetilde{B}$ into its own L3 cache. Then a cache coherency protocol must be relied upon to place every portion of $\widetilde{B}$ in each L3 cache.

3.5 Parallelizing the fourth loop around the inner-kernel (indexed by p_c).
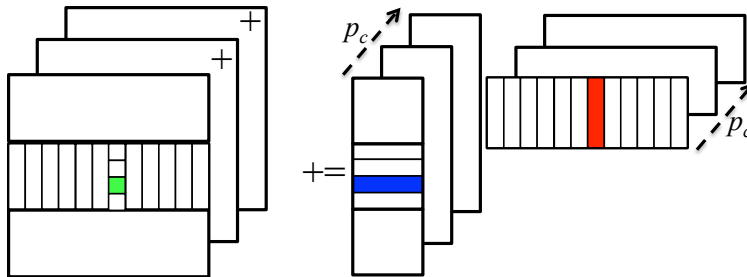

Figure 6: Parallelization of the p_c loop requires local copies of the block of C to be made, which are summed upon completion of the loop.

Consider the fourth loop around the micro-kernel. If one parallelizes this second loop around the macro-kernel (indexed by p_c), each thread will be assigned a different block of $\widetilde{A}$ and a different block of $\widetilde{B}$. Unlike in the previously discussed opportunities for parallelism, each thread will update the same block of C, potentially creating race conditions. Thus, parallelizing this loop either requires some sort of locking mechanism or the creation of copies of the block of C (initialized to zero) so that all threads can update their own copy, which is then followed by a reduction of these partial results, as illustrated in Figure 6. This loop should only be parallelized under very special circumstances. An example would be when C is small so that (1) only by parallelizing this loop can a satisfactory level of parallelism be achieved and (2) reducing (summing) the results is cheap relative to the other costs of computation. It is for this these reasons that so-called 3D (sometimes called 2.5D) distributed memory matrix multiplication algorithms [11, 2] choose this loop for parallelization (in addition to parallelizing one or more of the other

loops).

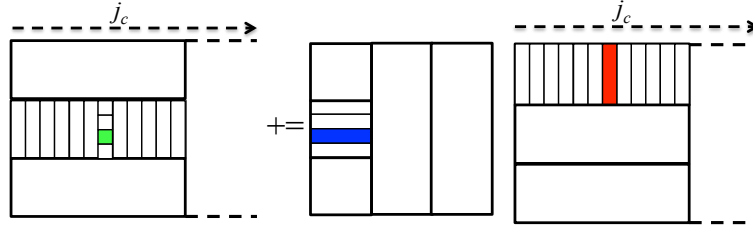## 3.6 Parallelizing the outer-most loop (indexed by j_c).

Figure 7: The fifth (outer) loop around the micro-kernel.

Finally, consider the fifth loop around the micro-kernel, third loop around the macro-kernel, and the outer-most loop. If one parallelizes this loop, each thread will be assigned a different row panel of $\widetilde{B}$, and each thread will share the whole matrix A which resides in main memory.

Consider the case where there is a single L3 cache. Then the size of a panel of $\widetilde{B}$ must be reduced so that multiple of $\widetilde{B}$ will fit in the L3 cache. If n_c is reduced, then this is equivalent to parallelizing the 2nd loop around the micro-kernel, in terms of how the data is partitioned among threads. If instead each thread has its own L3 cache, then the size of $\widetilde{B}$ will not have to be altered, as each panel of $\widetilde{B}$ will reside in its own cache.

Parallelizing this loop thus may be a good idea on multi-socket systems where each CPU has a separate L3 cache. Additionally, such systems often have a non-uniform memory access (NUMA) design, and thus it is important to have a separate panel of $\widetilde{B}$ for each NUMA node, with each panel residing in that node's local memory.

Notice that since threads parallelizing this loop do not share any packed buffers of $\widetilde{A}$ or $\widetilde{B}$, parallelizing this loop is, from a data-sharing perspective, equivalent to gaining parallelism outside of BLIS.

4 Intel Xeon Phi

We now discuss how BLIS supports high performance and scalability on the Xeon Phi architecture.

4.1 Architectural Details

The Xeon Phi has 60 cores, each of which has its own 512 KB L2 cache and 32 KB L1 data cache. Each core has four hardware threads, all of which share the same L1 cache. A core is capable of dispatching two instructions per clock cycle, utilizing the core's two pipelines. One of these may be used to execute vector floating point instructions or vector memory instructions. The other may only be used to execute scalar instructions or prefetch instructions. If peak performance is to be achieved, the instruction pipeline that is capable of executing floating point operations should be executing a fused multiply accumulate instruction (FMA) as often as possible. One thread may only issue one instruction to each pipeline every other clock cycle. Thus, utilizing two hardware threads is the minimum necessary to fully occupy the floating point unit. Using four hardware threads further alleviates instruction latency and bandwidth issues.

Although these hardware threads may seem similar to the hyper-threading found on more conventional CPUs, the fact is that hyper-threading is not often used for high-performance computing applications, and

these hardware threads must be used for peak performance.

4.2 The BLIS implementation on the Intel Xeon Phi

Because of the highly parallel nature of the Intel Xeon Phi, the micro-kernel must be designed while keeping the parallelism gained from the core-sharing hardware threads in mind. On conventional architectures, slivers of $\widetilde{A}$ and $\widetilde{B}$ are sized such that $\widetilde{B}$ resides in the L1 cache, and $\widetilde{B}$ is streamed from memory. However, this regime is not appropriate for the Xeon Phi. This is due to the fact that with four threads sharing an L1 cache, parallelizing in the m and n dimensions means that there must be room for at least two slivers of $\widetilde{A}$ and two slivers of $\widetilde{B}$ in the L1 cache. On the Xeon Phi, to fit so much data into the L1 cache would mean reducing k_c to a point where the cost of updating the $m_r \times n_r$ block of C is not amortized by enough computation. The solution is to instead only block for the L2 cache. In order for the GotoBLAS approach to be applied to this case, we can think of the region of the L2 cache that contains the slivers of $\widetilde{A}$ and $\widetilde{B}$ as a virtual L1 cache, where the cost of accessing its elements is the same as accessing elements in the L2 cache.

We now discuss the register and cache blocksizes for the BLIS implementation of Xeon Phi, as they affect how much parallelism can be gained from each loop. Various pipeline restrictions for the Xeon Phi mean that its micro-kernel must either update a 30×8 or 8×30 block of C. For this paper, we have chosen 30×8. Next, the block of $\widetilde{A}$ must fit into the 512 KB L2 cache. m_c is chosen to be 120, and k_c is chosen to be 240. There is no L3 cache, so n_c is only bounded by main memory, and by the amount of memory we want to use for the temporary buffer holding the panel of $\widetilde{B}$. For this reason we choose n_c to be 14400, which is the largest n dimension for any matrix we use for our experiments.[6]

4.3 Which loops to parallelize

The sheer number of threads and the fact that caches are shared within hierarchically nested loops suggests that we will want to consider parallelizing multiple loops. We will now take the insights from the last section to determine which loops would be appropriate to parallelize, and to what degree. In this section we will use the name of the index variable to identify each loop.

- The i_r loop: With an m_c of 120 and m_r of 30, this loop only has four iterations, thus it does not present a favorable opportunity for parallelization.

- The j_r loop: Since n_c is 14400, and n_r is only 8, this loop provides an excellent opportunity for parallelism. This is especially true among the hardware threads. The four hardware threads share an L2 cache, and if this loop is parallelized among those threads, they will also share a block of $\widetilde{A}$.

- The i_c loop: Since this loop has steps of 120, and it iterates over all of m, this loop provides a good opportunity when m is large. Additionally, since each core of the Xeon Phi has its own L2 cache, parallelizing this loop is beneficial because the size of $\widetilde{A}$ will not have to be changed as long as the loop is not parallelized by threads within a core. Notice that if the cores share an L2 cache, parallelizing this loop would result in multiple blocks of $\widetilde{A}$, each of which would have to be reduced in size since they would all have to fit into one L2 cache.

- The p_c loop: We do not consider this loop for reasons explained in Section 3.5 above.

---

[6] If we instead choose n_c to be 7200, performance drops by approximately 2 percent of the peak of the machine.

- The j_c loop: Since the Xeon Phi lacks an L3 cache, this loop provides no advantage over the j_r loop for parallelizing in the n dimension. It also offers worse spatial locality than the j_r loop, since there would be different buffers of $\widetilde{B}$.

We have now identified two loops as opportunities for parallelism on the Xeon Phi, the j_r and i_c loops.

4.4 Parallelism within cores

It is advantageous for hardware threads on a single core to parallelize the j_r loop. If this is done, then each hardware thread is assigned a sliver of $\widetilde{B}$, and the four threads share the same block of $\widetilde{A}$. If the four hardware threads are synchronized, they will access the same sliver of $\widetilde{A}$ concurrently. Not only that, if all four threads operate on the same region of $\widetilde{A}$ at the same time, one of the threads will load an element of $\widetilde{A}$ into the L1 cache, and all four threads will use it before it is evicted. Thus, parallelizing the j_r loop and synchronizing the four hardware threads will reduce bandwidth requirements of the micro-kernel. The synchronization of the four hardware threads is accomplished by periodically executing a barrier. Synchronizing threads may be important even when threads are located on different cores. For example, multiple cores conceptually will share a sliver of $\widetilde{B}$, which is read into their private L2 caches. If they access the $\widetilde{B}$ sliver at the same time, the sliver will be read just once out of L3 (or memory) and replicated using the cache coherence protocol. However, if cores fall out of synch, a sliver of $\widetilde{B}$ may be read from main memory multiple times. This may penalize performance or energy.

For our Xeon Phi experiments, the four threads on a core parallelize the j_r loop, and a barrier is executed every 8 instances of the micro-kernel. However, we do not enforce any synchronization between cores.

4.5 Parallelism between cores

As noted, it is particularly advantageous to parallelize the i_c loop between cores as each core has its own L2 cache. However, if parallelism between cores is only attained by this loop, performance will be poor when m is small. Also, all cores will only work with an integer number of full blocks of $\widetilde{A}$ (where the size of the $\widetilde{A}$ is $m_c \times k_c$) when m is a multiple of 7200. For this reason, we seek to gain parallelism in both the m and n dimensions. Thus, we parallelize the j_r loop in addition to the i_c loop to gain parallelism between cores, even though this incurs the extra cost of the cache-coherency protocol to duplicate all of $\widetilde{A}$ to each L2 cache.

4.6 Performance results

Given that (1) each core can issue one floating point multiply-accumulate instruction per clock cycle, and (2) the SIMD vector length for double-precision real elements is 8, each core is capable of executing 16 floating point operations per cycle, where a floating point operation is either a floating point multiply or addition. At 1.1 GHz, this corresponds to a peak of 17.6 GFLOPS per core, or 1056 GFLOPS for 60 cores. In the performance results presented in this paper, the top of each graph represents the theoretical peak of that machine.

Figure 8 compares the performance of different parallelization schemes within BLIS on the Xeon Phi. There are four parallelization schemes presented. They are labeled with how much parallelism was gained from the i_c and j_r loops. In all cases, parallelization within a core is done by parallelizing the j_r loop. Single-thread results are not presented, as such results would be meaningless on the Xeon Phi.

The case labeled 'j_r: 240 way', where all parallelism is gained from the j_r loop, yields very poor performance. Even when $n = 14400$, which is the maximum tested (and a rather large problem size),
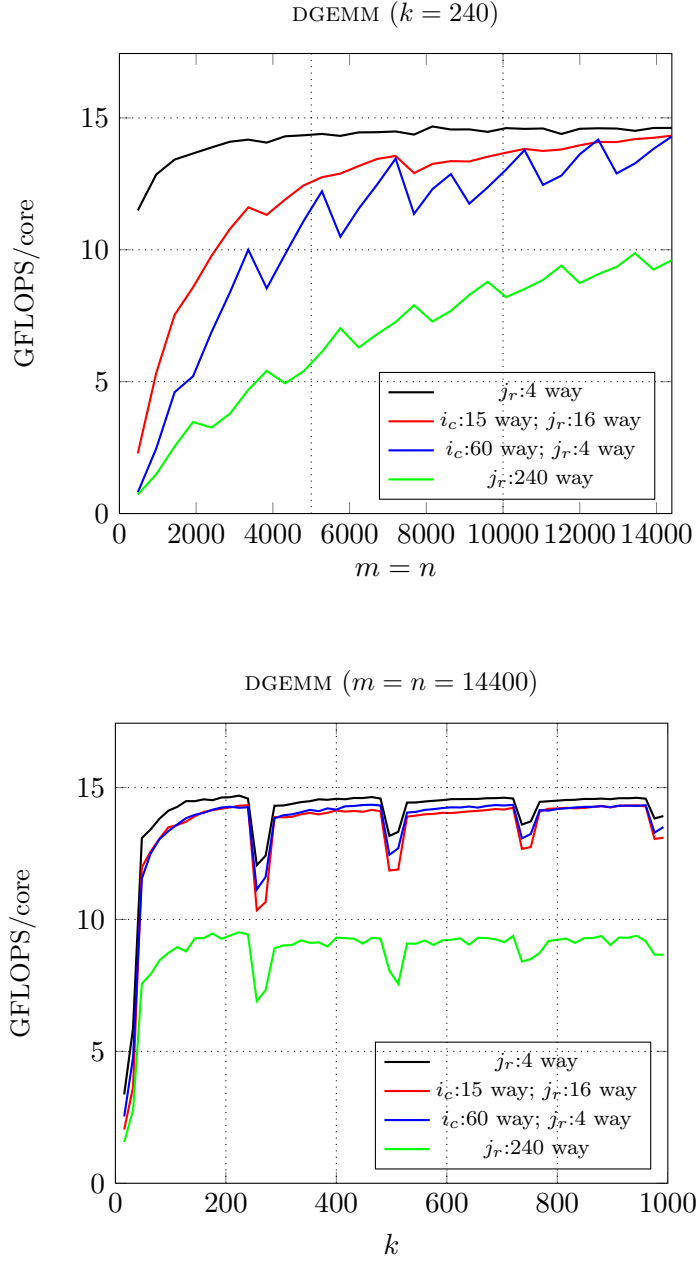
Figure 8: Different parallelization schemes of BLIS on the Intel Xeon Phi. 'i_c:n way' indicates n-way parallelization of the third loop (indexed by i_c) around the micro-kernel, and 'j_r:n way' indicates n-way parallelization of the second loop (indexed by j_r) around the micro-kernel.

each thread is only multiplying each $\widetilde{A}$ with seven or eight slivers of $\widetilde{B}$. In this case, not enough time is spent in computation to amortize the packing of $\widetilde{A}$. Additionally, $\widetilde{A}$ is packed by all threads and then the cache coherency protocol duplicates all slivers of $\widetilde{A}$ among the threads (albeit at some cost due to extra
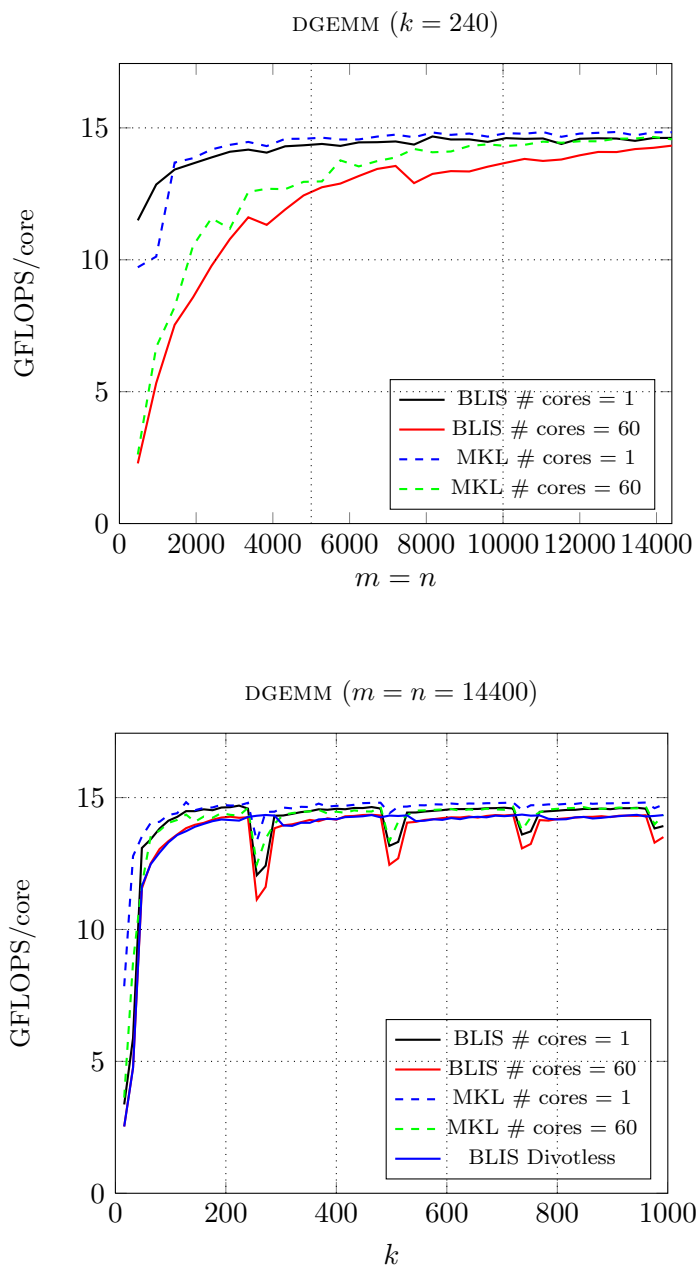
Figure 9: Performance comparision of BLIS with MKL on the Intel Xeon Phi.

memory traffic). Finally, $\widetilde{A}$ is rather small compared to the number of threads, since it is only 240×120. A relatively small block of $\widetilde{A}$ means that there is less opportunity for parallelism in the packing routine. This makes load balancing more difficult, as some threads will finish packing before others and then sit idle.

Next consider the case labeled 'i_c:60 way; j_r:4 way'. This is the case where parallelism between cores

is gained from the i_c loop, and it has good performance when m is large. However, load balancing issues arise when m_r multiplied by the number of threads parallelizing the i_c loop does not divide m (that is, when m is not divisible by 1800). This is rooted in the $m_r \times n_r$ micro-kernel's status as the basic unit of computation. Now consider the case labeled 'i_c: 15; j_r:16'. This case ramps up more smoothly, especially when m and n are small.

In Figure 9, we compare the best BLIS implementation against Intel's Math Kernel Library (MKL), which is a highly-tuned implementation of the BLAS. For the top graph, we compare the 'i_c:15 way; j_r:16 way' scheme against MKL when m and n vary. For the bottom graph, we use the 'i_c:60 way; j_r:4 way' scheme, since it performs slightly better when m is large. Notice that this case is particularly favorable for this parallelization scheme because each thread is given an integer number of blocks of $\widetilde{A}$. This only happens when m is divisible by 7200.

In the bottom graph of Figure 9, when both m and n are fixed to 14400, notice that there are 'divots' that occur when k is very slightly larger than k_c, which is 240 in the BLIS implementation of GEMM, and evidently in the MKL implementation of GEMM as well. When k is just slightly larger than a multiple of 240, an integer number of rank-k updates will be performed with the optimal blocksize k_c, and one rank-k update will be performed with a smaller rank. The rank-k update with a small value of k is expensive because in each micro-kernel call, an $m_r \times n_r$ block of C must be both read from and written to main memory. When k is small, this update of the $m_r \times n_r$ submatrix of C is not amortized by enough computation. It is more efficient to perform a single rank-k update with a k that is larger than the optimal k_c than to perform a rank-k update with the optimal k_c followed by a rank-k update with a very small value of k. This optimization is shown in the curve in Figure 9 labeled "BLIS Divotless".

Figure 9 shows BLIS attaining very similar performance to that of Intel's highly-tuned MKL, falling short by only one or two percentage points from the peak performance of the Xeon Phi. We also demonstrate great scaling results when using all 60 cores of the machine. Additionally, we demonstrate that the performance 'divots' that occur in both MKL and BLIS when k is slightly larger than some multiple of 240 can be eliminated.

5 IBM Blue Gene/Q

We now discuss how BLIS supports high performance and scalability on the IBM Blue Gene/Q PowerPC A2 architecture [8].

5.1 Architectural Details

The Blue Gene/Q PowerPC A2 processor has 16 cores available for use by applications. Much like the Intel Xeon Phi, each core is capable of using up to four hardware threads, each with its own register file. The PowerPC A2 supports the QPX instruction set, which supports SIMD vectors of length four for double-precision real elements. QPX allows fused multiply-accumulate instructions, operating on SIMD vectors of length four. This lets the A2 execute 8 flops per cycle.

The 16 cores of Blue Gene/Q that can be used for GEMM share a single 32 MB L2 cache. This cache is divided into 2 MB slices. When multiple threads are simultaneously reading from the same slice, there is some contention between the threads. Thus there is a cost to having multiple threads access the same part of $\widetilde{A}$ at the same time. The L2 cache has a latency of 82 clock cycles and 128 byte cache lines.

Each core has its own L1 prefetch, L1 instruction, and L1 data cache. The L1 prefetch cache contains the data prefetched by the stream prefetcher and has a capacity of 4 KB [4]. It has a latency of 24 clock

cycles and a cache line size of 128 byes. The L1 data cache has a capacity of 16 KB, and a cache line size of 64 bytes. It has a latency of 6 clock cycles [5].

The PowerPC A2 has two pipelines. The AXU pipeline is used to execute QPX floating point operations. The XU pipeline can be used to execute memory and scalar operations. Each clock cycle, a hardware thread is allowed to dispatch one instruction to one of these pipelines. In order for the A2 to be dispatching a floating point instruction each clock cycle, every instruction must either execute on the XU pipeline, or it must be a floating point instruction. Additionally, since there are inevitably some instructions that are executed on the XU pipeline, we use four hardware threads so that there will usually be an AXU instruction available to dispatch alongside each XU instruction.

5.2   The BLIS implementation on the IBM PowerPC A2

As on the Intel Xeon Phi, $\widetilde{A}$ and the sliver of $\widetilde{B}$ reside in the L2 cache and no data resides in the L1 cache. (Notice that the amount of L1 cache per thread on the A2 is half that of the Xeon Phi.)

For the BLIS PowerPC A2 implementation, we have chosen m_r and n_r to both be 8. The block of $\widetilde{A}$ takes up approximately half of the 32 MB L2 cache, and in the BLIS implementation, m_c is 1024 and k_c is 2048. The PowerPC A2 does not have an L3 cache and thus n_c is limited by the size of memory; therefore, we have choosen a rather large value of $n_c = 10240$.

5.3   Which loop to parallelize

While there are fewer threads to use on the PowerPC A2 than on the Xeon Phi, 64 hardware threads is still enough to require the parallelization of multiple loops. Again, we refer to each loop by the name of its indexing variable.

- The i_r loop: With an m_c of 1024 and m_r of 8, this loop has many iterations. Thus, unlike the Intel Xeon Phi, the first loop around the micro-kernel presents an excellent opportunity for parallelism.

- The j_r loop: Since n_c is large and n_r is only 8, this loop also provides an excellent opportunity for parallelism. However when threads parallelize this loop, they share the same $\widetilde{A}$, and may access the same portions of $\widetilde{A}$ concurrently. This poses problems when it causes too many threads to access the same 2 MB portion of the L2 cache simultaneously.

- The i_c loop: Since all threads share the same L2 cache, this loop has similar advantages as the i_r loop. If multiple threads parallelize this loop, $\widetilde{A}$ will have to be reduced in size. This reduction in size reduces the computation that amortizes the movement of each sliver of $\widetilde{B}$ into the virtual L1 cache. Notice that if we reduce m_c, then parallelizing the i_c loop reduces this cost by the same amount as parallelizing i_r.

- The p_c loop: Once again, we do not consider this loop for parallelization.

- The j_c loop: This loop has the same advantages and disadvantages as the j_r loop, except that this loop should not be parallelized among threads that share a core, since they will not then share a block of $\widetilde{A}$.

Since the L2 cache is shared, and there is no L3 cache, when parallelizing for the PowerPC A2, our choice is as follows: we may parallelize the i_c and i_r loops, or the j_c and j_r loops. In both of these cases, we prefer the inner loops to the outer loops. The reason for this is two-fold. Firstly, it is convenient to not change any of the cache blocking sizes from the serial implementation of BLIS when parallelizing. But

more importantly, parallelizing the inner loops instead of the outer loops engenders better spatial locality, as there will be one contiguous block of memory, instead of several blocks of memory that may not be contiguous.

5.4   Performance results

Like the Xeon Phi, each core of the PowerPC A2 can issue one double-precision fused multiply-accumulate instrucion each clock cycle. The SIMD vector length for double-precision arithmetic is 4, so each core can execute 8 floating point operations per cycle. At 1.6 GHz, a single core has a double-precision peak performance of 12.8 GFLOPS. This becomes the top line in Figures 10 and 11. The theoretical peak with all 16 cores is 204.8 GFLOPS.

Figure 10 compares the performance of different parallelization schemes within BLIS on the PowerPC A2. It is labeled similarly to Figure 8, described in the previous section. Notice that all parallelization schemes have good performance when m and n are large, but the schemes that only parallelize in either the m or the n dimensions have performance that varies according to the amount of load balancing. Proper load balancing for the 'i_r:64 way' case is only achieved when m is divisible by 512, and similarly, proper load balancing for the 'j_r:64 way' case is only achieved when n is divisible by 512.

The performance of BLIS is compared with that of ESSL in Figure 11. The parallelization scheme used for this comparision is the one labeled 'j_r:8 way; i_r:8 way'. Notice that parallel performance scales perfectly to 16 cores for large m and n.

6   Concluding remarks

In this paper, we exposed the five loops around a micro-kernel that underly matrix-matrix multiplication within the BLIS framework. We discussed where, at a prototypical point in the computation, data resides and used this to motivate insights about opportunities for parallelizing the various loops. We discussed how parallelizing the different loops affects the sharing of data and amortization of data movement. These insights were then applied to the parallelization of this operation on two architectures that require many threads to achieve peak performance: The IBM PowerPC A2 that underlies the Blue Gene/Q supercomputer and the Intel Xeon Phi. It was shown how parallelizing multiple loops is the key to high performance and scalability. While on the Xeon Phi the resulting performance was respectable and similar to that of Intel's MKL library, for the PowerPC A2, the more careful parallelization yielded a considerable performance boost over IBM's ESSL. Regardless, adding parallelism to the BLIS framework for matrix multiplication appears to support high performance on these architectures. This will soon give the community another open source solution for multithreaded BLAS.

A curiosity is that on both of these architectures the L1 cache is too small to support the multiple hardware threads that are required to attain near-peak performance. This is overcome by using a small part of the L2 cache for data that on more conventional architectures resides in the L1 cache. It will be interesting to see whether this will become a recurring theme in future many-core architectures.
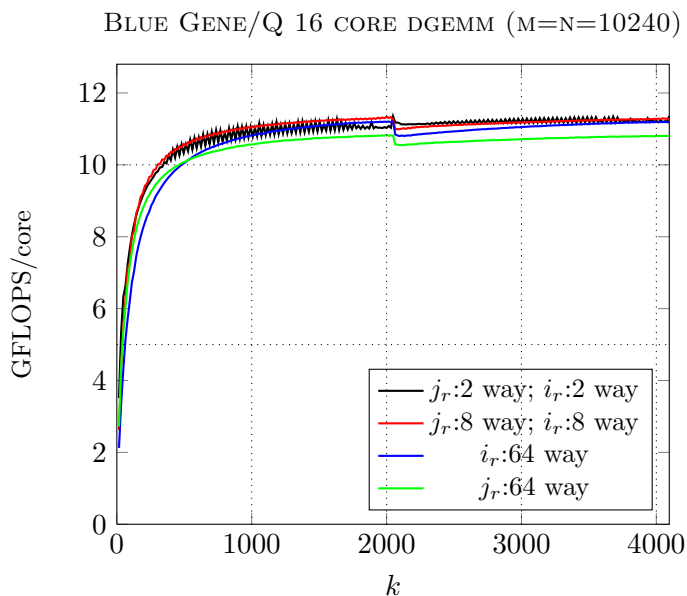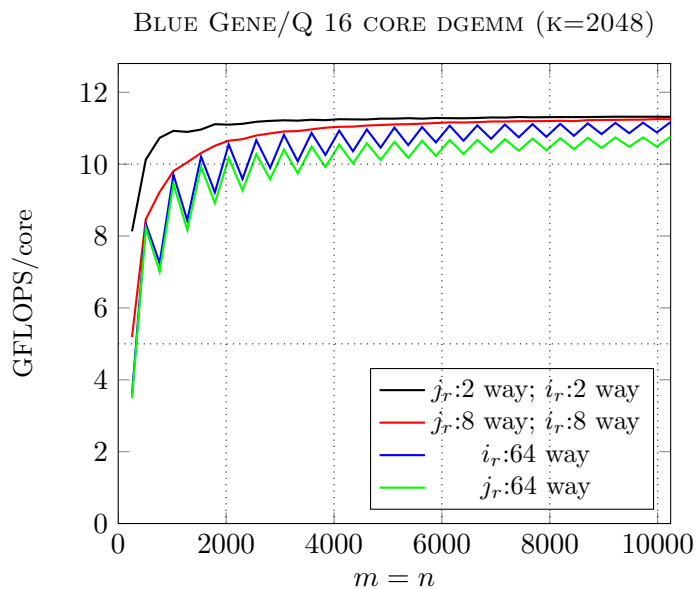
Figure 10: Different parallelization schemes for the IBM PowerPC A2. 'j_r:n way' indicates n-way parallelization of the second loop (indexed by j_r) around the micro-kernel, and 'i_r:n way' indicates n-way parallelization of the first loop (indexed by i_r) around the micro-kernel.

16
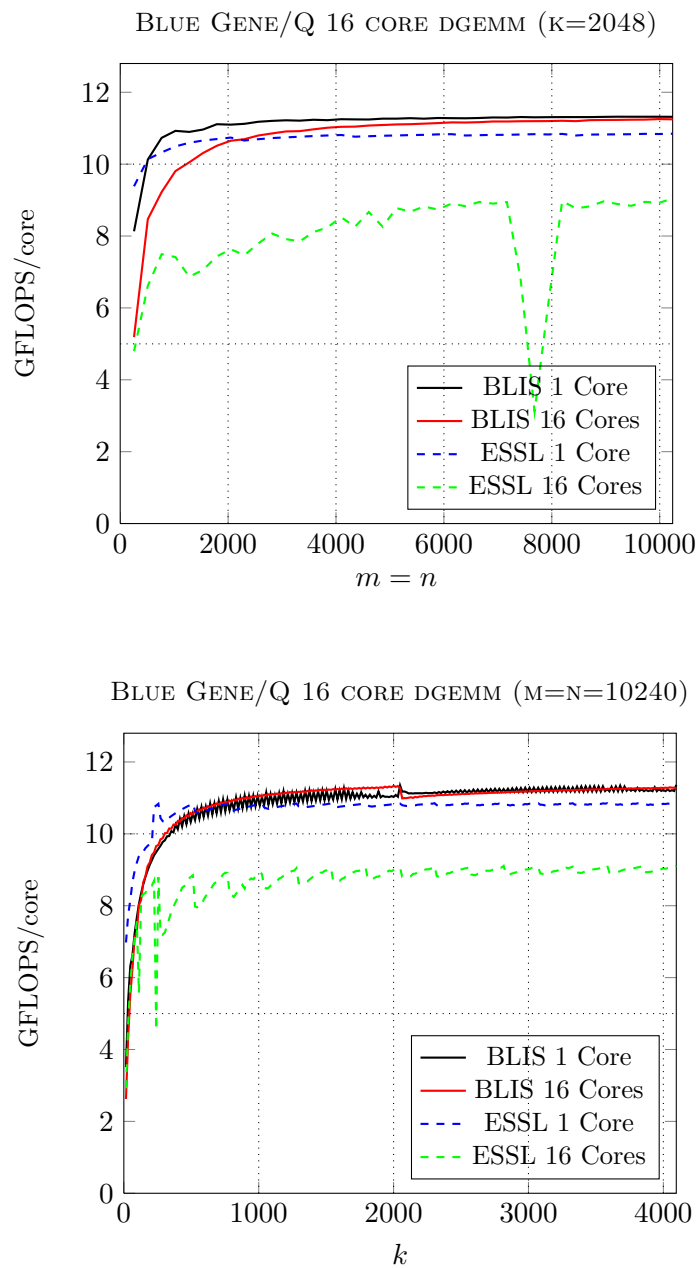
Figure 11: Performance comparision of BLIS with ESSL on the PowerPC A2.

References

[1] R. C. Agarwal, F. Gustavson, and M. Zubair. Exploiting functional parallelism on Power2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, 38(5), 1994.

[2] R.C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39, 1995.

[3] J. Bilmes, K. Asanovic, C.W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*. ACM SIGARC, July 1997.

[4] I-Hsin Chung, Changhoan Kim, Hui-Fang Wen, and Guojing Cong. Application data prefetching on the IBM Blue Gene/Q supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 88:1–88:8, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[5] Megan Gilge. IBM system Blue Gene solution: Blue Gene/Q application development. *IBM Redbook SG24-7948-01*, June 2013.

[6] Kazushige Goto and Robert van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft.*, 34(3), May 2008.

[7] John Gunnels. Making good enough... better - addressing the multiple objectives of high-performance parallel software with a mixed global-local worldview. Synchronization-reducing and Communication-reducing Algorithms and Programming Models for Large-scale Simulations, talk presented at ICERM, 2012.

[8] R.A. Haring, M. Ohmacht, T.W. Fox, M.K. Gschwind, D.L. Satterfield, K. Sugavanam, P.W. Coteus, P. Heidelberger, M.A. Blumrich, R.W. Wisniewski, A. Gara, G.L.-T. Chiu, P.A. Boyle, N.H. Chist, and Changhoan Kim. The IBM Blue Gene/Q compute chip. *Micro, IEEE*, 32(2):48 –60, march-april 2012.

[9] Alexander Heinecke, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, Alexander Kobotov, Roman Dubtsov, Greg Henry, Aniruddha G. Shet, George Chrysos, and Pradeep Dubey. Design and implementation of the linpack benchmark for single and multi-node systems based on intel(r) xeon phi(tm) coprocessor. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2013)*, 2013. To appear.

[10] Bryan A. Marker, Field G. Van Zee, Kazushige Goto, Gregorio Quintana-Ortí, and Robert A. van de Geijn. Toward scalable matrix multiply on multithreaded architectures. In *European Conference on Parallel and Distributed Computing*, pages 748–757, February 2007.

[11] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] 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 Trans. Math. Soft.*, 2013. submitted.

[13] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for rapid instantiation of BLAS functionality. *ACM Trans. Math. Soft.*, 2013. To appear.

[14] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.

[15] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems (ICPADS)*, 2012.