The Dissertation Committee for Ardavan Pedram
certifies that this is the approved version of the following dissertation:

Algorithm/Architecture Codesign of Low Power and High Performance Linear Algebra Compute Fabrics

Committee:

Andreas Gerstlauer, Supervisor

Robert van de Geijn, Supervisor

Peter Hofstee

Lizy John

Keshav Pingali

Algorithm/Architecture Codesign of Low Power and High Performance Linear Algebra Compute Fabrics

by

Ardavan Pedram, B.E., M.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2013

Dedicated to My Late Role Models:

Doctor Bahman Pedram and Professor Caro Lucas

Acknowledgments

"Losing family obliges us to find our family. Not always the family that is our blood, but the family that can become our blood. And should we have the wisdom to open our door to this new family, we will find that the wishes and hopes we once had ... for the father who once guided us, for the brother who once inspired us, ... those wishes are there for us once again."

I wish to thank the incredible people whose company fulfilled different aspects of my new life with joy and the sense of creativity. People that intentionally and unintentionally taught me great lessons about excellence in action. People who maybe without even knowing it became my family, inspired me to dream beyond my limits, and guided me to the path of achieving those dreams.

First and foremost, I want to express my gratitude to my supervisors Professor Robert van de Geijn and Professor Andreas Gerstlauer. I am glad that I got the chance to know and introduce these great guys to each other and gain success while being protected in their safe hands.

The first time I got to know Robert goes back to 2006 when I found his paper on the material and it got me hooked. The excellent style of explanation got me interested on Robert's research and I read several other papers of his work wondering if I could join his group someday. I met him personally and after two short conversations he offered to hire me in his group and my dream came true. Robert is an excellent teacher and a caring supervisor. Robert and Doctor Maggie Mayers, who I want to thank dearly, went far beyond a student-advisor relationship and made me feel like I have a family here. Some relationships just develop by themselves and this one turned my whole PhD studies into a joyful life.

I first met Andreas when he had just joined UT as a new faculty member. I took his course and experienced a great instructor-student relationship. Andreas is always patient with questions and gives me confidence to ask more and learn more. He has a lot of energy and interest in challenging research. As time went by in my PhD studies he backed me up more and more in all of the tight deadlines for different venues and worked persistently to keep me motivated. Andreas always considered my preferences in his decisions and it makes me proud to have an advisor who understands me so deeply.

I want to thank each of the committee members who were great motivators and supporters of this project. A special thanks to Doctor Peter Hofstee who I am honored to have on my committee. Peter spent several hours in different locations having one-to-one discussions to give me perspective and encouraged me to follow the correct scientific approach. I am really grateful to Professor Lizy John for her support when I needed guidance and was struggling to make a correct decisions with my graduate studies. I experienced one of the most exciting courses in graduate school with her and learnt the process of critical thinking from her. Lizy has always been a great and kind person to refer to when I had questions about my project and gave me directions to correct resources and people in the field. A special thank you is extended to Professor Keshav Pingali for being a great role model. Keshav is a great scholar with an extra-ordinary personality that I have thorough respect for. Keshav is an expert in the field and gave me great subjects to think about for future work and possible extensions to my ideas. I learnt from him a great deal about the applications of my research topic in science and how to think outside of the box. He gave me the great opportunity of teaching his class as a guest lecturer.

I met great scholars and professors in UT during these last years. Specially, I want to thank Professor Earl Swartzlander. Earl has always emboldened me in my career to strive higher. He always found time for me to talk about our research overlap and gave me great advice on how to approach new problems with confidence. He introduced me to many successful people whose work helped me substantially. One of these great people was Doctor Eric Quinnell. I want to express my gratitude to Eric for his support and his research. I used his dissertation to learn about floating-point units. Eric also read my first paper's early drafts when I needed guidance to lay the foundation of my research correctly.

I want to specially thank my main collaborator Doctor John McCalpin. I first met John in 2012 in Texas Advance Computing Center and bombarded him with my questions. He answered all of them in great detail and his intuition helped me to start extending my research. After a year, I returned to him with a plan and he kindly spent his research time with me. He also gave me the opportunity to work under his supervision in Texas Advanced Computing Center as an intern to experience even more. I always found valuable gems of knowledge in his conversations with me and am really grateful for his support and his friendship.

My friends in the FLAME research group kept me going for these years. I want to show gratitude to Field van Zee who is a great scientist and a genuine friend. He kindly offered to read my papers and gave me great feedback on the material. Field always answered my questions with great patience. I want to express my thanks to Tze Meng Low, Bryan Marker, and Martin Schatz with whom I shared many of my research and life issues and they always offered a hand to help. I have had great discussions with these friends and never felt bored. I want to thank Doctor Jack Poulson who is my first American friend and cared for me in time of hardship. Jack is an extremely smart and a valuable friend. My friendship with Jack is a case that proved to me that there is a state of mind that two people can reach to understand each other perfectly regardless of their backgrounds. I also want to thank Doctor Victor Eijkhout for the many occasions that he gave me great feedback on my work.

I want to thank my co-authors and collaborators from The University of Wisconsin Maddison Doctor Zohaib Gilani and Professor Nam Sung Kim, and from AMD Research Doctor Michael Schulte. I first met Zohaib and Mike in ASAP 2011 and we discussed possible collaboration. Mike is a great motivator and encouraged me to follow the path of my research. Our collaboration yielded into a successful paper in ASAP 2012.

Through these years I was blessed to get to know exceptional individuals in different fields of art and science and each inspired me to be a better person. These friends helped me to know my weaknesses better and helped me to solve my life struggles. Their care for me made me believe in myself and the bliss of friendship more and more. I want to thank Roja Najafi and Doctor Behnam Robatmilli, who were my best Persian friends in Austin and gave me unconditional support like a sister and a brother. I wish to thank Bonnie Gammill who taught me how to see beauty in life with a more optimistic view and who cared for me in great deal and made me feel appreciated and loved. Finally, I am indebted to our Graduate Program Coordinator Melanie Gulick in great deal. I want to thank her for being a true friend to me who like my own mother gave me great advice about my personal life and my graduate studies. I want to thank my other great friends Doctor Arjang Hassibi, Benini, Khosrow Afroozeh, Doctor Mehdi Haghshenas, Babak Fallah, Shahrzad Mirkhani, Doctor Sahar Ayazian, Maysam Lavasani, Khubaib, and Mehmet Basoglu.

Finally, I want to thank my family members who always believed in me. I want to thank my parents Doctor Bahman Pedram and Nikou Zandie. I did not get a chance to visit my dad for a last time although he was asking for me all the time. I owe whatever makes me happy to my mother who patiently hid all the sad stories back home and went trough all the hardship by herself. I want to thank my great sister Doctor Elham Pedram. Elham is the burning flame of life and a great motivator to me and people around her. She is a true role model and a great mother. I am obliged to my uncle Robert Kayvon who took care of me like my own father would and kept in touch with me all of these years abroad. Bob is the most optimistic and generous person I have ever met. His door was always open to me and he treats me like his own son.

Algorithm/Architecture Codesign of Low Power and High Performance Linear Algebra Compute Fabrics

Publication No. ____________________

Ardavan Pedram, Ph.D.
The University of Texas at Austin, 2013

Supervisors: Andreas Gerstlauer
Robert van de Geijn

In the past, we could rely on technology scaling and new micro-architectural techniques to improve the performance of processors. Nowadays, both of these methods are reaching their limits. The primary concern in future architectures with billions of transistors on a chip and limited power budgets is power/energy efficiency. Full-custom design of application-specific cores can yield up to two orders of magnitude better power efficiency over conventional general-purpose cores. However, a tremendous design effort is required in integrating a new accelerator for each new application.

In this dissertation, we present the design of specialized compute fabrics that maintain the efficiency of full custom hardware while providing enough flexibility to execute a whole class of coarse-grain operations. The broad vision is to develop integrated and specialized hardware/software solutions that are co-optimized and co-designed across all layers ranging from the basic hardware foundations all the way to the application programming support through standard linear algebra libraries.

We try to address these issues specifically in the context of dense linear algebra applications. In the process, we pursue the main questions that architects will face while designing such accelerators. How broad is this class of applications that the accelerator can support? What are the limiting factors that prevent utilization of these accelerators on the chip? What is the maximum achievable performance/efficiency? Answering these questions requires expertise and careful codesign of the algorithms and the architecture to select the best possible components, datapaths, and data movement patterns resulting in a more efficient hardware-software codesign. In some cases, codesign reduces complexities that are imposed on the algorithm side due to the initial limitations in the architectures.

We design a specialized Linear Algebra Processor (LAP) architecture and discuss the details of mapping of matrix-matrix multiplication onto it. We further verify the flexibility of our design for computing a broad class of linear algebra kernels. We conclude that this architecture can perform a broad range of matrix-matrix operations as complex as matrix factorizations, and even Fast Fourier Transforms (FFTs), while maintaining its ASIC level efficiency.

We present a power-performance model that compares state-of-the-art CPUs and GPUs with our design. Our power-performance model reveals sources of inefficiencies in CPUs and GPUs. We demonstrate how to overcome such inefficiencies in the process of designing our LAP.

As we progress through this dissertation, we introduce modifications of the original matrix-matrix multiplication engine to facilitate the mapping of more complex operations. We observe the resulting performance and efficiencies on the modified engine using our power estimation methodology. When compared to other conventional architectures for linear algebra applications and FFT, our LAP is over an order of magnitude better in terms of power efficiency. Based on our estimations, up to 55 and 25 GFLOPS/W single- and double-precision efficiencies are achievable on a single chip in standard 45nm technology.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Background

Computer systems with the equivalent of 1 million to 10 million processing elements (e.g., cores) are on the horizon, heralding the age of exascale computing. This level of parallelism will be achieved by configuring computers as clusters on the macro level and, several layers down, in each processor exploiting VLSI technology that will allow on the order of 50 billion transistors to be packed onto a single chip [126]. In the past, we could rely on technology scaling to provide exponentially more and faster transistors in a constant area and at constant power with each new chip generation [31]. However, in the future only a fraction of this integration capacity can be utilized due to power constraints [36]. This provides the opportunity to integrate specialized cores that are only utilized when needed. At the same time, sustaining significant continued performance improvements will drive the need for optimization through specialization. One of the key questions going forward will be how to minimize, or at least greatly reduce, the power consumption while retaining or improving the achieved performance.

It is well known that full-custom, application-specific design of on-chip hardware accelerators, can provide orders of magnitude improvements in both power and performance for a wide variety of application domains [55, 154]. The flexibility provided by the programmability of general-purpose machines comes with inherent overhead. By contrast, in application-specific designs implementations are hardwired to directly realize the desired computation in fixed hardware. This is possible in domains such as embedded or mobile computing where applications are standardized and exponentially growing costs of chip design can be reaped across a large volume of units. The question is whether these concepts can be applied to a broader class of more general applications.

1.1 Thesis Statement

Modern processors often integrate application-specific IP cores to meet power restrictions of the dark silicon era. However, each Application-specific IP core is limited to only one or a few applications/routines because the cost of extra flexibility is substantial loss in efficiency.

An accelerator design without instruction pipeline or register file, and optimized for matrix multiplication supports enough flexibility to perform level-3 BLAS, matrix-factorizations, other complicated linear algebra operations, and FFTs only by exploiting the competence of algorithm/architecture codesign. It is conjectured that such a design with all of the mentioned flexibility maintains at least an order of magnitude better power and area efficiency compared to current existing programmable architectures.

The broader vision of this project that goes beyond this dissertation is to develop integrated and specialized hardware/software solutions that are co-optimized and co-designed across all layers ranging from the basic hardware foundations all the way to the application programming support through standard linear algebra packages. We study upper limits on performance/power ratios that can be achieved, and fundamentally investigate both limitations in current architectures and opportunities for targeted improvements in future architectures that are specially designed to efficiently support this crucial class of operations.

1.2 Linear Algebra Processor Overview

We target an ASIC implementation that will allow us to fully exploit state-of-the-art technologies instead of programmable hardware like FPGAs. Within this context, our goal is to develop a fixed Linear Algebra Processor (LAP) architecture that avoids inefficiencies of general-purpose processors and GPUs. We remove the overheads of program pipeline and rearrange processing elements in a more efficient way. Finally, in contrast to full custom specialized accelerators, our architecture is flexible enough to optimally execute different matrix operations, but with the same level of efficiency.

We recognize that our class of linear algebra operations essentially consists entirely of Multiply-Accumulate (MAC) computations with regular and predictable, looping access patterns. As such, we design a datapath that consists of specialized MAC units, which include local accumulators to avoid the need for unnecessary transfers to/from the register file in every operation.

We build on the SIMD concept of replicating Functional Units (FUs) to exploit parallelism; instead of communicating through shared storage, wasting cycles and instructions, our approach is based on partitioned and distributed memories local to each FU. Memory hierarchies and interconnects are specifically designed to realize available locality and required access patterns with efficient reuse as well as careful prefetching of data that moves between memory layers. Control is predominantly hardwired with a minimal set of micro-coded commands to switch between different processing modes.

A taxonomy of accelerators and the possible programming models in accelerator systems is proposed by Cascaval et al. [21], which we will discuss further in the related work. Here, we classify the LAP along this design space taxonomy. Based on all of the possible choices in the design space of accelerators, [21] introduces a classification across three dimensions. The following summarizes the possible choices and where the LAP design stands in this design space.

1. **Architecture type**: Fixed architectures such as Floating-Point Units (FPUs), programmable architectures like GPUs, or reprogrammable architectures like FPGAs are the possible choices. Our proposed architecture is reprogrammable and flexible so it can handle variety of linear algebra problems by microprogrammed control.

2. **Invocation and Completion**: There is a correlation between the invocation granularity and the coupling of the accelerator to the host sys-

tem. In instruction-level invocation, Streaming SIMD Extensions (SSE) or FPUs are invoked as a part of the Instruction Set Architecture (ISA). In command packet invocation, an accelerator is connected as a device with memory mapped I/O, and its invocation is asynchronous. In task-level invocation, the accelerators are programmable standalone systems with coarse-grain invocation and are typically asynchronous. Finally, workload systems are standalone systems connected to the host through a network and perform the entire tasks offline.

The invocation granularity of our LAP is coarse-grain at the task level to maintain maximum utilization of the host processor. The host could assign a series of tasks on the same data without interfering in between. The LAP interrupts the host when the result matrix is ready.

3. **Memory Addressing**: Memory addressing determines whether the address space of the host is shared with the accelerator. Memory partitioning shows if the accelerator's addressable memory is completely distributed, shared with, or hidden from the CPU. Possible coherency of address spaces with the CPU or other devices is an option.

   The LAP's memory address space could be shared or separated from the CPU memory; it depends on the granularity and the complexity of tasks. In case of multiple LAPs, we need to schedule them in a shared memory environment. We avoid cache coherency overheads in our solution.

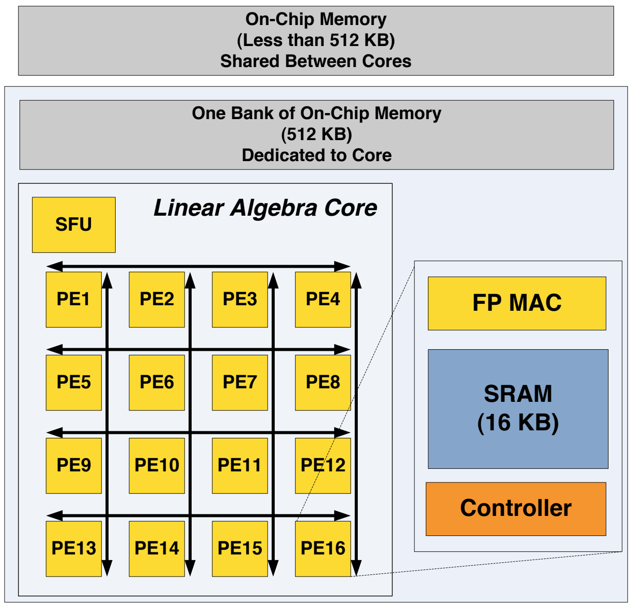


Figure 1.1: A single Linear Algebra Core (LAC) in LAP Architecture, SFU is Special Functional Unit.

1.2.1 Architecture

In contrast to the conventional 1D arrangement of FUs in SIMD architectures, we use row and column buses in a 2D arrangement of PEs as illustrated in Figure 1.1. Each Processing Element (PE) contains a MAC unit, SRAM Local Storage (LS), a microprogrammed controller, and necessary datapath. Each PE is connected to all of the PEs in the same row and in the same column via corresponding row and column broadcast buses, respectively. A Special Functional Unit (SFU) performs special functions such as divide and square-root operations. Cores are connected to the on-chip shared memory and their own dedicated on-chip memory bank and communicate data in and out using column broadcast buses.

We will see in Chapter 3 that this arrangement naturally maps a ma-

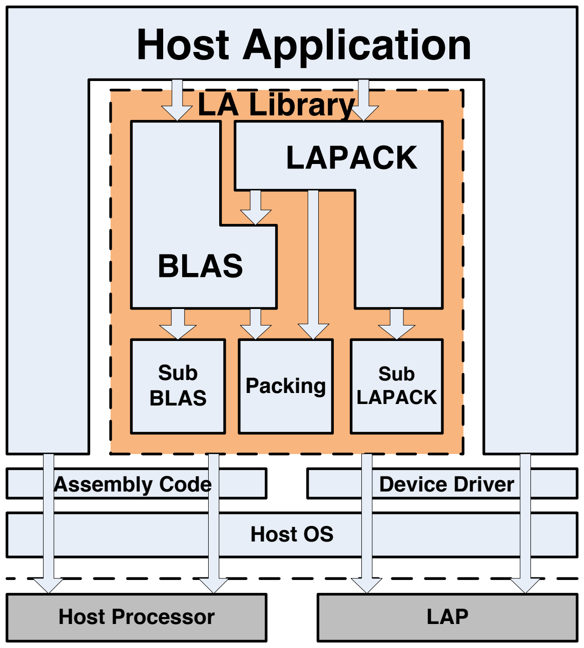


Figure 1.2: LAP programming environment.

trix multiplication kernel using broadcast buses, and it eliminates the need for communication through a register file by fully exploiting the communication network. By distributing the control, we also remove the overheads of control and unnecessary data communication between processing elements. Further, we exploit the concept of separating the memory interface and intra-PE communication interface by streaming the data through a particular channel to the core.

1.2.2 Programming model

Figure 1.2 shows how a linear algebra application can employ a LAP. Linear algebra (LA) libraries, such as the libflame [138, 139] and LAPACK [12] support built-in software layers that decompose big problems into smaller subproblems with so-called algorithms-by-blocks [90]. Routines with higher level functionality (e.g, a LU factorization) are called from the host application. The LA library's internal routines break the large problem recursively into smaller, simpler subroutine calls to Basic Linear Algebra Subroutines (BLAS) and communication & packing routines, until the problems reach a certain size. These small problems (for example 128×128) are atomic units of data with which atomic computations are performed. On a typical general-purpose CPU, these kernels are all implemented very efficiently, often in target machine assembly code [52]. One can view the LAP as an accelerator for these atomic kernel operations and the atomic size of the kernels depends on the LAP-supported kernel sizes. Instead of calling the assembly-coded kernel on the host processor, necessary information including the data location address and type of operation to be performed is passed to the LAP through the device driver. After finishing the operation, the LAP puts the computed data back in the memory. The LAP can overlap the communication with computation by pipelining multiple operations.

1.3 Evaluation Methodology

We have developed both simulation and analytical power and performance models of the LAP in comparison with other architectures. We validated the performance model and LAP operation in general by developing a cycle-accurate LAP simulator. The simulator is configurable in terms of PE pipeline stages, bus latencies, and memory and register file sizes. Furthermore, by plugging in power consumption numbers for MAC units, memories, register files, and buses, our simulator is able to produce an accurate power profile of the overall execution. We accurately modeled the cycle-by-cycle control and data movement for GEneral Matrix-matrix Multiplication (GEMM), TRiangular Solve with Multiple Right-hand Sides (TRSM), and Cholesky factorization, and we verified functional correctness of the produced results. The simulator provides a testbed for future investigation of other linear algebra operations.

1.3.1 Performance Analyses

Linear algebra applications have predictable memory access behavior and the custom-designed LAP architecture does not contain caches or any other processing units with non-deterministic behavior. Therefore, one can model the data movement and access patterns with analytical formulae. We have verified our analytical formulae against our in-house cycle-accumulate simulator for some of the applications. We derived the analytical formulae in two different ways and matched the answers to bolster our confidence in their the correctness. We derived the results first from inside of the core to the next levels of the memory hierarchy as problem size grows, and then from lower levels of memory hierarchy perspective into the core as problem size shrinks.

1.3.2 Component selection

To investigate and demonstrate the performance and power benefits of the LAP, we have studied the feasibility of a LAP implementation in current bulk CMOS technology using publicly available components and their characteristics as published in the literature.

State-of-the-art implementations of Fused Multiply Add (FMA) units use various optimization techniques to reduce latency, area and power consumption [117]. Fused Multiply Accumulate (FMAC) units with delayed normalization achieve a throughput of one accumulation per cycle [141, 142] and save around 15% of total power [63]. The number of pipeline stages typically ranges between 5 and 9 and the same FMAC units can be reconfigured to perform either integer, single-, or double-precision operations [132]. A precise and comprehensive study of different FMA units across a wide range of both current and estimated future implementations, design points and technology nodes was presented in [43].

Our design utilizes SRAM storages with no tags and no associativity. Given the sequential nature of access patterns to 64-bit wide double-precision numbers, we carefully selected memories with one or two banks to minimize power consumption by using CACTI [93] memory simulator. The optimized choice is the low-power ITRS technology model and aggressive interconnect projection.

To estimate latencies and power consumption of row and column buses, we use data reported in CACTI. Since the LAP does not require complex logic for bus arbitration and address decoding, we only consider the power consumption of the bus wires themselves.

For the overall system estimation, we project the dynamic power results reported by CACTI to the target frequencies of the MAC units. According to the CACTI low-power ITRS model, leakage power of the memory blocks is estimated to be negligible in relation to the dynamic power. When more bandwidth is needed for the on-chip memory, the technology changes into a faster model and the leakage power ratio increases.

1.3.3 Power Modeling of Architectures

We developed a general analytical power model that builds on existing component models (e.g. for FPUs and memories) described in the previous section. The model is derived from methods described in [20, 88] and we applied it to both our LAP and various existing architectures. Our power model computes the total power as the sum of the dynamic power and idle power over all components in the architecture:

$$\begin{aligned}
Power &= P_{dyn} + P_{idle} = \sum_{i=1}^{n}(P_{dyn,i}) + \sum_{i=1}^{n}(P_{idle,i}) \\
P_{dyn,i} &= P_{max,i} \times activity_i \\
P_{idle,i} &= P_{max,i} \times ratio.
\end{aligned}$$

Dynamic power is modeled as a maximal component power multiplied by the component's activity factor. We estimate activity of memory components based on access patterns for matrix multiplications. Otherwise, we assume activity factors of one or zero depending on whether a component is utilized during the targeted operations. For leakage and idling, we use a model, derived from calibrations, that estimates idle power as a constant fraction of dynamic power ranging between 25% and 30%, depending on the technology used.

We calibrate our power model and its parameters against power and performance numbers presented for the NVidia GTX280 Tesla GPGPU when performing matrix multiplication [60, 149]. We used the sizes of different GPU memory levels reported in [149] together with numbers from [60] and [4] to match logic-level, FPU, CACTI and leakage parameters and factors in order to achieve consistent results across published work and our model. We then apply this model to other architectures, such as the NVidia GTX480 Fermi GPGPU [2, 68] or the Intel Penryn [47] dual-core processor. To the best of our knowledge, there are no detailed power models yet for these architectures. We adapted our model to the architectural details as far as reported in literature using calibrated numbers for basic components such as scalar logic, FPUs or various memory layers. In all cases, we performed sanity checks to ensure that total power numbers match reported numbers in literature.

1.4 Contributions

The main contributions of this dissertation are as follows:

1. The design, simulation, and power estimation of a highly optimized linear

algebra core for matrix computations.

2. A multi-dimensional design space exploration of a multi-core linear algebra processor for the GEMM algorithm.

3. An analytical tool for evaluating the memory hierarchy size and bandwidth balance for linear algebra operations.

4. A thorough study of the behavior of level-3 BLAS operations across the linear algebra core and conventional SIMD architectures.

5. A generalization of the base architecture to support all level-3 BLAS and important matrix factorizations.

6. The power and performance details for PE and floating-point unit extensions to support special functions for divide and square-root operations.

7. A study of Fast Fourier Transform (FFT) operation in contrast with GEMM operation and the corresponding algorithm/architecture tradeoffs.

8. A design of a Hybrid FFT/Linear Algebra core with minimum loss in efficiency.

Together, these advance the state-of-the-art in this domain.

1.5 Thesis Outline

The rest of this dissertation is organized as follows. In Chapter 2, we present an overview and analysis of the state-of-the-art conventional and custom designed architectures. In Chapter 3, we introduce the linear algebra core (LAC) design, using matrix multiplication as a driving example, and report the estimated area, power, performance, and efficiency of the core for that operation. In Chapter 4, we develop a multi-LAC system and discuss tradeoffs for matrix multiplication at different levels of the memory hierarchy. We demonstrate the potential for flexibility and support of level-3 BLAS kernels on the linear algebra core in Chapter 5. Chapter 6 discusses generalization opportunities by showing how more complicated linear algebra and signal processing algorithms like matrix factorizations and FFT, can be mapped to the LAC. We summarize the dissertation and discuss future goals in Chapter 7.

In Appendix A the details of matrix factorization algorithms are discussed. Appendix B provides the details of the algorithm and mapping for FFT operation on the LAC.

Chapter 2

Related Work

Matrix-matrix multiplication and related kernels are of interest because these operations are often what deliver high-performance to many crucial applications [120]. Key to a successful implementation are insights related to the optimal exploitation of parallelism and locality in general-purpose processors [52], GPUs [133, 143], and other examples of parallel architectures [141]. These insights can have a greater impact if directly applied to the co-design of algorithms and architectures.

Within the domain of linear algebra computations, it is well understood that many problems can be efficiently reduced down to a canonical set of Basic Linear Algebra Subroutines (BLAS), such as matrix-matrix operations (level-3 BLAS) and matrix-vector operations (level-2 BLAS). The response to this has been the definition of interfaces to these key operations [32, 33, 85], and high-performance libraries that are layered upon the BLAS, like the Linear Algebra Package (LAPACK) [12] and, more recently, the libflame library [138, 139]. As a result, the time to solution for the complete application is often heavily dictated by the performance of dense linear algebra operations with relatively small matrices (e.g., of size 100×100 to $10,000 \times 10,000$). If one improves the performance and/or reduces the power consumption of such operations at the node or core level, all applications potentially benefit.

In the following, we will briefly re-examine traditional general-purpose architectures, vector extension architectures, and GPGPUs, with a discussion of their strengths and sources of overhead specifically when performing matrix computations. Next, we focus on accelerator designs and discuss their architectures. This provides the basis for developing our proposed matrix processor architecture aimed at removing such inefficiencies.

2.1 General-Purpose Processors

A general-purpose data path, illustrated in Figure 2.1, executes computation by repeatedly reading operands from storage, performing ALU operations on them, and writing results back to register files. In order to provide flexibility and generality, functional units are typically only provided for basic operators, and every sequence of two or more operations has to go through the register file and interconnect. In many modern general-purpose CPUs, only 15%-25% of the area and power consumption is actually dedicated to Functional Units (FUs) [1, 11]. The rest is spent on aggressive superscalar, out of order execution, and multi-threading techniques to recover instruction-level parallelism out of a serialized instruction stream, and keep the FUs utilized. Furthermore, with unknown sequences of operands, the storage and interconnect has to be effectively designed for random access patterns.

A CPU takes advantage of temporal and spatial locality to reduce demand on the remote slow DRAM. It supports complex memory hierarchies and multiple levels of caches to provide local high bandwidth to the core. Speculative prefetching, and associated bookkeeping and prediction overheads, are often employed to keep the core utilized. In the lowest level of memory hierarchy data access through multi-port register files is expensive and can become the bottleneck [121]. In higher levels of hierarchy, complexity of tag handling and address decoding in caches limits the size available for actual data storage. For example, in large data sets such as matrices the bulk of data is stored in higher levels of the hierarchy. While deep caching allows general-purpose architectures to recover enough locality and hence parallelism to keep their FUs busy, extraneous transfers to bring data in and out from/to far memories consume a large amount of energy, often far more than that used for computing with the data.

The costs for increased single-threaded performance gains have reached the point where old techniques incur tremendous overhead [55] and outweigh the benefits of any further improvements. Even more importantly, technology scaling is also reaching physical limits. Additional transistors will only be provided at reduced performance and increased power consumption [46, 61].

General matrix multiplication (GEMM) implementation on traditional general-purpose architectures has received a lot of attention [3, 8, 51, 118, 146]. However, general instruction handling overhead remains and, even with SIMD instructions, long computations have to be split into multiple operations that exchange data through a wide register file.

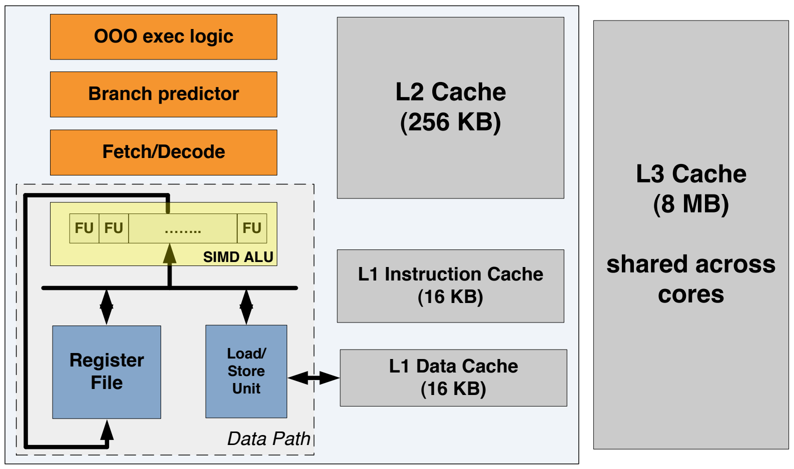


Figure 2.1: (a) A typical general-purpose processor memory hierarchy and core architecture.

2.1.1 SIMD ALUs and Vector Processors

Adding vector units to conventional processors has been a solution to increase efficiency of CPUs [38, 39]. Modern CPUs include Single-Instruction Multiple-Data (SIMD) vector units, such as Intel's Streaming SIMD Extensions (SSE) [119]. In a SIMD solution, the data path contains multiple FUs of the same type that can simultaneously perform a single operation on multiple data items. SIMD processors exploit data parallelism while reducing the number of instructions, which is particularly beneficial for matrix operations. A taxonomy of register file architectures is presented in [121]. Along three main dimensions: data-parallel, instruction level parallel, and memory hierarchy resulting into 12 different organizations, each organization shows different behaviors in terms of area, power, and delay.

Three main limitations of conventional vector architectures are known to be (1) complexity of central register file; (2) implementation difficulties of precise exception handling; and (3) expensive on-chip memory [75]. Although the throughput has increased in these architectures, basic instruction handling overhead still remains and fused operations like multiply-accumulate still have to be performed in multiple instructions that exchange data through a shared register file or, when spilled, through the memory. Associated costs are amplified by the fact that in each step a complete vector has to be transferred through multiple ports of a register file, wide wires, and complex point-to-point interconnects such as crossbars. CODE architecture [75] is designed around a clustered vector register file with decoupled interconnect trying to overcome these inherent limitations.

In recent years several projects were dedicated to evaluation and optimization of vector architectures. Tarantula [37] is an alpha EV8 architecture with vector unit capable of 32 FLOPs per cycle. The vector and multithreaded compute models are unified in the SCALE [76] vector-thread architecture. The vector architectures are compared with conventional superscalar and VLIW architectures for multimedia benchmarks in [74]. Energy-efficiency potentials of vector accelerators for high performance computing systems are discussed in [86]. The efficiency of an architecture depends on the organization of the SIMD units and how they are employed with regard to instruction pipeline and memory hierarchy. In the rest of this section we present examples of the different architectures based on SIMD concept and their different power and performance features.

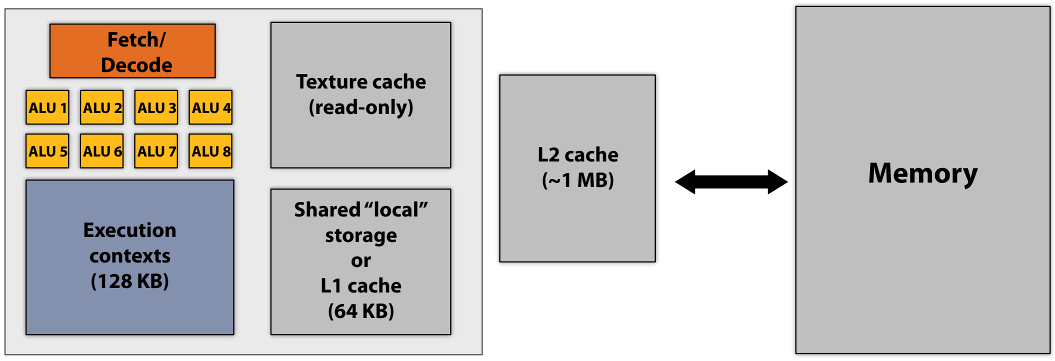

Fetch/
Decode
ALU 1
ALU 2
ALU 3
ALU 4
ALU 5
ALU 6
ALU 7
ALU 8
Execution contexts (128 KB)
Texture cache (read-only)
Shared "local" storage or L1 cache (64 KB)
L2 cache (~1 MB)
Memory

Figure 2.2: Modern GPU core, and memory hierarchy architecture.

2.1.2 GPGPUs

Graphical Processing Units(GPUs) have recently shifted away from specialization to providing general-purpose computing capabilities. While this makes such General-Purpose GPUs (GPGPUs) interesting for a wider class of applications, the added flexibility invariably re-introduces overhead. Apart from some remaining special graphics FUs, GPGPUs essentially replicate a large number of SIMD processors on a single chip. To provide matching locality, SIMD processors are clustered into groups that share common levels of the memory hierarchy. Their pipeline is kept simple with no branch prediction or out-of-order execution but multithreading is used in GPU cores to hide long memory access latency. However, inherent characteristics and deficiencies of a SIMD processor remain.

A typical GPU today, shown in Figure 2.2, has 64Kbytes or more of local storage per core to keep the execution context. A read-only texture cache has been a part of GPUs. Modern GPUs like Nvidia Fermi [2] and

Intel Larabee [125] have recently supported the memory cache hierarchy but their on-chip cache size is relatively small (around 0.5 Mbytes). Although GPUs support high bandwidth DRAM organizations like DDR3-5 with wide 150 Gbytes/sec bus, their bandwidth to computation ratio is still much lower than CPUs. As a result GPUs have to carefully schedule memory requests to efficiently use the available bandwidth.

GPUs were originally developed as specialized hardware for graphics processing that provided massive parallelism but were not a good match for matrix computations because they did not support enough data throughput to computation ratio [40]. In recent years, GPUs have become a popular target for acceleration.shifting back towards general-purpose architectures. Such GPGPUs replicate a large number of Single Instruction Multiple Data (SIMD) processors on a single shared-memory chip. GPGPUs can be effectively used for matrix computations [10, 143] with throughputs of more than 300 GFLOPS for Single-precision GEMM (SGEMM), utilizing around 30-60% of the theoretical peak performance. In the latest GPGPUs, two single-precision units can be configured as one double-precision unit, achieving more than 700 single-precision and 350 double-precision GFLOPS at around 70% utilization [133] for matrices larger than 512×512. Even when ignoring the power consumption of components such as texture caches or special functional units (SFUs), actual GEMM efficiencies in terms of GigaFlops/Watt are an order of magnitude lower than what is inherently possible. Later in this dissertation we will address the causes of this inefficiency.

The main difference between LAP and GPU design lies with the communication through a shared context and instruction handling in the GPU cores. Multithreading to hide memory latencies also adds overhead for GPU cores. In the same manner as in GPGPUs with shared on-chip memory, LAP basic cores can in the future be replicated and dropped into a larger linear algebra processor arrangement.

2.2 Custom Design Architectures and Accelerators

Accelerators are “specialized functional units integrated with the core, specialized cores, attached processors, or attached appliances” [21]. They are tuned to provide low power, low cost, higher performance, and less development, while maximizing throughput per unit area of silicon. An accelerator does not function on its own; it requires invocation from host programs [103, 112]. The strategy is specialization and it cannot be used as a “general-purpose” compute engine. A sustainable accelerator model requires an application domain where “too much performance is never enough” [103]. These domains are open to an accelerator-based solution for which a combination of parallelism, pipelining, and regularity of computation is necessary. “The single-thread performance reduction of Moore’s law makes accelerators economically viable to a degree they have never been before” [112]. Since parallelizing the code is far from trivial in the case of multithreaded solutions, avoiding it by direct hardware implementation may be a major benefit of accelerators.

There are major problems with the development of accelerators in today's technology. By nature, accelerators are separate from the host CPU and as a result data transfer overheads affect performance. There is no standard architecture model and most accelerator design spaces are less mature, fragmented, and highly dynamic. Accelerators are designed to maximize computation throughput, which is often achieved at the expense of ease of programmability. They often have software managed memories and special-purpose, or raw hardware, interfaces. With their narrow applicability, the optimization process for accelerators is a multivariable optimization that includes parallelization, data structure selection, thread granularity, data tiling dimensions, register usage, data prefetch distance, and loop unrolling. These parameters are not necessarily orthogonal to each other [116] .

According to the taxonomy of accelerators and the possible programming models in accelerator systems in [21], the system characterization is affected by two factors: system architecture and workload characteristics. Workload characterization determines parallelism granularity and type of synchronization between accelerator and host. Parallelism granularity affects invocation overhead, CPU and memory coupling, and addressing. The authors in [21] recognize architecture, invocation and completion, and memory addressing as the main dimensions of the design space.

To manage and program the hardware, device drivers and initialization routines are needed even for tightly coupled accelerators. Typically, "libraries are the first and the universal programming model that is developed for any accelerator, and higher-level programming models are often built (and depend internally) on the interfaces provided by libraries" [21]. High-level libraries encapsulate the functionality of an accelerator into an API. Once a set of services is defined, one can change the implementation of the library without the need to change the application using the services. In auto-exploitation like auto vectorization, the compiler and runtime system discover sections of code (instructions or entire procedures) that can be offloaded on an acceleration engine.

GEMM on accelerators with 2D grid of PEs A taxonomy of matrix multiplication algorithms on 2D grids of Processing Elements (PE)s and their interconnect requirements is presented in [87]. The algorithms for matrix multiplication are based on three basic classes: Cannon's algorithms (roll-roll-multiply) [22, 91], , Fox's algorithm (broadcast-roll-multiply) [26, 42, 87], , and SUMMA (broadcast-broadcast-multiply) [6, 137]. Cannon's algorithm shifts the data in two of the three matrices circularly and keeps the third one stationary. Required initial and final alignment of the input matrices needs extra cycles and adds control complexity. In addition, a torus interconnect is needed to avoid data contention. Fox's algorithms and its improvements broadcast one of the matrices to overcome alignment requirements. However, a shift operation is still required and such algorithms may show poor symmetry and sub-optimal performance. Finally, the SUMMA algorithm does not need any initial or post-computation alignment. The broadcast is a simple and

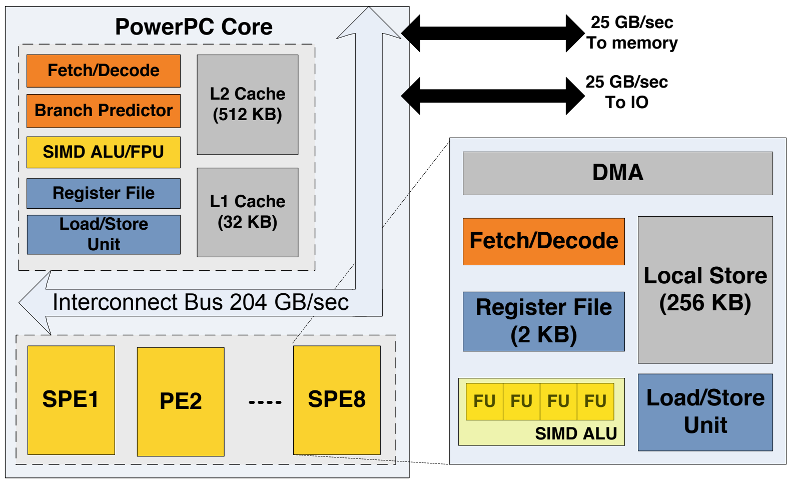


Figure 2.3: (a) Cell BE processor architecture [23].

uniform, single communication primitive, and does not have any bandwidth contention as in circular shifting. In addition, SUMMA is much easier to generalize to non-square meshes of processing units.

The flexibility of the SUMMA algorithm has made it the most practical solution for distributed memory systems [137] and FPGAs [34], and the SUMMA class of algorithms is the basis for our design. A broadcast operation is an efficient way of data movement to achieve high performance in other matrix operations. We will see that the cost and latency of broadcast operation does not add extra overhead in our cores.

2.2.1 Cell Broadband Engine

Cell [66] is a heterogenous multi-core design with Power Architecture compatibility. Three following main objectives were sought in the design of this architecture: power and area efficiency while maintaining programmability, good responsiveness, and wide applicability. Cell was targeted to work at 3.2 GHz frequency in which it can deliver up to 230 GFLOPS single and 19 GFLOPS double precision theoretical peak performance. The third generation of Cell in 45nm technology with 40 Watt estimated power consumption achieves over 5 SP-GFLOPS/Watt power efficiency [131].

The Cell architecture(Figure 2.3), contains a dual-threaded, 4-way in-order 64-bit PowerPC core (PPE), and eight synergetic processing elements (SPEs). These cores are connected to each other through a high bandwidth (204 GB/S) coherent element interconnect bus(EIB [73]). PPE and SPE architectures are both based on SIMD vector unit organization. PPE supports a conventional cache hierarchy and virtualization for multiple operating systems. The SPE architecture [41, 54, 59] is designed to optimize power and performance on media applications as well as compute-intensive applications. SPE is dual issue, coarse grain multi-threaded RISC architecture. SPE does not support hardware branch prediction; its pipeline is kept short to overcome branch miss penalties and reduce area. Instead of a cache a 256-KB Local Store (LS) is employed that allows a large number of memory transactions to be in flight. The SRAM design of the LS eliminates the complexities and latency of caches and also occupies less area on the chip. The LS that holds both instructions and data, is shared between SPE load store unit, instruction fetch unit, and the DMA unit. DMA unit facilitates direct access to main memory with high bandwidth (25 GB/S). A large (2 KB) 128-entry 8-ported register file provides data for the SIMD ALU. SIMD unit can perform four single precision multiply-accumulate operations in each cycle.

The ground breaking techniques and rational described in [59], were used in the Cell architecture to improve power and area efficiency. However, SPEs still support conventional pipeline overheads. A big, multi-ported register file is a huge bottleneck here. All SPEs might end up executing the same code over and over wasting power, while each is only 4-way SIMD. The local store is software controlled resulting in more energy consumption, becaus more instructions must be executed to manage it. At the chip level keeping the interconnect network coherent adds to complexity and energy usage. These overheads are paid to make this architecture more flexible.

Implementations of scientific applications have been targeted on the Cell processor by many works in the literature [23, 148]. Cell can reach over 200 GFLOPS, 90% of theoretical peak performance, for single precision matrix multiplication problems [23, 80, 84, 122, 148]. Level 1-3 BLAS [122], Cholesky factorization [80], QR factorization [81], LINPACK benchmark [23], and sparse vector matrix multiplication [148] achieve high performance on this architecture. Other scientific kernels like 1-D and 2-D FFT are also mapped on the Cell processor [14, 53, 148]. Cell achieves 5 GFLOPS/W for linear algebra kernels that is an order of magnitude less than what is possible.

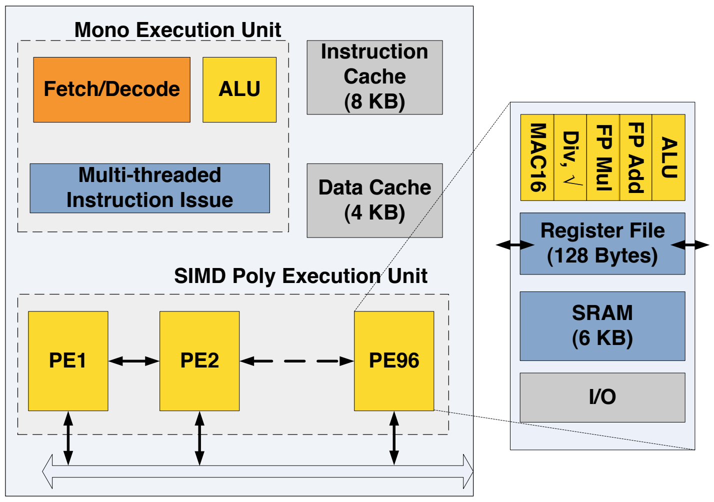


Figure 2.4: Clearspeed Multi-threaded array processor architecture [28].

2.2.2 ClearSpeed CSX

CSX architecture [28] is the computation core of ClearSpeed CSX600[95, 97] and CSX700 [5] processors. ClearSpeed CSX700 is well known as the cutting edge accelerator that targets scientific computing and provides BLAS and LAPACK library facilities with double precision. This chip delivers up to 96 GFLOPS theoretical peak for just 12 watts power consumption (8 DGFLOPS/Watt power efficiency) at 250 MHz frequency in 90nm technology.

ClearSpeed CSX is a SIMD architecture with long 96 PE dimension similar to vector architectures. The major difference is that the data streaming can be done independent of the control path (similar to SPEs in Cell [66]). Each PE is a VLIW core with a complete pointer model that results in inherent overheads. This 1D long arrangement of PEs has the problem of communication between PEs. By supporting up to 8 prioritized multithreaded execution, the long 1D array of PEs can be broken into smaller groups. Although multithreading helps hiding memory access latencies to 128 KB on chip scratch pad memory and 2Gbyte DDR2 external memory, it adds overhead to the hardware. CSX PEs do not contain a fused multiply accumulate unit, hence they have to pay the overheads of performing two instructions for multiplication and addition. The computational units are connected to a five ported 128-byte register file that is closely coupled with a 6KByte SRAM local store. All data access to local SRAM and even communication to adjacent PEs are through the register file that could become a bottleneck in design. PEs can communicate with each other through the "Clearconnect" network. In [96] authors demonstrate that sending and receiving overheads at each core are the bottleneck of this architecture. They suggest that evaluation of other topologies like 2-D mesh for future SIMD interconnects can affect the performance of such architectures.

75 GFlops for double precision matrix multiplication with 78% of theoretical peak performance is achieved on this architecture [5]. Scientific applications like FFT [5], singular value decomposition, and QR factorization [152] have been mapped on this accelerator as well. This architecture has high power efficiency but low performance and area efficiency. The frequency of the chip is kept low because the memory cannot sustain the bandwidth demands of PEs in high frequencies.

In contrast with Clearspeed CSX architecture, LAP design has a micro-programmed distributed control. The data movement and computation order is microcoded in the local controller of the PEs. Caches and instructions are excluded from the LAP design. LAP supports specialized fused MAC units with throughput of one, which helps eliminate complexities of data/instruction handling through a register file. The LAP design is a 2D arrangement with broadcast buses, which benefits from a simple control. There is no send and receive or any acknowledgement involved in this communication mechanism, which is the main communication overhead in the ClearSpeed architecture. Finally, the register file used in LAP PEs is 32 bytes with only two ports and is bypassed in most of the data transfers to significantly reduce power consumption.

2.2.3 Systolic Arrays

Systolic arrays were popularized in the 80s [78, 79]. Different optimizations and algorithms for matrix multiplication and more complicated matrix computations are compared and implemented on both 1D [104, 115, 134] and 2D systolic arrays [62, 89, 134]. In [65], the concept of a general systolic array and a taxonomy of systolic array designs is presented.

Systolic arrays are usually designed as a 2D array of processing elements, where each PE shares its processed data with its adjacent neighbor PEs immediately in the next cycle. The data flows in the pipe network across the array, often with different data flowing in different directions in a pipeline fashion. The PEs do not hold more than a few pieces of storage. Their inefficiency is in their complex design and difficulties in building them.

The LAP core design has several similarities and differences with systolic arrays. Both designs use the same 2D arrangement of PEs: PEs have simple functional units and both perform very well for applications like matrix multiplication. PEs in a LAP core are sitting on a shared bus and there is no data streaming flow in them. Each PE in a LAP core has a relatively simple but large local store. The communication is done in a broadcast fashion across rows and columns. There are no data dependencies between the computed values by adjacent neighboring PEs for operations like GEMM. In other words, the PEs do not pass processed data to their neighbors in the LAP design.

A LAP core supersedes systolic solutions by decreasing unnecessary communication between the PEs and performing inner products within each PE. This way, operations on elements of matrix C have register/accumulator level access locality, and no extra transaction in the PE data-path or between PEs is required. Performing the inner product operations locally allows optimization for single-cycle accumulation in the MAC units and hence, as we will discuss later, saving power by using a wide accumulator register to avoid unnecessary consecutive normalizations.

The broadcast bus nature of communication avoids pipelining the input data, reduces the data read transactions and register accesses. This way the interface of each PE to the other neighbors requires much simpler logic. Later, we will see that a broadcast bus solution saves cycles in the inner kernels of complicated operations where a whole row or column is dependent on a result of a certain PE. The length of the critical path decreases since the critical data arrives at the same time to all the PEs in the same row or column.

2.2.4 FPGA Implementation

Moving toward TeraFlops peak performance, recent FPGAs [45, 101] have achieved high standards both in performance and power efficiency. Recent designs have provided floating point logic blocks along with fixed point multipliers and adders, and fused data facilities in their toolchain [101]. Given the potential of FPGAs there is more motivation to use FPGAs as accelerators next to the processor with hardcoded functional units. Complex functional units that might be used frequently and do not achieve efficiency or performance on the processors could be hardcoded in the FPGAs.

Some of the drawbacks of using FPGAs are that they typically have much higher power dissipation compared to an ASIC implementation of the same logic, they are un-programmed at power up and need a PROM or host to store an image of the hardware program. FPGAs offer limited logic capacity on the chip and with slow clock frequency (100-300 MHz) FPGAs can reach high GFLOPs/Watt, but their peak performance is then limited. According to FPGA vendors like Altera/Xilinx, an FPGA with 40nm technology can achieve at most 100 DP-GFLOPS performance at 7 GFLOPs/Watt of power efficiency [100].

Specialized hardware implementations of GEMM on FPGAs have been explored before, either as dedicated hardware implementation [155, 156] or in combination with a flexible host architecture [82]. Such approaches show promising results (up to 99% utilization) but are limited by the performance and size restrictions in FPGAs. Matrix multiplication on Stratix III with up to 50 GFLOPS [83], on Xilinx vertex II FPGA with up to 15.6 GFLOPS [34], and on Virtex-5 SX240T with up to 30 GFLOPs [77] are some of the many implementations in the literature. Performance and energy efficiency of FPGA implementation of matrix multiplication with DSPs and embedded processors has been compared in [124]. New algorithms and architectures [64] offer trade-offs among the number of I/O ports, the number of registers, and the number of PEs to significantly reduce the energy dissipation and latency. However, with the flexibility of being able to implement various algorithms directly in hardware comes an inherent overhead for a general, reconfigurable hardware fabric.

Chapter 3

Linear Algebra Core (LAC) Design

Our design methodology starts by focusing on the inner kernel of general matrix-matrix multiplication (GEMM). Most dense linear algebra algorithms can be cast to spend most computations in GEMM. With its high ratio of computation to data motion and its balanced use of addition and multiplication, GEMM provides the opportunity to attain near peak sustainable floating-point computation rates for a given computer system. The lessons learned from optimizing the design for GEMM are crucial and fundamental for other important linear algebra operations. We start in a bottom-up fashion with algorithm/architecture co-design of a linear algebra core and study its memory hierarchy tradeoffs. We fine tune the core design for efficiency and performance. We design our engine with an outlook of supporting algorithms beyond GEMM.

A high-level design for a *Linear Algebra Core* (LAC) is shown in Figure 3.1. It consists of a 2D array of $n_r \times n_r$ processing elements (PEs), each of which has a MAC unit with a local accumulator, local storage, simple distributed control, and bus interfaces to communicate data within rows and columns. For illustrative purposes we will focus our discussion on the case of

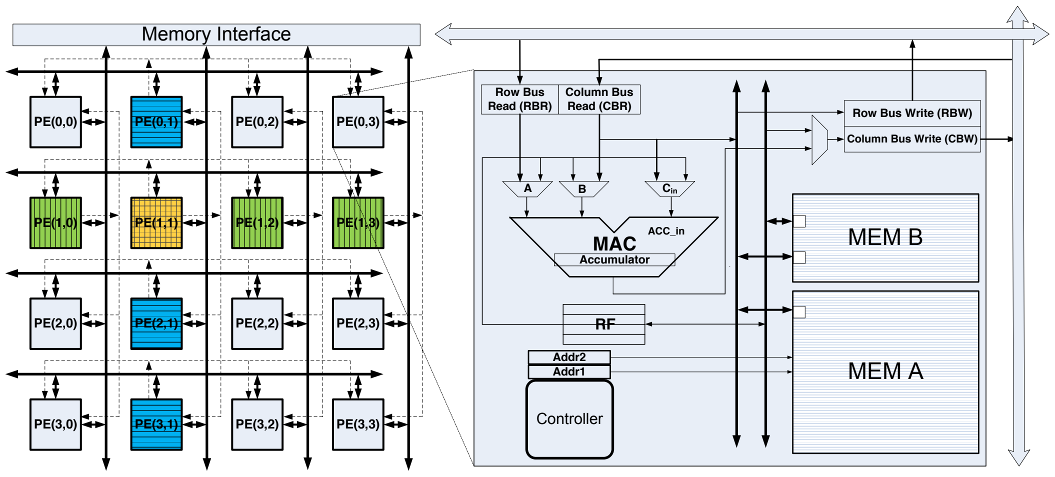


Figure 3.1: The LAC Architecture. The highlighted PEs on the left illustrate the PEs that own the current column of $4 \times k_c$ matrix A and the current row of $k_c \times 4$ matrix B for the second rank-1 update ($p = 1$). It is illustrated how the roots (the PEs in second columns and row) write elements of A and B to the buses and the other PEs read these. The dashed lines show the current data movement on the buses.

a mesh with $n_r \times n_r = 4 \times 4$ PEs.

3.1 Basic Operation

A special case of GEMM will be used in this section to describe the Linear Algebra Processor: Let C, A, and B be 4×4, $4 \times k_c$, and $k_c \times 4$ matrices, respectively[1]. Then C += AB can be computed as a "block dot product" illustrated by Figure 3.2.

[1]The choice of parameter labels like n_r and k_c mirrors those used in [52].

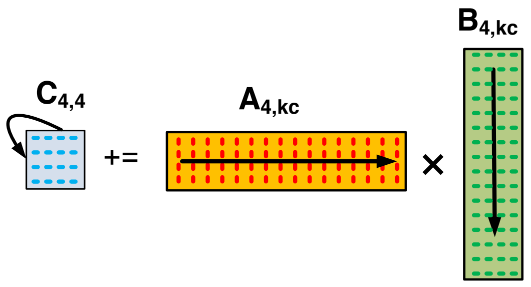


Figure 3.2: Matrix multiplication as a series of Rank-1 updates.

$$\begin{pmatrix} \gamma_{0,0} \cdots \gamma_{0,3} \\ \vdots \quad \ddots \quad \vdots \\ \gamma_{3,0} \cdots \gamma_{3,3} \end{pmatrix} \mathrel{+}= \begin{pmatrix} \alpha_{0,0} \\ \vdots \\ \alpha_{3,0} \end{pmatrix} \left(\beta_{0,0} \cdots \beta_{0,3}\right) + \begin{pmatrix} \alpha_{0,1} \\ \vdots \\ \alpha_{3,1} \end{pmatrix} \left(\beta_{1,0} \cdots \beta_{1,3}\right) + \cdots$$

so that C is updated in the ith iteration with

$$\begin{pmatrix} \gamma_{0,0} + \alpha_{0,i}\beta_{i,0} & \cdots & \gamma_{0,3} + \alpha_{0,i}\beta_{i,3} \\ \vdots & \ddots & \vdots \\ \gamma_{3,0} + \alpha_{3,i}\beta_{i,0} & \cdots & \gamma_{3,3} + \alpha_{3,i}\beta_{i,3} \end{pmatrix}. \tag{3.1}$$

Each such update is known as a rank-1 update. In our discussions, upper case letters denote (sub)matrices while Greek lower case letters denote scalars.

Let us assume that $4\times k_c$ matrix A and $k_c\times 4$ matrix B are distributed to the array in a 2D cyclic round-robin fashion, much like one distributes matrices on distributed memory architectures [25, 57]. In other words, $\alpha_{i,j}$ and $\beta_{i,j}$ are assigned to PE $(i \mod 4, j \mod 4)$. Also, element $\gamma_{i,j}$ of matrix C is assumed to reside in an accumulator of PE (i, j). A simple algorithm for performing this special case of GEMM among the PEs is to, for $p = 0, \ldots, k_c - 1$, broadcast the pth column of A within PE rows, the pth row of B within PE columns, after which a local MAC operation on each PE updates the local element of C.

3.2 PE Micro-Architecture

The prototypical rank-1 update given in Equation 3.1 gives a clear indication of possible parallelism: all updates to elements of C can be performed in parallel. Elements of C are repeatedly updated by a multiply-add operation. This suggests a natural top-level design for a processor performing repeated rank-1 updates as a 2D mesh of PEs, depicted in Figure 3.1 (left). Each PE (i, j) will update element $\gamma_{i,j}$.

Details of the PE-internal architecture are shown in Figure 3.1 (right). At the core of each PE is a MAC unit to perform the computations $\gamma_{i,j} \mathrel{+}= \alpha_{i,p}\beta_{p,j}$. Each MAC unit has a local accumulator register that holds the intermediate and final values of one inner dot product of the result matrix C being updated. Apart from preloading accumulators with initial values of γ, all accesses to elements of C are performed directly inside the MAC units, avoiding the need for any register file or memory accesses. We utilize pipelined units that can achieve a throughput of one MAC operation per cycle. Such throughputs can be achieved by postponing normalization of results until the last accumulation [142]. Being able to leverage a fused MAC unit with delayed normalization significantly decreases power consumption while increasing precision.

As outlined in Section 3.1, we store the $4 \times k_c$ matrix A and the $k_c \times 4$

matrix B distributed among the PEs in local memories. It is well-understood for dense matrix operations [25, 57] that communication is greatly simplified and its cost is reduced if it is arranged to be only within PE rows and columns. When considering $\gamma_{i,j}$ += $\alpha_{i,p}\beta_{p,j}$, one notes that if $\alpha_{i,p}$ is stored in the same PE row as $\gamma_{i,j}$, it only needs to be communicated within that row. Similarly, if $\beta_{p,j}$ is stored in the same column as $\gamma_{i,j}$, it only needs to be communicated within that PE column. This naturally leads to the choice of a 2D round-robin assignment of elements, where $\alpha_{i,p}$ is assigned to PE $(i, p \mod n_r)$ and $\beta_{p,j}$ to PE $(p \mod n_r, j)$.

Each rank-1 update (fixed p, Eqn. 3.1) then requires simultaneous broadcasts of elements $\alpha_{i,p}$ from PE $(i, p \mod n_r)$ within PE rows and of elements $\beta_{p,j}$ from PE $(p \mod n_r, j)$ within PE columns. This is illustrated for the $p = 1$ update in Figure 3.1. In our design, we connect PEs by horizontal and vertical broadcast buses. Interconnect is realized in the form of simple, data-only buses that do not require overhead for address decoding or complex control. PEs are connected to horizontal and vertical data wires via separate read and write latches. This allows for simultaneous one-cycle broadcast of two elements, $\alpha_{i,p}$ and $\beta_{p,j}$, to all PEs in the same row and column.

3.2.1 LAC Communication

The simple, symmetric and regular 2D mesh is scalable and easy to route during physical design and layout. However, the number of PEs determines the length and capacitive load of data buses. As such, wire delays put limits on the possible size, n_r, of a LAP array that can perform one-cycle broadcasts. In this case, buses can be pipelined and latencies are hidden by overlapping with successive computations in the pipelined MAC units. This makes the design reminiscent of a systolic array, with the major difference being that we locally store inputs and results. Hence, we only pipeline a subset of input data but no intermediate results through the array.

Column buses in the PE mesh are multiplexed to perform column broadcasts and also transfer elements of A, B and C to/from external memory during initial preloading of input data and writing back of results at the end of computation. For the latter purpose, PEs can internally read and write column bus values from/to the MAC accumulator or local memory. In regular operation, row and column buses carry $\alpha_{i,p}$ and $\beta_{p,j}$ values that continuously drive PE-internal MAC inputs in a pipelined fashion. Sending PEs $(i, p \mod n_r)$ and $(p \mod n_r, j)$ drive the buses in each row and column with values out of their local memories, where diagonal PEs $(i = j)$ simultaneously load two values from local memory onto both buses. For simplicity and regularity, sending PEs receive their own broadcasted values back over the buses into the MAC inputs like all other PEs. In such a setup, no additional registers or control are necessary.

Alternatively, we can consider a setup in which all elements $\beta_{p,j}$, $p = 0, \ldots, k_c - 1$ of B are replicated among all PEs in each row j. This eliminates the need to broadcast these values across columns. Instead, elements of B

are always accessed locally through an additional register file[2]. Trading off storage for communication requirements, this setup avoids all column transfers, freeing up column buses for prefetching of subsequent input data in parallel to performing computations (see Section 3.3).

3.2.2 Local Store

Overall, the local storage in each PE consists of a larger single-ported and a smaller dual-ported memory to store elements of matrix A and B respectively. A small register file with one write and two read ports is considered to store temporary values. Access patterns are predictable and in most cases sequential. As such, only simple, auto-incrementing address generators are required. Furthermore, memories can be efficiently banked to increase bandwidth and reduce power. All combined, the data path is regular and simple without any overhead associated with tags, large multiplexers or complex address computations to support random accesses.

We note that the described approach, where essentially $n_r \times n_r$ inner products update a $n_r \times n_r$ submatrix of C, adds the benefit of saving power in MAC units by keeping elements of C in accumulator as long as possible and performing normalization rarely.

[2]We include a small, general register file that carries little additional overhead but provides the flexibility of storing a number of intermediate values that can be (re)used as MAC inputs and can be read or written from/to local memory. This will be beneficial in supporting other linear algebra operations in the future.

3.2.3 Control

LAC control is distributed and each PE has a state machine that drives a predetermined sequence of communication, storage and computation operations. Local controllers in each PE are equally smart and all agents operate in parallel and in lock step. PE executions are implicitly coordinated and synchronized without any additional handshaking. Instead, inter- and intra-PE data movement is predetermined, and each PE implicitly knows when and where to communicate. Global control and handshaking is limited to coarse-grain coordination for simultaneous triggering or stalling of all PEs at the start of operation or in combination with external memory accesses. State machines are microprogrammed via a few external control bits to select the type of linear algebra operation that the PE should perform. Using only these control signals and counter presets, we expect to be able to support the full flexibility we want for executing, for example, all level-3 BLAS (matrix-matrix operations) [32].

The basic state machine in each PE requires eight states, two address registers and one loop counter. In the following sections, we will discuss LAP and PE operation for bigger matrix multiplications that are broken into a sequence of basic rank-k updates using a hierarchical blocking of input matrices. Each additional level of blocking will require an additional loop and loop counter. Since there are no loop-carried dependencies, we pipeline the outer loops to effectively overlap the rank-k computation of the current kernel with prefetching of the next kernel's input data and writeback of the previous

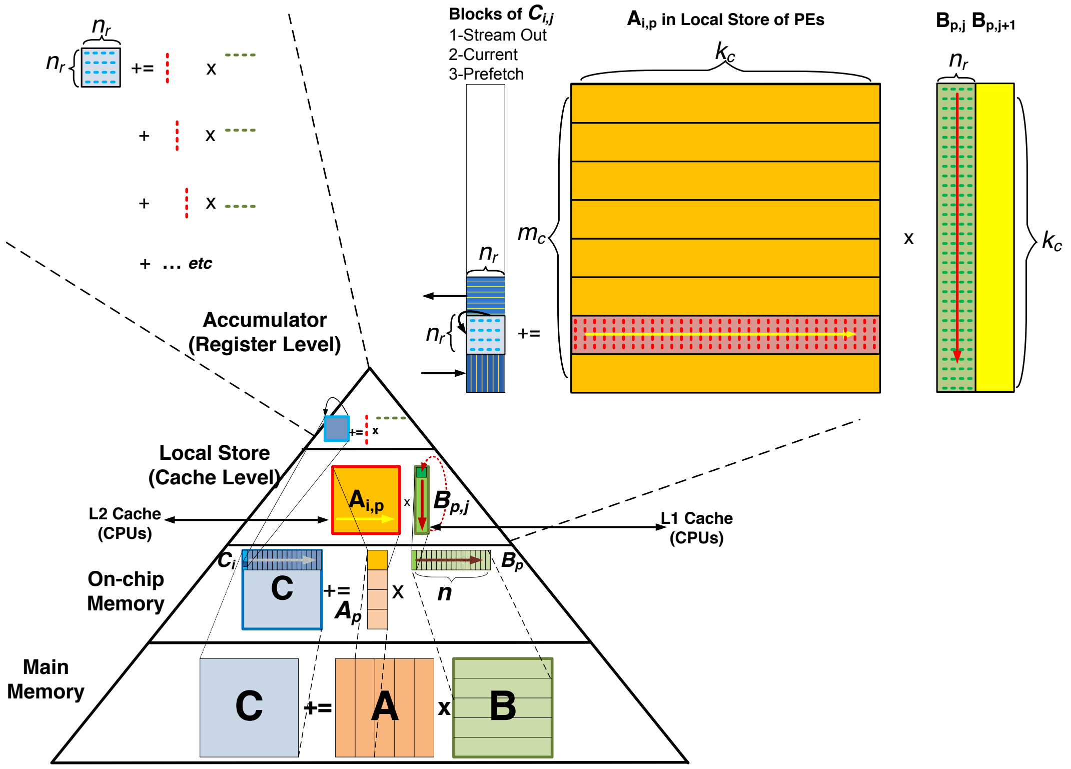


Figure 3.3: Memory hierarchy while doing GEMM. In each of the top three layers of the pyramid, the largest matrix is resident, while the other matrices are streamed from the next layer down.

kernel's results. With B replicated and all of a larger A in local store, the resulting state machine has a combined inner core state that runs all operations in a single-cycle loop with full parallelism and 100% sustained LAP utilization. With three levels of blocking, such PE control only requires a total of four counters and ten states.

3.3 GEMM Algorithm

In designing a complete Linear Algebra Processor (LAP), we not only need to optimize the core, but also describe how data can move and how computation can be blocked to take advantage of multiple layers of memory. In order to analyze the efficiency attained by the core itself, we first need to describe the multiple layers of blocking that are required. We do so with the aid of Figure 3.3. For now it suffices to think of the LAP as consisting of one of the described cores plus on-chip memory. Later, we will generalize this to one with multiple cores.

Assume the matrices A, B and C are stored in memory external to the LAP. We can observe that $C \mathrel{+}= AB$ can be broken down into a sequence of smaller matrix multiplications (rank-k updates with $k = k_c$ in our discussion):

$$C \mathrel{+}= \begin{pmatrix} A_0 & \cdots & A_{K-1} \end{pmatrix} \begin{pmatrix} B_0 \\ \vdots \\ B_{K-1} \end{pmatrix} = \sum_{i=0}^{K-1} A_i B_i$$

so that the main operation to be mapped to the LAP becomes $C \mathrel{+}= A_p B_p$. This partitioning of matrices is depicted in the bottom layer in Figure 3.3.

In the next higher layer (third from the top), we then focus on a single update $C \mathrel{+}= A_p B_p$. If one partitions

$$C = \begin{pmatrix} C_0 \\ \vdots \\ C_{M-1} \end{pmatrix} \text{,and } A_p = \begin{pmatrix} A_{0,p} \\ \vdots \\ A_{M-1,p} \end{pmatrix},$$

then each panel of C, C_i, must be updated by $C_i \mathrel{+}= A_{i,p} B_p$ to compute $C \mathrel{+}= A_p B_p$.

Let us further look at a typical $C_i \mathrel{+}= A_{i,p}B_p$. At this point, the $m_c \times k_c$ block $A_{i,p}$ is loaded into the local memories of the PEs using the previously described 2D round-robin distribution. Partition C_i and B_p into panels of $n_r(= 4)$ columns:

$$C_i = \begin{pmatrix} C_{i,0} & \cdots & C_{i,N-1} \end{pmatrix} \text{ and } B_p = \begin{pmatrix} B_{p,0} & \cdots & B_{p,N-1} \end{pmatrix}.$$

Now $C_i \mathrel{+}= A_{i,p}B_p$ requires the update $C_{i,j} \mathrel{+}= A_{i,p}B_{p,j}$ for all j. For each j, $B_{p,j}$ is loaded into the local memories of the PEs in a replicated column-wise fashion. The computation to be performed is described by the second layer (from the top) of the pyramid, which is also magnified to its right.

Finally, $A_{i,p}$ is partitioned into panels of four rows and $C_{i,j}$ into squares of 4×4, which are processed from top to bottom in a blocked, row-wise fashion across i. The multiplication of each row panel of $A_{i,p}$ with $B_{p,j}$ to update the 4×4 block of $C_{i,j}$ is accomplished by the individual cores via the rank-1 updates described in Section 3.1. What is still required is for the 4×4 blocks $C_{i,j}$ to be brought in from main memory.

This blocking of the matrices facilitates reuse of data, which reduces the need for high bandwidth between the memory banks of the PEs, the on-chip LAP memory and the LAP-external storage: (1) fetching of a 4×4 block $C_{i,j}$ is amortized over $4 \times 4 \times k_c$ MAC operations (4×4 of which can be performed simultaneously); (2) fetching of a $k_c \times 4$ block $B_{p,j}$ is amortized over $m_c \times 4 \times k_c$ MAC operations; and (3) fetching of a $m_c \times k_c$ block $A_{i,p}$ is amortized over $m_c \times n \times k_c$ MAC operations.

This approach is very similar to how GEMM is mapped to a general-purpose architecture [52, 140]. There, $A_{i,p}$ is stored in the L2 cache, $B_{p,j}$ is kept in the L1 cache, and the equivalent of the 4×4 block of C is kept in registers. The explanation shows that there is symmetry in the problem: one could have exchanged the roles of A_p and B_p, leading to an alternative, but very similar, approach. Note that the description is not yet complete, since it assumes that, for example, C fits in the on-chip memory. Even larger matrices can be accommodated by adding additional layers of blocking, as will be described later (see Section 4.2.3).

3.4 Core Architecture

With an understanding of LAC operation, the basic core design, and how matrix multiplication can be blocked, we can now investigate specific core implementations including tradeoffs between the size of the local store and the bandwidth between the on-chip memory and the core (we will consider external memory later). In our subsequent discussion, 4×4, the size of the submatrices of C, is generalized to $n_r \times n_r$. Furthermore, in accordance with the blocking at the upper memory levels, we assume that each core locally stores a larger $m_c \times k_c$ block of $A_{i,p}$, a $n_r \times n_r$ subblock of $C_{i,j}$ and a $k_c \times n_r$ panel of $B_{p,j}$ (replicated across PEs).

The local memory requirements for the core are that matrices $A_{i,p}$ and $B_{p,j}$ must be stored in the aggregate memories of the PEs. To avoid power and area waste of a dual ported SRAM, we decided to separate the local stores for $A_{i,p}$ and $B_{p,j}$ in the PEs. A single ported SRAM keeps elements of $A_{i,p}$ with one access every n_r cycles. Since the size of $B_{p,j}$ is small, we can keep copies of B in all PEs of the same column. This avoids extra column bus transactions and allows overlapping of computation with data movement in and out of the core. As a result, the second SRAM is dual ported and is much smaller compared to the first one. In each cycle, an element of B is read from this SRAM to feed the local MAC unit in each PE. This strategy reduces the aggregate local store size and power consumption in each PE.

The goal is to overlap computation of the current submatrix of $C_{i,j}$ with the prefetching of the next such submatrix. This setup can achieve over 90% of peak performance when k_c is sufficiently large. Thus, the size of the local store, aggregated over all PEs, is given by $m_c \times k_c$ elements for $A_{i,p}$, and by $2 \times k_c \times n_r \times n_r$ elements for the current and next $B_{p,j}$ and $B_{p+1,j}$. In total, the local memory must be able to hold $m_c k_c + 2k_c n_r^2 = (m_c + 2n_r^2)k_c$ single or double precision floating point numbers. Note that the $n_r \times n_r$ submatrix of $C_{i,j}$ is always in the accumulators and never stored. However, concurrent prefetching and streaming out of the next and previous such submatrix, respectively, occupies two additional entries in the register file of each PE. Together with a register each for internal transfers of locally replicated $\beta_{p,j}$, every PE requires a register file of size 4 (a size of 3, rounded up to the next power of two).

To analyze performance, let us assume an effective bandwidth of x elements/cycle and focus on one computation $C_i \mathrel{+}= A_{i,p}B_p$. Reading $A_{i,p}$ requires $m_c k_c / x$ cycles. Reading and writing the elements of C_i and reading the elements of B_p requires $(2m_c n + k_c n)/x$ cycles. Finally, computing C_i += $A_{i,p}B_p$ assuming peak performance requires $(m_c k_c n)/n_r^2$ cycles. Overlapping the communication of C_i and B_p with the computation of C_i gives us an estimate for computing C_i += $A_{i,p}B_p$ of

$$\frac{m_c k_c}{x} + \max\left(\frac{(2m_c + k_c)n}{x}, \frac{m_c n k_c}{n_r^2}\right) \text{ cycles.}$$

Given that at theoretical peak this computation would take $(m_c k_c n)/n_r^2$ cycles, the attained core utilization can easily be estimated as the fraction of the two. Notice that the complete computation C += AB requires loops around this "inner kernel" for one C_i. Thus, it is this kernel that dictates the performance of the overall matrix multiplication.

To achieve peak performance, the prefetching of the next block of A, $A_{i,p+1}$ should also be overlapped with the computations using the current block of $A_{i,p}$ resulting in full overlapping of communications with computation. In such a scenario, each PE requires a bigger local memory for storing the current and prefetching of the next block of A. Thus, the size of the local store, aggregated over all PEs, will become $2m_c k_c + 2k_c n_r^2 = 2(m_c + n_r^2)k_c$. This extra memory is effective if there is enough bandwidth to bring data to the cores.

3.5 Core-Level Exploration

Figure 3.4 reports performance of a single core as a function of the size of the local memory and the bandwidth to the on-chip memory. Here

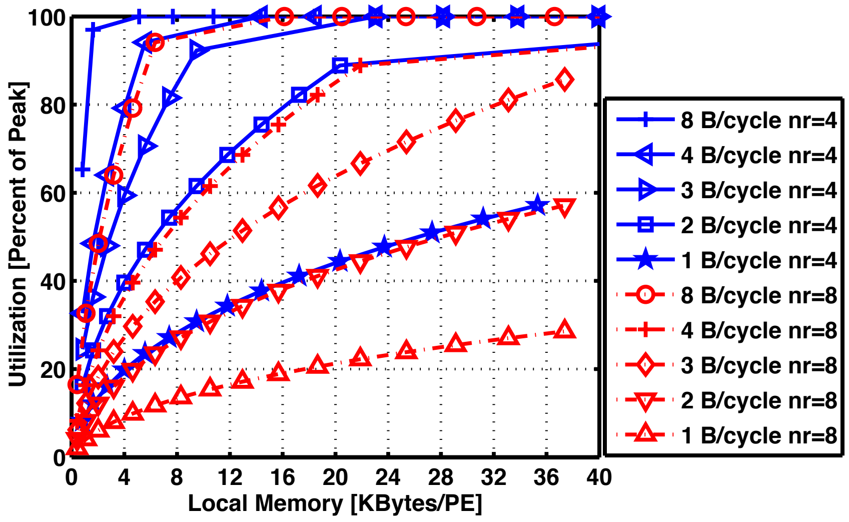


Figure 3.4: Estimated core performance as a function of the bandwidth between LAC and on-chip memory, and the size of local memory with $n_r = 4$ and $n_r = 8$, $m_c = k_c$, and $n = 512$.

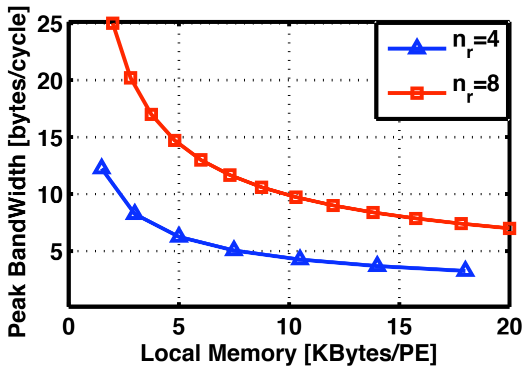


Figure 3.5: Core Performance vs. bandwidth between LAC and on-chip memory for peak performance with $n_r = 4$ and $n_r = 8$, $m_c = k_c$, and $n = 512$.

we use $n_r \in \{4, 8\}$, $m_c = k_c$ (the submatrix $A_{i,p}$ is square), and $n = 512$ (which is relatively small). This graph clearly shows that a trade-off can be made between bandwidth and the size of the local memory, which in itself is a function of the kernel size (k_c, m_c, and n_r).The graph also shows under what conditions we can achieve 100% utilization.

The tradeoff between the needed bandwidth per core and local store per PE is shown in Figure 3.5. The curve shows the relation between the bandwidth and local store size needed to maintain peak performance. It (and the equation that generated it) shows that by doubling the dimension, n_r, while fixing the local store size, the bandwidth demand doubles and performance quadruples. This suggests that making n_r as large as possible is more efficient. However, n_r cannot grow arbitrarily: (1) when n_r becomes too large, the intra-core broadcast require repeaters, which adds overhead; (2) exploiting task-level parallelism and achieving high utilization is easier with a larger number of smaller cores; and (3) with our choice of $n_r = 4$, the number of MAC units in each core is comparable to modern GPUs, allowing us to more easily provide a fair comparison.

3.6 Power Analysis

To investigate and demonstrate the performance and power benefits of the LAP, we have studied the feasibility of a LAP implementation in current bulk CMOS technology using publicly available components and their characteristics as published in the literature.

For our analysis, we use area and performance data reported in [43]. We estimate that a single- and double-precision FMAC unit occupies an area of $0.01mm^2$ and $0.04mm^2$, respectively. Furthermore, all recent literature reports similar power consumption estimates of around 8-10mW and 40-50mW (at $\approx$ 1GHz and 0.8V operation), respectively.

Using CACTI [93] with low-power ITRS models and aggressive interconnect projection, we obtained area estimates of around $0.13mm^2$ and we calculated the dynamic power of the local SRAM at frequencies over 2.5 GHz to be around 13.5mW per port. For the overall system estimation (see Section 4.5), we project the dynamic power results reported by CACTI to the target frequencies of the MAC units. According to the CACTI with low-power ITRS setting, leakage power is estimated to be negligible in relation to the dynamic power.

With a $n_r \times n_r$ 2D array of PEs, our design contains a total of $2 \times n_r$ 32-bit (single precision) or 64-bit (double-precision) row and column buses. However, per PE we only have $2/n_r$ of the power consumption of a single bus. CACTI reports three different classes of wires (fast local, semi-global, and global) for different layers of the memory hierarchy. For intra-core communication, we assume fast local wires. For wires with 30% overhead, the distance between repeaters is a maximum of more than 1.62mm. The delay optimal wire has the shortest latency but consumes much more power due to closer and bigger repeaters compared to slower and less power hungry wires like wire with 30% latency overhead. The 30% latency overhead wire on the other hand

	Speed [GHz]	Area [mm^2]	Memory [mW]	FMAC [mW]	PE [mW]	PE [W/mm^2]	PE [GFLOP/ mm^2]	PE [GFLOP/W]	PE [GFLOP^2/W]
SP	2.08	0.148	15.22	32.3	47.5	0.331	28.12	84.8	352.7
	1.32	0.146	9.66	13.4	23.1	0.168	18.07	107.5	283.8
	0.98	0.144	7.17	8.7	15.9	0.120	13.56	113.0	221.5
	0.50	0.144	3.66	3.3	7.0	0.059	6.94	117.9	117.9
DP	1.81	0.181	13.25	105.5	118.7	0.670	19.92	29.7	107.5
	0.95	0.174	6.95	31.0	38.0	0.235	10.92	46.4	88.2
	0.33	0.167	2.41	6.0	8.4	0.068	3.95	57.8	38.1
	0.20	0.169	1.46	3.4	4.8	0.046	2.37	51.1	20.4

Table 3.1: 45nm scaled performance and area for a LAP PE with 16KBytes of dual-ported SRAM.

is 30% slower but consumes much less power and has longer distance between its repeaters. According to our area estimates, each PE will not be wider than 0.4 mm. Hence, for $n_r = 4$, a broadcast bus will not require any overhead (no wire repeaters and even less power consumption) compared to a point-to point connectivity. The wire model suggests that with any type of wire, we can reach over 2.2 GHz or over 1.4GHz bus frequency on the broadcast bus for nr = 4,8 or $n_r = 16$, respectively. The area of the bus per PE is 0.023 mm^2 and the worst case the bus power is negligible.

Overall area, power and performance estimates for our PE design at various operating points are summarized in Table 3.1. Running at a clock frequency of 1 GHz, a 4×4 LAP core is estimated to achieve an efficiency of 110 single-precision or 45 double-precision GFLOPS/W. We stress that the point of this section is not to present the ultimate design.

To find the best combination of components and the best operating frequency we used energy-delay W/$GFLOPS^2$ [50], as well as GFLOPS/W and GFLOPS/mm^2 efficiency metrics. The best design choice has a lower

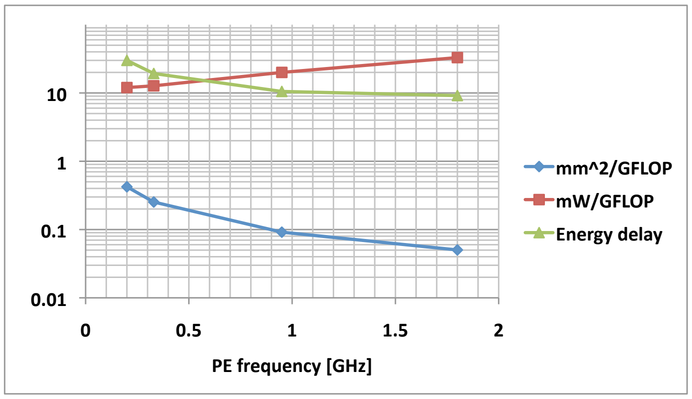


Figure 3.6: Efficiency metrics of PE. 1GHz appears to be the sweet-spot of the design.

energy-delay value and maintains high efficiency. Figure 3.6 shows the power/ throughput and the energy-delay for different PE frequencies. At 1.8 GHz there is not much degradation in the energy-delay while power/throughput increases significantly. At the left side of the spectrum low frequency designs have high efficiency but with high energy-delay and low area efficiency. A good tradeoff is achieved at a frequency of around 1 GHz, where energy-delay is still decreasing and there is high area and power efficiency. Figure 3.7 shows the trade-off between area/throughput, power/throughput and energy-delay. Low frequency designs are on the right side of spectrum. At 1 GHz, more than twice the area efficiency and energy-delay (0.1 mm^2/GFlop and 10 mW/$GFLOPS^2$) is achieved when compared to a design at 0.3 GHz. Also, compared to 1.8 GHz core, while having almost the same energy-delay, the power efficiency is 40% better.

Table 5.4 summarizes key metrics for various systems running GEMM as a representative matrix computation. For GPU and CPU architectures we

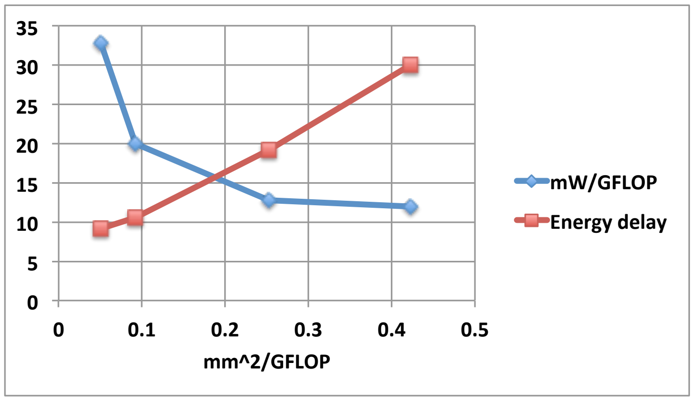


Figure 3.7: Power efficiency and energy-delay vs. area efficiency at different frequencies.

compare the LAC to Streaming Multiprocessors (SMs) and CPU cores, respectively [133]. For FPGAs we searched for the best GEMM implementation in 45nm technology, as reported on an Altera Stratix IV [100]. For the Cell Broadband Engine, we scaled the power reports for the SPEs [147] to 45nm technology and used the utilization numbers from [84]. We used the performance, power, and area reports for ClearSpeed CSX700 cores in [5] and scaled them to 45nm technology. Finally, we include core-level comparisons with tiles in a 80-tile network-on-chip architecture [141] and clusters of the Rigel accelerator [70].

We note that for a single-precision LAC at around 1GHz clock frequency, the estimated performance/power ratio is an order of magnitude better than GPUs. The double-precision LAC design shows around 55 times better efficiency compared to CPUs. The power density is also significantly lower as most of the LAC area is used for the local store. Finally, the performance/area ratio of our LAC is in all cases equal to or better than other processors. All

Architecture	$\frac{\text{W}}{\text{mm}^2}$	$\frac{\text{GFLOPS}}{\text{mm}^2}$	$\frac{\text{GFLOPS}}{\text{W}}$	Utilization
Cell SPE	0.4	6.4	16	83%
Nvidia GTX280 SM	0.6	3.1	5.3	66%
Rigel cluster	0.3	4.5	15	40%
80-Tile @ 0.8V	0.2	1.2	8.3	38%
Nvidia GTX480 SM	0.5	4.5	8.4	70%
Altera Stratix IV	0.02	0.1	7.0	90+%
LAC (SP)	0.2	19.5	104	95+%
Intel Core	0.5	0.4	.85	95%
Nvidia GTX480 SM	0.5	2	4.1	70%
Altera Stratix IV	0.02	0.05	3.5	90+%
ClearSpeed CSX700	0.02	0.28	12.5	78+%
LAC (DP)	0.3	15.6	47	95+%

Table 3.2: 45nm scaled performance and area of various cores running GEMM.

in all, with a double-precision LAC we can get up to 40 times better performance in the same area as a complex conventional core but using less than three quarter the power.

3.7 Summary

In this chapter we presented algorithm/architecture codesign of the linear algebra core. We showed the mapping of the GEMM algorithm on our proposed architecture. We developed analytical formulae and used it to study the core's design space tradeoffs. Power and performance estimates of the core and its components were presented. A LAC is expected to achieve a power efficiency of up to 50 GFLOPS/W, which is two orders of magnitude better than CPU cores and an order of magnitude better than GPU cores.

Chapter 4

Linear Algebra Processor (LAP) Design

In the previous chapter, we showed how a LAC can easily compute with data that already resides in on-chip memory. The question is now how to compose the GEMM $C \mathrel{+}= AB$ for general (larger) matrices from the computations that can occur on a (larger) Linear Algebra Processor (LAP) that is composed of multiple cores. The key is to amortize the cost of moving data in and out of the cores and the LAP. We describe that in this chapter again with the aid of Figure 3.3 (refined in Figure 4.1) . This framework will allow us to generally study tradeoffs in the memory hierarchy built around the execution cores. Using an optimized linear algebra core, we further extend our studies and codesign methodology to the next level of memory hierarchy and discuss the tradeoffs and power analyses of the multi-core linear algebra processor [111]. We observe the sources of inefficiency in other state-of-the-art architectures using our power studies.

4.1 The LAP Architecture

We start by translating the insights about the hierarchical implementation of GEMM on the LAC into a practical implementation of a LAP system.

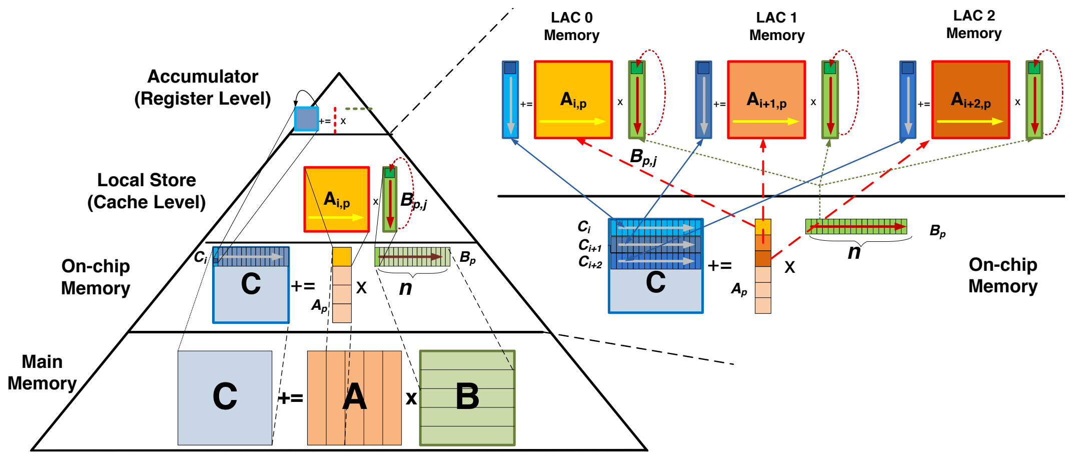


Figure 4.1: Memory hierarchy with multiple cores in a LAP system.

We investigate a simple system architecture that follows traditional GPU and multi-processor styles in which multiple cores are integrated on a single chip together with a shared on-chip L2 memory. The shared memory can in turn be banked or partitioned with a corresponding clustering of cores. In doing so, we derive formulae for the size of the shared on-chip memory and the required bandwidth between the LAP and external memory, all in relation to the number and size of the LAP cores themselves (see Section 3.4).

Figure 4.1 shows the use of the memory hierarchy for a larger matrix multiplication distributed across multiple cores. As discussed previously, each core locally stores a $m_c \times k_c$ (or $2m_c \times k_c$ to allow for prefetching to achieve peak performance) block of $A_{i,p}$, a n_r^2 subblock of $C_{i,j}$ and a $k_c \times n_r$ panel of $B_{p,j}$ (replicated across PEs), where different row blocks and panels of A and C are assigned to different cores. Bigger panels and blocks A, B and C are then stored at the next higher level of the memory hierarchy. Since elements of C are both read and written, we aim to keep them as close as possible to the execution units. Hence, the shared on-chip memory is mainly dedicated to storing a complete $n \times n$ block of matrix C. In addition, we need to share the current $k_c \times n$ row panel of B among the cores. With S cores in the LAP system and space for prefetching of blocks and panels of A and B, the total size of the on-chip shared memory therefore becomes $n^2 + S \times m_c \times k_c + 2k_c \times n$. This on-chip memory size does not reflect full overlapping of computations with communication in the chip level.

The intra-chip bandwidth required between cores and the on-chip memory for optimal performance can be computed as: $S \times m_c \times n$ elements of C have to be fed into the cores and the results collected back in $Sm_c n k_c / S n_r^2$ cycles, and $k_c \times n$ elements of B have to be broadcast to all cores in $m_c k_c n / n_r^2$ cycles. With this, the maximum bandwidth required for the shared, on-chip memory becomes $\frac{2S \times nr^2}{k_c} + \frac{n_r^2}{m_c}$. Extrapolating from the analysis presented in Section 3.4 with n/m_c row panels and subblocks evenly distributed across S parallel cores, and again assuming a limited memory bandwidth of y elements/cycle, a whole C += $A_p B_p$ computation including fetching of S $m_c \times k_c$ blocks of $A_{i,p}$ will require the following number of cycles:

$$\frac{n}{Sm_c}\left(\frac{Sm_c k_c}{y} + \max\left(\frac{(2Sm_c + k_c)n}{y}, \frac{Sm_c n k_c}{Sn_r^2}\right)\right).$$

When computation dominates (the second term in the “max” dominates) the peak performance is independent of m_c, i.e. independent of the granularity at which C and the A panel are split into row chunks. Thus, m_c

can be chosen to optimize memory bandwidth and the size of local store.

Finally, the required bandwidth between the LAP and external memory can be estimated. The bandwidth required for transfering the $k_c \times n$ panels of A_p and B_p in the $n^2 k_c / S n_r^2$ cycles required to process one such set of blocks, is $2Sn_r^2/n^2$. Furthermore, assuming we were to amortize reading and writing of n^2 elements of C over the n^3/Sn_r^2 cycles required to perform the whole computation for all n/k_c panels, the external bandwidth required would be the same as what is internally needed to feed the cores, i.e. $2Sn_r^2/n$. All combined, the maximum bandwidth required at the LAP's memory interface can be estimated as $3Sn_r^2/n$ for reading and Sn_r^2/n for writing from/to external memory. Conversely, if we assume an external memory bandwidth of z elements/cycle and overlap computation with communication of A and B but not of C, the whole matrix multiplication will take

$$\frac{2n^2}{z} + \max\left(\frac{2n^2}{z}, \frac{n^3}{Sn_r^2}\right) \text{ cycles.}$$

Overlapping transfers of C can be estimated in a similar fashion. Furthermore, given that at theoretical peak this computation would take n^3/Sn_r^2 cycles, the achievable utilization can be estimated.

4.2 Chip-Level Exploration

The overall system design is an optimization and exploration problem that strives to minimize the size of and bandwidth between layers of the memory hierarchy, while optimizing the performance and utilization of the cores.

Core	Local Memory size [Words/PE]	Intra-core BW [Words/Cycle]	Core-chip BW [Words/Cycle]
partial overlap	$(m_c k_c/n_r^2 + 2k_c)$	$n_r(1 + (2/k_c + 1/m_c))$	$(2/k_c + 1/m_c)n_r^2$
full overlap	$(2m_c k_c/n_r^2 + 2k_c)$	$n_r(1 + (2/k_c + 1/m_c + 1/n))$	$(2/k_c + 1/m_c + 1/n)n_r^2$

Chip	Memory Size [Words]	Intra-chip [Words/Cycle]	Off-chip BW [Words/Cycle]
partial overlap	$n^2 + Sm_c k_c + 2k_c n$	$(2S/k_c + 1(S)/m_c)n_r^2$	$2Sn_r^2/n$
full overlap	$2n^2 + Sm_c k_c + 2k_c n$	$(2S/k_c + 1(S)/m_c + S/n)n_r^2$	$4Sn_r^2/n$

Table 4.1: Bandwidth and memory requirements of different layers of memory hierarchy.

Given specific restrictions, e.g. on memory bandwidth or input matrix size, this yields the number of PEs in each core, the number of cores on a chip and the sizes and organization of the different levels of the memory hierarchy.

Table 4.1 summarizes the bandwidth and sizes of different layers of the memory hierarchy. This table shows the demands of the partially overlapped and the fully overlapped versions of the algorithm as a function of the number of cores, block sizes, and matrix size when $m = n = k$. In the core level analyses, the partially overlapped version assumes that bringing blocks of $A_{i,p}$ to the core is not overlapped with computation. At the chip level, partially overlapped versions assume that transferring of matrix C to and from off-chip memory is not overlapped with computation.

The main design challenge is to understand the dependency of design parameters on each other and their effects on power, area, and performance. In the following, we describe several explorations of the design space and analyze the tradeoffs between parameters and the overall performance. Later, we will merge the knowledge gained from these studies with power and area models to explore the design space from a practical perspective.

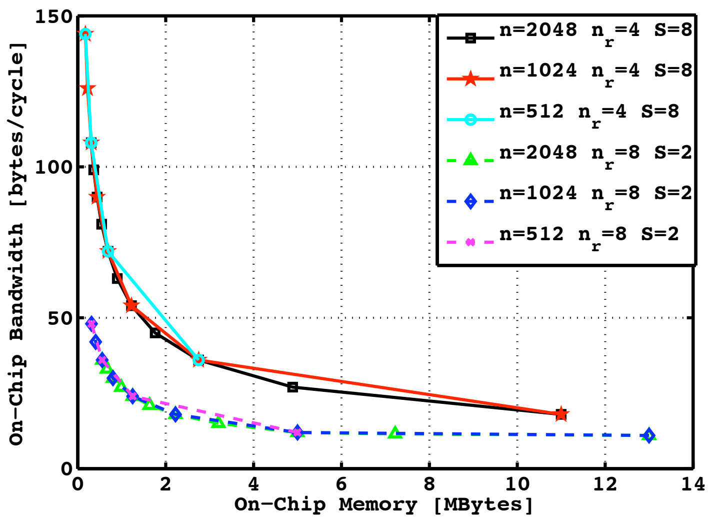


Figure 4.2: On-chip bandwidth vs. memory size for different core organizations, and problem sizes for fixed number of total PEs, and $m_c = k_c$. The utilization in all cases is over 93%.

4.2.1 Memory size vs. bandwidth

Based on our analytical model, we can evaluate the trade-off between the size of the on-chip memory and the intra-chip bandwidth between cores, and the on-chip memory, as shown in Figure 4.2. The resulting utilization in all cases is over 90%. We explore this trade off for $S = 8, n_r = 4$ and $S = 2, n_r = 8$ with a total number of PEs on the chip $(S \times n_r^2)$ equal to 128 in both cases. We can note that bandwidth demands grow quadratically as the size of available on-chip memory is reduced. This graph also demonstrates that bigger but fewer cores on the chip demand much less on-chip bandwidth. However, for a fixed problem size of C, bigger cores will require a bigger size for the on-chip memory, leading to a tradeoff between on-chip memory size and bandwidth. This extra space requirement is due to wider panels of A and

B that must be stored in the shared memory.

4.2.2 Number of LACs vs. on-chip bandwidth and memory size

We analyze the overall performance of the design when the number of cores is increased for different on-chip memory sizes and on-chip memory bandwidths. The curves in Figure 4.3 show the percentage of performance compared to a single 4×4 core for different numbers of cores and available on-chip bandwidths. The graph contains four sets of four curves where each set has the same ratio for the number of cores to available on-chip bandwidth S/BW, (indicated by same marker type). We observe that for small memory sizes different points of the same set with the same S/BW ratio all exhibit similar performance. Although the on-chip bandwidth is increased linearly with the number of cores, there is no performance improvement. To achieve performance gains when increasing the number of cores, the bandwidth has to grow superlinearly. However, as the size of memory increases, there is more benefit for using more cores to gain performance even with linear bandwidth increases.

For configurations with the same number of cores S, (indicated by the same line style or color) we observe that, as the bandwidth increases, the curves reach a peak eventually. The point in each curve with the smallest on-chip memory and peak performance is the optimal design point. Note that such a point is on the optimal design curve in Figure 4.2, too. For example, for $S = 8$ cores, a bandwidth of 4 bytes or words/cycle, with an on-chip memory size of

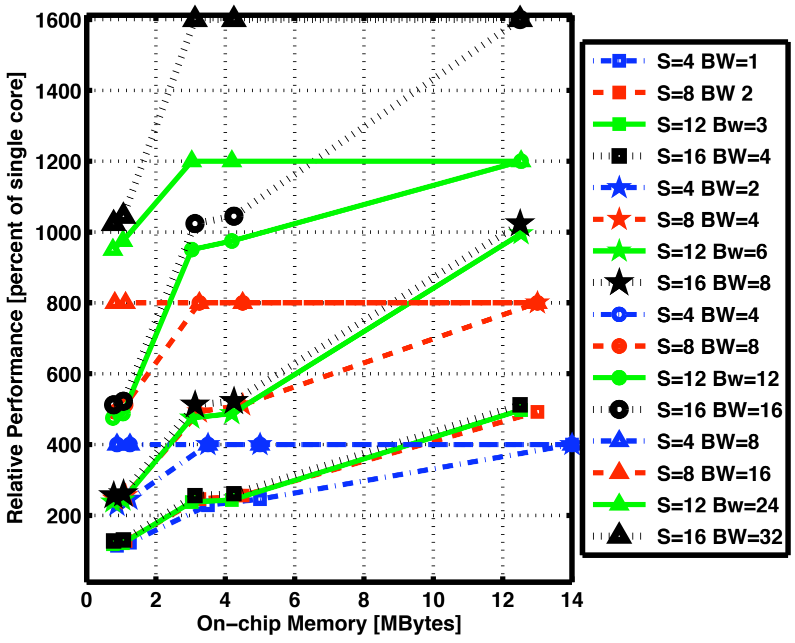


Figure 4.3: LAP performance for different on-chip memory sizes, different number of cores, and different total on-chip bandwidths with $n_r = 4$ and s=4, 8, 12, 16.

13 Mbytes, and a bandwidth of 8 bytes/cycle with an with on-chip memory size around 3 MBytes are both optimal design points.

As mentioned above, the increase in bandwidth requirements needed for maintaining optimal performance with an increase in the number of cores is exponential. This can be further studied by finding the optimal points that have same on-chip memory size, but a different number of cores. For example, to achieve peak performance with different number of cores $S = 4, 8, 16$ at 2.5 MBytes on-chip memory, the required bandwidth is 2, 8, 32. This shows the quadratical growth in bandwidth demand to maintain utilization when increasing the number of the cores.

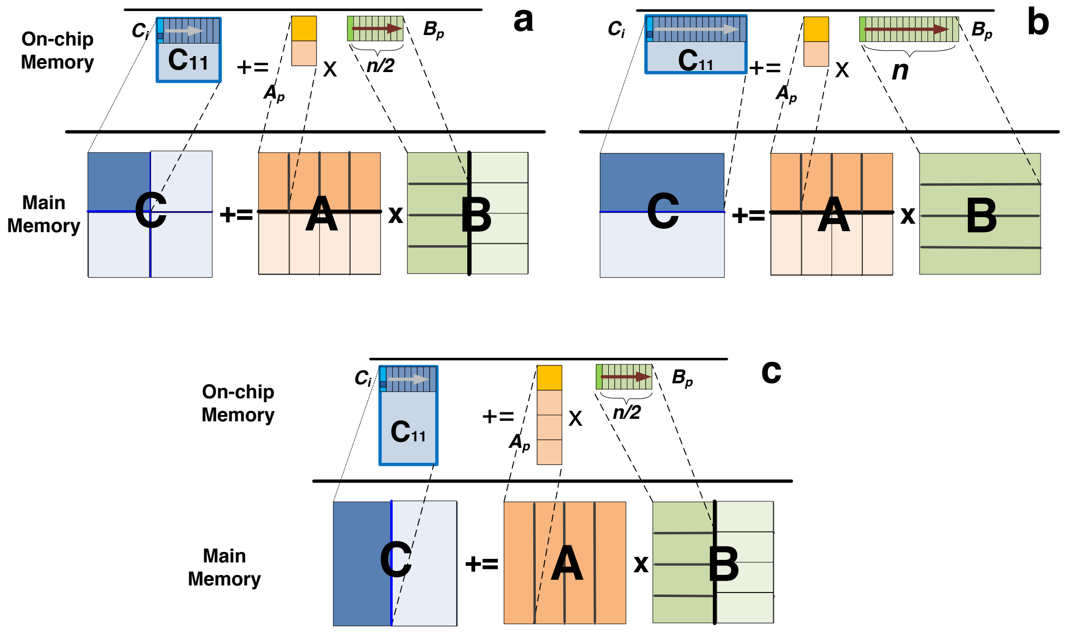


Figure 4.4: Blocking algorithm to map a big problem on a small on-chip memory. a) blocking for quarter size b,c)blocking for half size.

4.2.3 On-chip memory size vs. off-chip bandwidth

Finally, we analyze the tradeoff between the size of the on-chip memory and the external, off-chip bandwidth. We assume that the problem size and number of cores are fixed, and initially the optimal local store size is allocated in the cores and PEs on the chip. Next, we shrink the available on-chip memory and compute the external bandwidth demands to keep the performance over 90%. The algorithmic solution to this problem is adding another layer of blocking as shown in Figure 4.4. The matrix dimension of the original problem size is is n and the new block size is n_s. We call this ratio $d = \frac{n}{n_s}$. After shrinking the available on-chip memory, the solution assumes that a single

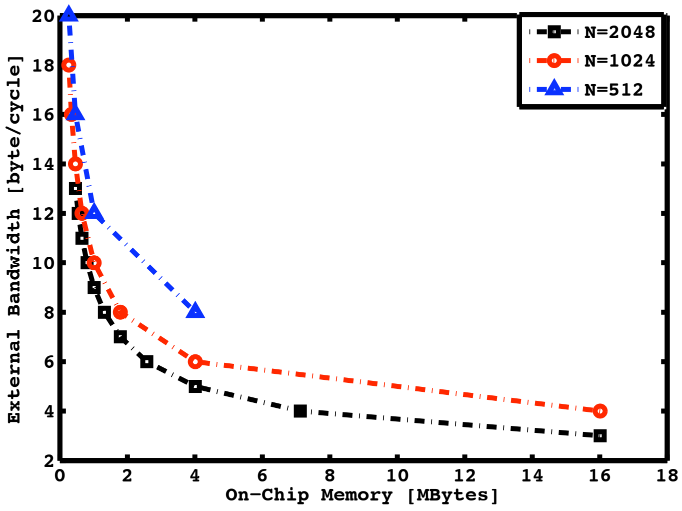


Figure 4.5: External Bandwidth vs. Size of on-chip memory tradeoff for different original problem sizes. All utilization numbers are over 92%.

(Figure 4.4-(a)) or $k \leq d$ (Figure 4.4-(b,c) k=d) sub-blocks of the original matrix C can fit on the new on-chip memory. Then, the algorithm performs all operations and data movements necessary to compute these k sub-blocks of C. The new off-chip bandwidth for the new smaller on-chip memory and a sub-problem size $k \times (n_s \times n_s)$ as part of the original $n \times n$ matrix multiplication can be computed as

$$\frac{k((2)n_s^2) + (k+1)nn_s}{kn_s^2 n} = \frac{(2)k + (k+1)d}{kn} \text{ elements/cycle}$$

Figure 4.5 shows the external bandwidth demands for three different problem sizes and how they increase as the size of on-chip memory is reduced. With growing original problem sizes $n \times n$, for the same on-chip memory size, the external bandwidth drops. We observe that as the original problem size increases, the external off-chip bandwidth requirement for the same system

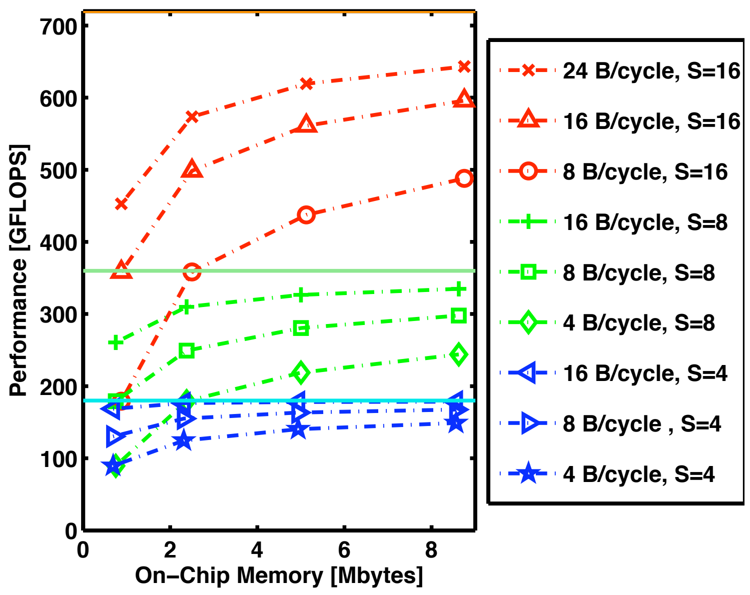


Figure 4.6: LAP performance as a function of external off-chip bandwidth and the size of on-chip memory with $n_r = 4$, $m_c = k_c$.

configuration decreases slightly. Still, the similar bandwidth vs. on-chip memory size trade-off exists to maintain high system utilization.

Figure 4.6 summarizes overall performance of a 1.4GHz LAP as a function of the size of the on-chip memory (dictating the possible kernel size), the number of cores, and the external bandwidth to the off-chip memory. Here we use $n_r = 4$, $m_c = k_c$ (the submatrix $A_{i,p}$ is square) and $n = 256, 512, 768$ or 1024 as the dimension of matrix C (kernel size, which translates into a corresponding on-chip memory size). As we increase the available core parallelism, the needed off-chip bandwidth increases for the same problem size[1]. Also when problem size grows, with same off-chip bandwidth

[1]Note that the needed on-chip memory size also increases slightly due to additional

we get better performance. This graph shows that a small L2 memory size (e.g. as is the case in GPUs), which determines the possible on-chip problem size, limits the achievable peak utilization (”exploitable parallelism”). Overall, with 16 cores, 5 Mbytes of shared on-chip memory and an external bandwidth of 16B/cycle, we can achieve 600 GFLOPS out of 700 GFLOPS peak.

4.3 Model Validation and Performance Prediction

The analytical models that we presented so far can help designers in early stages of the design process verify performance and utilization of their architecture for the class of matrix operations. In this section, we demonstrate the benefits and feasibility of our analytical models for early performance prediction by using them to discuss common sources of inefficiencies in existing architectures. We specifically study examples of state-of-the-art GPU and other accelerated architectures.

There are two common limitations in parallel architectures that restrict their performance and efficiency. First, the core architectural and micro-architectural features can limit the accessibility of local register files and the number of instructions executed in each cycle [106]. Second, the memory hierarchy organization that includes sizes of layers and bandwidths between them might not be able to sustain data movement from/to the computation cores. In the following, we assume that the cores are perfectly designed. The main

storage required for prefetching across more cores.

metric affected by core-level design issues is the achievable peak efficiency in terms of both energy spent per operation (GFLOPS/W) and achievable utilization. We have shown how to design such an ideal core in Chapter 3. A further study of core-level micro-architectural tradeoffs is outside of the scope of this document. Instead, we focus on analysis of the memory hierarchy. The main efficiency metric affected by the memory hierarchy trade-off is achievable utilization. In the following, we will specifically show how we can apply our analytical memory hierarchy model to predict limitations in Nvidia's Fermi and Clearspeed's CSX architectures.

The Nvidia Fermi C2050 architecture has 14 cores with 16 double-precision MAC units in each core. The size of the on-chip cache is 768 KBytes. The clock frequency is 1.15 GHz. Let us assume that cores are designed to achieve up to peak performance. With 768 KBytes and $S = 14$ cores, the dimension of the largest block of matrix C that is evenly divisible by S and $n_r = 4$ while fitting in the on-chip memory is $n_s = 280$. Including the corresponding panels of A and B, this setup fills 700 KBytes of on-chip L2 cache. Dividing the block C into row panels among the 14 cores results in $m_c = n_s/S = 280/14 = 20$. Hence, the size of each row panel of C is $m_c \times n_s = 20 \times 280$. Thus, the parameters of the design are as follows: $m_c = k_c = 20, S = 14, n_s = 280$. Assuming full overlapping, the maximum required off-chip bandwidth according to Figure 4.1 is $(\frac{4\times14\times4^2}{280})\times1.15\text{GHz}\times8\text{Bytes}=$ 30GBytes/second, which is within the 144 GBytes/second that Fermi offers. The required on-chip bandwidth is $(\frac{2S}{k_c} + \frac{S}{m_c})n_r^2 = (\frac{2\times14}{20} + \frac{14}{20})4^2 \times 1.15$ GHz $\times 8$ Bytes$=$ 310 GBytes/second, which is much more than the 230 GBytes/second that Fermi offers. To calculate theoretically achievable utilization using such a configuration, we divide the available bandwidth by the demanded bandwidth: 230/310 = 74%. In reality, implementations of GEMM on C2050 achieve 70% of peak performance [133]. Hence, our model accurately predicts that the on-chip bandwidth of Fermi does not meet the needs of matrix multiplication. One can overcome this under-utilization by increasing the on-chip bandwidth (see above), or by increasing the on-chip memory size. If the size of on-chip memory is doubled in the previous case, the required on-chip bandwidth can drop to half, or 160 GBytes/second, using the solution in Figure 4.4-c.

We use the same methodology to analyze the Clearspeed CSX architecture. The CSX architecture achieves up to 78% of peak performance for matrix multiplication [5]. The CSX architecture has 128 KBytes of on-chip memory. The block of C that fits on this memory is 64×128. Again, we assume that this architecture has six optimal 4×4 cores. Using the algorithm described in Figure 4.4, with $d = 16, \widetilde{k} = 2$, the minimum off-chip bandwidth demand is 4.7 GBytes/second. With an actual 4 GBytes/second off-chip bandwidth, our predicted upper limit for achievable utilization for this architecture is 83%. We can increase the utilization by increasing the size of on-chip memory. If one doubles the size of memory it can fit 128×128 blocks of C. Using the same algorithm with $d = 8, \widetilde{k} = 1$, the minimum off-chip bandwidth drops to 3.375 GBytes/second, which is less than off-chip bandwidth provided by the CSX architecture.

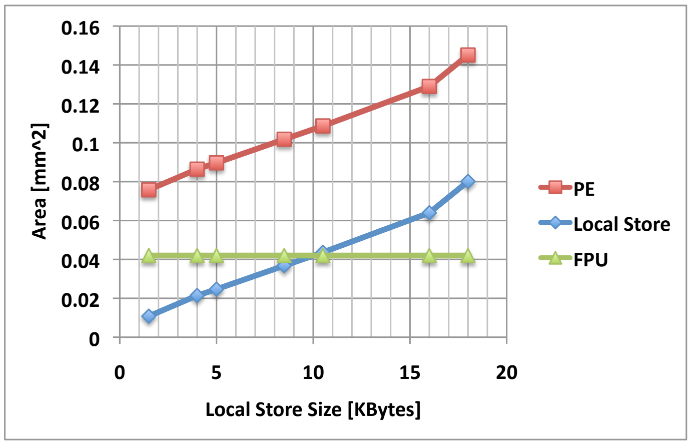


Figure 4.7: Area of a single PE in a 4x4 core for different local store sizes at 45nm.

4.4 Power and Area Exploration

In this section, we use power and area models to study the design space that we created in Section 4.2. We explore various trade-offs and how each design feature can affect the power and area consumption of the whole system. We use analytical results from Section 4.2 and apply representative power and area numbers to each point in the design space. This will allow us to evaluate how size and bandwidth of different layers of the memory hierarchy affect the overall performance and efficiency of the design.

At the core level, the goal is to have enough bandwidth and local store to maintain peak performance (equivalent to Figure 3.5) . We select the size of the core to be $n_r = 4$, and show the core-level area and power consumptions. Figure 4.7 illustrates the area of different components within the PE. With a local store size of 18 KByte, the local store occupies at most 2/3 of the PE, which exhibits a linear relation to the local store capacity size. The

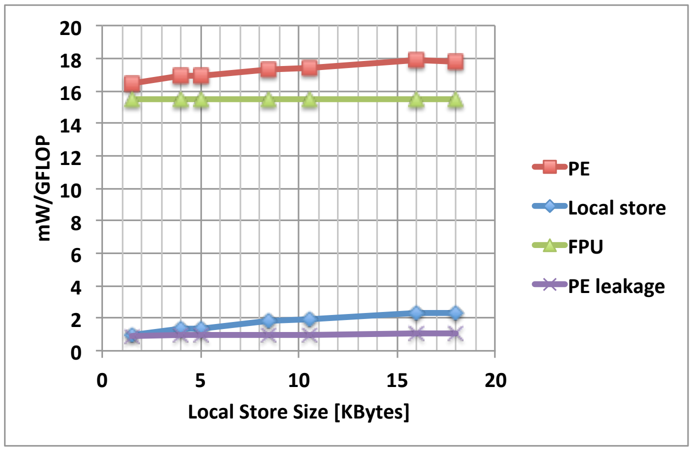


Figure 4.8: Leakage, local store, and total power efficiency of a PE at in a 4x4 core at 45nm.

power/throughput ratio of the PE, the local store, and the total leakage is shown in Figure 4.8. The graph suggests that with smaller local stores and even with higher bandwidths still less power is consumed by each PE. The overall PE power consumption is dominated by the FPU. These graphs advocate smaller local store sizes in terms of power and area consumption. However, there are three reasons that force larger PE local stores. First, the power density increases if local store size is reduced, which may limit the overall performance. Second, although decreasing the local store size does not affect the core power consumption, the required on-chip bandwidth will increase quadratically, which decreases the utilization and also results in a significant increase of the total power consumption. Finally, as we will discuss later for algorithms like Cholesky factorization where all the data is in-core, a bigger local store per PE yields to the ability of handling bigger kernels and amortizing more of the irregular computations over the available parallelism.

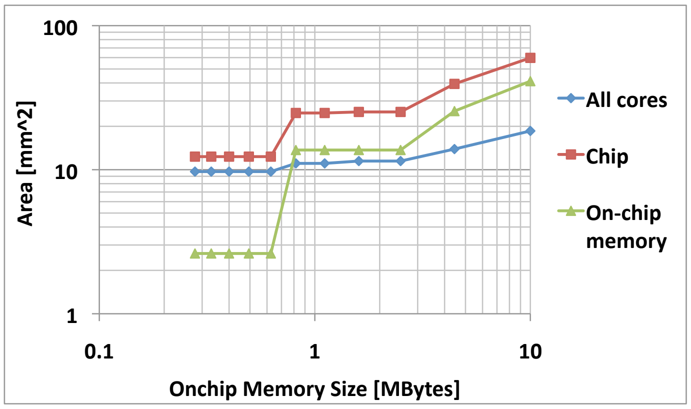


Figure 4.9: Area of cores, on-chip memory and a total 128 MAC unit system with S=8 4x4 cores, different on-chip SRAM memory sizes, and n=2048.

At the chip level, we estimate the effect of on-chip memory size on overall power and area while maintaining peak performance (similar to Figure 4.5). For each on-chip memory size, there are different options in terms of core configuration. We choose the biggest possible local store size to minimize intra-chip traffic and hence power consumption. Here, the power consumption due to external accesses is not included. Figure 4.9 shows the area consumption of the cores and on-chip memory. Figure 4.10 shows that with our domain specific design of on-chip SRAM memory almost all of the power of the chip is used by the eight cores and memory trade-offs are negligible.

In order to get a better sense of memory trade-offs in more general systems, we performed the same analysis using the NUCA [93] memory simulator of CACTI and replacing the SRAM design by Nuca caches. Here, the effects of increased bandwidth with smaller memory sizes are seen more realistically. In our LAP design, we use single-ported memory banks in low-power

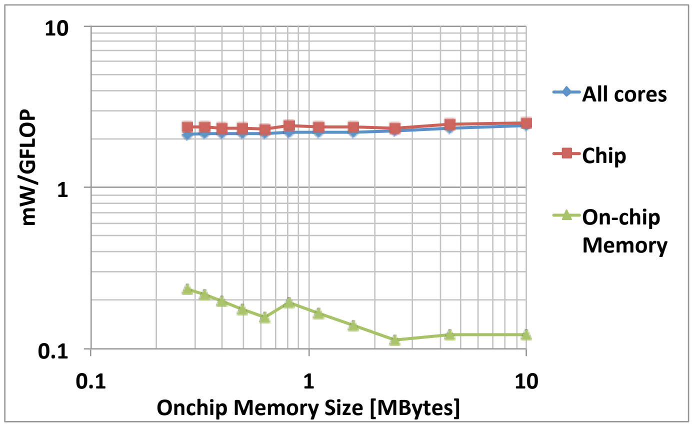


Figure 4.10: Power efficiency of cores, on-chip memory and a total 128 MAC unit system with S=8 4x4 cores, different on-chip SRAM memory sizes, and n=2048.

technology and with low clock frequencies. In a Nuca cache based design, either multi-ported caches or high-performance, high-power banks have to be used to maintain the same high bandwidths at small memory sizes. We chose high-performance, high-power caches since they require less area and power compared to multi-ported designs. As shown in Figure 4.11, in all cases the on-chip Nuca memory occupies more space than the computation cores do. Furthermore, a design with small capacity, high bandwidth banks ends up occupying more space than a larger, slower on-chip memory. Higher bandwidth also affects the power consumption of the system. Figure 4.12 shows that at lower capacities, on-chip Nuca memory consumes more power than the computation cores. In other words, a design with larger, simpler on-chip Nuca cache size is both more power and more area efficient.

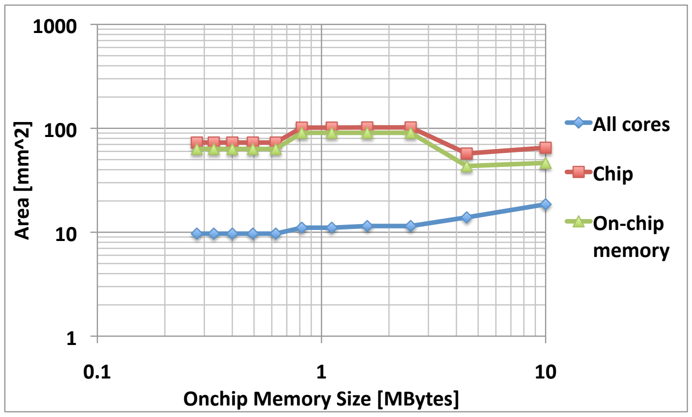


Figure 4.11: Area of cores, on-chip memory and a total 128 MAC unit system with S=8 4x4 cores, different on-chip NUCA memory sizes, and n=2048.

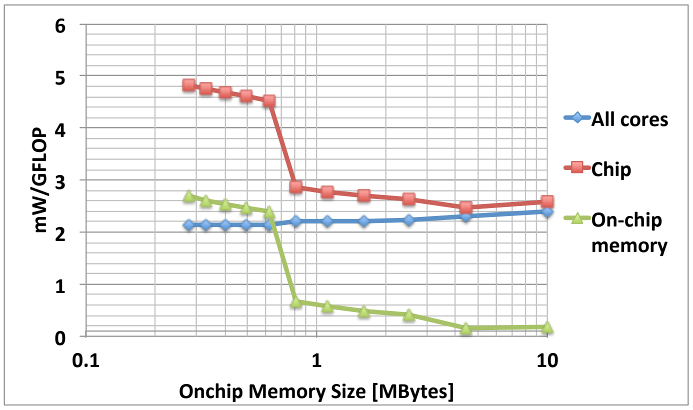


Figure 4.12: Power efficiency of cores, on-chip memory and a total 128 MAC unit system with S=8 4x4 cores, different on-chip NUCA memory sizes, and n=2048.

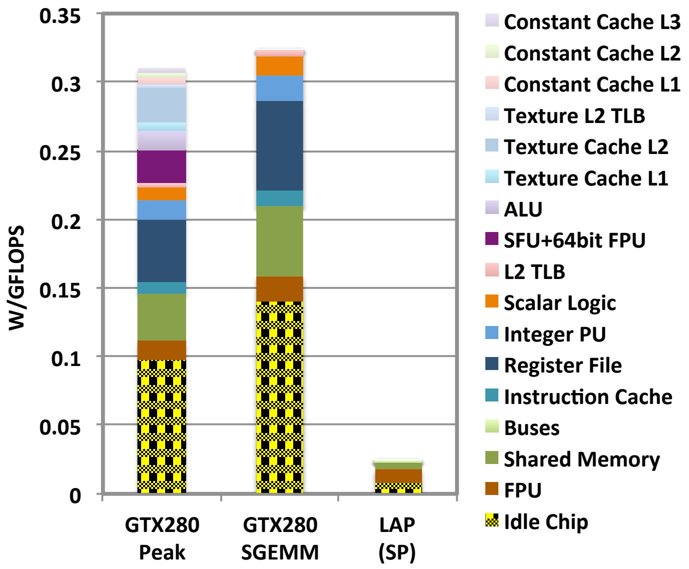


Figure 4.13: Normalized power breakdown of Nvidia Tesla GTX280 versus LAP at 65nm.

4.5 Comparative Power and Performance Analysis

Figures 4.13, 4.14, and 4.15 show a breakdown of performance-normalized power consumption for current high-performance GPGPU and multi-core architectures as compared to single- or double-precision versions of a prototypical LAP with an equivalent number of cores (i.e. Shared Multiprocessors, SMs, in GPUs[2]) running at equivalent raw single FMAC performance (1.3GHz or 1.4GHz). In the case of GPUs (Figures 4.13, 4.14), we show efficiencies for both peak operation and when running GEMM. Current GPUs run single- or double- precision GEMM (SGEMM or DGEMM) at only around 60% of their

[2]In the GTX480, each SM provides 16-way double-precision or 32-way single-precision parallelism. Correspondingly, we replace SMs with one or two 4×4 double- or single-precision LAP cores, respectively.

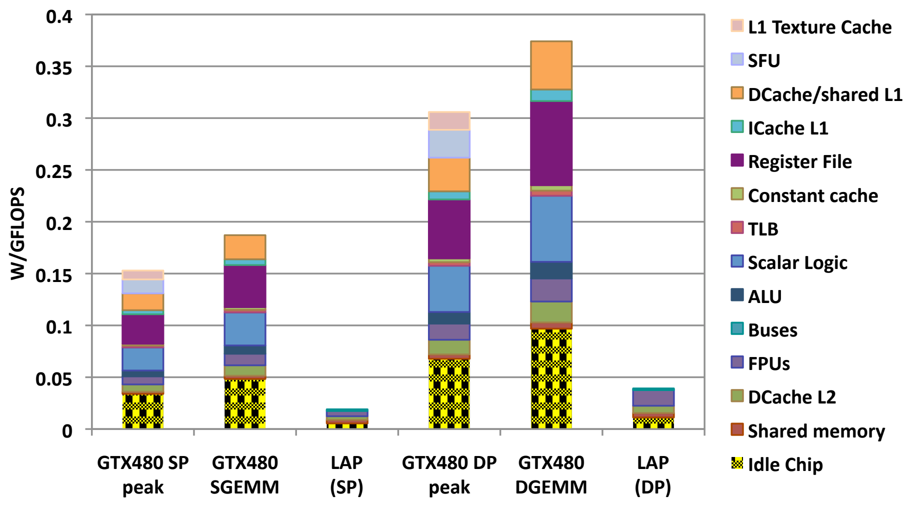


Figure 4.14: Normalized power breakdown of Nvidia Fermi GTX480 versus LAP at 45nm.

theoretical peak FPU performance [10, 94, 143]. As the graphs show, reduced utilization has a significant effect on achievable efficiencies, even when considering that unneeded components, such as constant caches, texture caches, extra ALUs or special functional units (SFUs) can be turned off. By contrast, the Intel Penryn dual-core processor and a LAP with two 4×4 cores running at 1.4GHz, i.e. at around half of the Penryn's 2.66GHz clock speed, achieve near peak utilization at a moderate performance of 20 and 90 double- precision GFLOPS, respectively (Figure 4.15).

Breakdowns show that traditional architectures include significant overhead. The only units that are really useful for performing matrix multiplication are FPUs/execution units, shared memories/L1 caches, L2 caches and TLBs. In the GPUs, components like shared memories, instruction caches or register files can consume up to 70% of the power, and in some cases the register

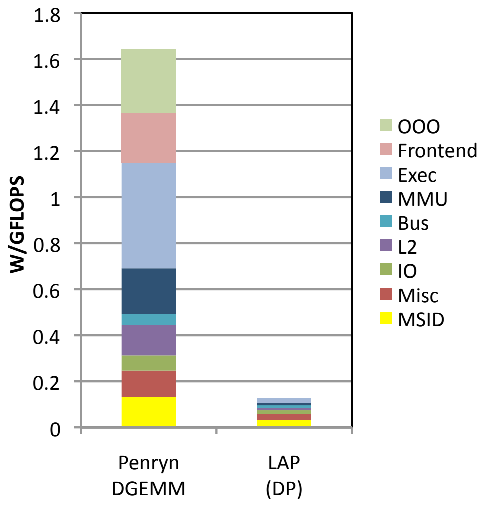


Figure 4.15: Normalized power breakdown of Intel dual-core Penryn versus LAP at 45nm.

file alone contributes more than 30%. By eliminating instructions, associated cache power is removed from the LAP. Similarly, register files are very small and shared memories are replaced by sequentially accessed, partitioned SRAM with a maximum of 2 read/write ports. For the Penryn, we mainly relied on the power breakdown presented in [47], where we assumed that GEMM utilizes all of the core. In the graph, the SRAMs and MACs of the LAP are listed under the MMU and execution unit categories. We conservatively added all of the miscellaneous and IO power consumption factors to the LAP, which favors the Penryn in this comparison. We can observe that the Penryn uses 40% of the core power (over 5 W) in the Out of Order and Frontend units that do not exist in LAP architecture. Furthermore, with around 5 W the execution unit consumes one third of the core power, which may be attributed by support of exception handling and IEEE-754 full compatibility.

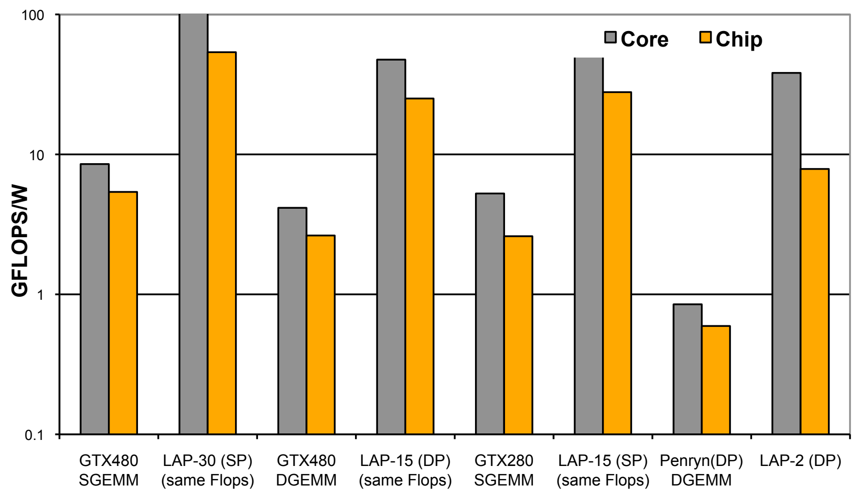


Figure 4.16: Comparison of efficiencies for single- and double-precisionbuses GEMM between NVidia Tesla GTX280, NVidia Fermi GTX480, Intel Penry and a LAP of equivalent throughput.

Finally, we compare overall efficiency and inverse energy-delay [50] of single- and double-precision realizations of our design against other systems. Figure 4.16 shows an analysis of core- and chip-level efficiencies for studied architectures and a LAP in which we vary the number of cores to match the throughput in existing architectures. Our LAP with 30 single- or 15 double-precision cores and 5Mbytes of on-chip memory achieves a GEMM performance of 1200 and 600 GFLOPS at a utilization of 90% in an area of 115 mm^2 or 120 mm^2, respectively. By comparison, the dual-core CPU achieves 22 GFLOPS in 100mm^2 and the GTX480 runs SGEMM/DGEMM with 780/390 GFLOPS and 58% utilization using 15 SMs in total 500mm^2 chip area.

Table 5.4 summarizes key metrics for various systems running GEMM as a representative matrix computation. For this table, we extended the analysis presented in [70] by including estimates for our LAP design, the 80-tile network-on-chip architecture from [141], the Power7 processor [144], the Cell processor [84], Intel Penryn [47], Intel Core i7-960 [27], CSX700 [5], Altera Stratix IV [100], and the NVidia Fermi GPU (GTX480) [68] all scaled to 45nm technology and to GEMM utilizations.

We note that for a single-precision LAP at around 1.4GHz clock frequency, the estimated performance/power ratio is an order of magnitude better than GPUs. The double-precision LAP design shows around 30 times better efficiency compared to CPUs. The power density is also significantly lower as most of the LAP area is used for local store. The performance/area ratio of our LAP is in all cases equal to or better than other processors. Finally, the inverse of energy-delay of LAP is at least an order of magnitude better that all other designs. All in all, with a double-precision LAP we can get up to 32 times better performance in the same area as a complex conventional core but using almost the same power.

Overall, some of the major differences between traditional general-purpose designs and a specialized linear-algebra architecture lie in the memory architecture and the core execution unit datapaths. The LAP has relatively large L1- and L2-equivalent PE and on-chip memories, comparable in size to multi-core architectures but an order of magnitude bigger than in GPUs. This keeps bandwidth between memory layers low. All memories are pure, banked SRAMs with no tagging or cache consistency overhead. Consequently, memories are more power efficient and smaller than in other architectures despite

Architecture	GFLOPS	$\frac{\text{W}}{\text{mm}^2}$	$\frac{\text{GFLOPS}}{\text{mm}^2}$	$\frac{\text{GFLOPS}}{\text{W}}$	$\frac{\text{GFLOPS}^2}{\text{W}}$	Utilization
Cell	200	0.3	1.5	5.0	1000	88%
Nvidia GTX280	410	0.3	0.8	2.6	1066	66%
Rigel	850	0.3	3.2	10.7	9095	40%
80-Tile @0.8V	175	0.2	1.2	6.6	1155	38%
80-Tile @1.07V	380	0.7	2.66	3.8	1444	38%
Nvidia GTX480	940	0.2	0.9	5.2	4230	70%
Core i7-960	96	0.4	0.50	1.14	109.44	95%
Altera Stratix IV	200	0.02	0.1	7	1400	90+%
LAP (SP)	1200	0.2	6-11	30-55	66000	90+%
Intel Quad-Core	40	0.5	0.4	0.8	32	95%
Intel Penryn	20	0.4	0.2	0.6	12	95%
IBM Power7	230	0.5	0.5	1.0	230	95%
Nvidia GTX480	470	0.2	0.5	2.6	1222	70%
Core i7-960	48	0.4	0.25	0.57	27.36	95%
Altera Stratix IV	100	0.02	0.05	3.5	350	90+%
ClearSpeed CSX700	75	0.02	0.2	12.5	937.7	78+%
LAP (DP)	600	0.2	3-5	15-25	15000	90+%

Table 4.2: 45nm scaled performance and area of various systems running GEMM.

being larger. Shared on-chip memory can be partitioned among groups of cores with each bank being only coupled with its set of cores. Note that we do not include external memory in our analysis. With system architectures increasingly integrating host processors and accelerators on a single die, we can expect similar benefits to extend into other such memory layers. Again, larger on-chip memories in the LAP help to decrease external memory bandwidth and power consumption requirements.

For execution units and data paths, we can observe that unnecessary overheads are removed by performing whole chains of operations in local accumulators without any register file moves that become necessary in traditional SIMD arrangements. This is further confirmed by low GEMM utilizations,

Power Waste Sources	CPUs	GPUs	LAP
Instruction Pipeline	ICache, Out of Order, Branch Prediction	ICache, In order, NA	No Instructions
Execution Unit	1D SIMD+RF	2D SIMD+RF	2D+Local SRAM/FPU
Register File & Move	Many Ported	Multiple Ported	8 Entry Single Ported
On-chip Memory Organization	Big Cache Strong Coherency	Small Cache Weak Coherency	Big SRAM Tightly Coupled Banks
Multi-Thread Support	SMT	Blocked MT	Not Supported
BW/FPU Ratio	High	High	Low (Enough)
Memory Size/ FPU Ratio	High	Low (Inadequate)	High

Table 4.3: Comparison between main design choices in the studied platforms.

which indicates that despite existing architectural features, idiosyncrasies of traditional architectures make it difficult to keep a large number of FPUs busy. Overall, the 2D PE arrangement with local, partitioned memory is scalable with exponential growth in computation power for a linear growth in interconnect and bus lengths. With relatively low overhead for specialized MAC units and broadcast buses, we can envision such specialized data paths to be integrated into standard processor pipelines for order of magnitude improved efficiency in a linear algebra computation mode. Table 4.3 summarizes the differences discussed in this section.

4.6 Summary

This chapter presented the integration of a multi-LAC architecture into a LAP with on-chip SRAM. Like Chapter 3, we studied the architecture design space of this multi-core environment. We completed our analytical formulae, which demonstrate architectural tradeoffs between the memory bandwidth and storage size in different layers of the memory hierarchy. We used our analytical analyses and successfully predicted the utilization of some examples of existing architectures. We further presented power breakdowns for the LAP, and some examples of existing CPUs and GPUs to compare sources of inefficiency in those architecture. Our study shows how efficiency necessarily drops from a core to a multi-core design.

Chapter 5

Generalization to Level-3 BLAS

In previous chapters, we provided detailed studies for mapping of the GEMM algorithm on the LAP and its design tradeoffs across different levels of the memory hierarchy. The next goal that we pursue is codesign for flexibility and support for a whole class of operations. In this chapter, we extend our studies to other level-3 BLAS operations, demonstrating that with small micro-architectural modifications, the computation cores (LACs) can be extended to support the full set of level-3 BLAS operations with negligible loss in efficiency.

Main key to this success is the intra-core interconnect, which is realized with simple, data-only buses that do not require overhead for address decoding or complex control. This interconnect is specifically designed to efficiently realize all collective communications, including broadcast or transposition, necessary to support the execution of level-3 BLAS operations. While other architectures waste cycles and instructions to move data to their desired destination, the LAC architecture can inherently and transparently overlap computation, communication, and transposition. In this chapter, we emphasize some of these abilities by demonstrating the details of representative level-3 BLAS operations on the LAC.

5.1 Level-3 BLAS Operations

We start by describing the level-3 BLAS operations and their computation and data handling requirements.

- General Matrix Multiplication (GEMM): We discussed the details of this operation in Chapters 3 and 4. GEMM is the building block of the rest of the level-3 BLAS operations. Most of the computation intensity in the rest of level-3 BLAS routines is cast as GEMM operations.

- Symmetric Matrix Multiplication (SYMM): The SYMM operation computes $C := C + AB$ with a symmetric matrix $A \in \mathbb{R}^{m \times m}$ and rectangular matrix $B \in \mathbb{R}^{m \times n}$, updating $C \in \mathbb{R}^{m \times n}$. This operation is like GEMM with the difference that only the lower triangular part of matrix A is stored. Hence, to perform this operation, some blocks of A need to be transposed to recover the upper triangular part of A.

- Triangular Matrix Multiplication (TRMM): The TRMM operation computes $B := LB$ with a lower triangular matrix $L \in \mathbb{R}^{m \times m}$, and rectangular matrix $B \in \mathbb{R}^{m \times n}$. This operation uses the same block panel multiplication as in GEMM. However, the length of the panels increases in each iteration.

- Symmetric Rank-K update (SYRK): The SYRK operation computes $C := C + AA^T$ with a rectangular matrix $A \in \mathbb{R}^{n \times m}$, updating only the lower triangular part of the symmetric matrix $C \in \mathbb{R}^{n \times n}$. Matrix transposition needs to be implemented to perform this operation efficiently. We will discuss this operation in detail.

- Symmetric Rank-2K update (SYR2K): The SYR2K operation computes $C := C + AB^T + BA^T$ with a rectangular matrices $A, B \in \mathbb{R}^{n\times m}$, updating only the lower triangular part of the symmetric matrix $C \in \mathbb{R}^{n\times n}$. This operation is very similar to SYRK and uses the same principles.

- Triangular Solve with Multiple right-hand sides (TRSM): The TRSM operation solves a system of equations $LX = B$, with lower triangular matrix $L \in \mathbb{R}^{n\times n}$ and rectangular matrix $B \in \mathbb{R}^{n\times m}$, for $X \in \mathbb{R}^{n\times m}$, so that upon completion $X = L^{-1}B$. This is the most complex operation of the level-3 BLAS. Extra functions like division are needed in addition to multiply and add. We will discuss this operation in details as well.

We chose SYRK and TRSM as the two representative operations for which we show implementations on the LAC. If the data handling and functions of these two operations are supported by the LAC efficiently, this provides strong evidence for support of the rest of Level-3 BLAS operations. SYRK requires special data handling, namely matrix transpose, as part of its operation. TRSM requires extra functionality, namely the $1/x$ or reciprocal operation. We will see that multiple techniques must be exploited to extract parallelism and overcome dependencies in the TRSM operation.

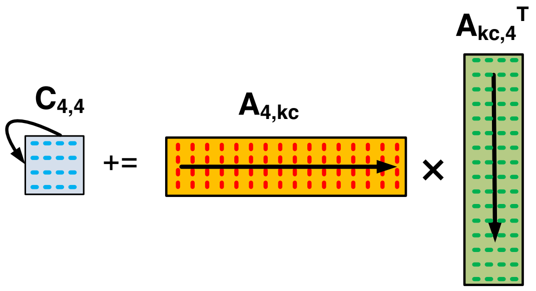


Figure 5.1: Computing the SYRK of a 4×4 matrix. While it looks similar to the matrix-matrix multiplication in Figure 3.2, notice that each column of A needs to be transposed as the sequence of rank-1 updates is performed.

5.2 SYRK and SYR2K

The SYRK operation computes $C := C + AA^T$ with a rectangular matrix $A \in \mathbb{R}^{n\times m}$, updating only the lower (or upper) triangular part of the symmetric matrix $C \in \mathbb{R}^{n\times n}$. There are two algorithms (blocked and unblocked) for computing SYRK that will be utilized. Different algorithms become appropriate at different levels of the memory hierarchy in which the data used by the computation is stored.

5.2.1 Unblocked SYRK on LAC

Let C and A be 4×4, and $4\times k_c$ matrices, respectively. Then $C \mathrel{+}= AA^T$ can be computed as a "block dot product" very similar to how GEMM is computed as series of rank-1 updates. The difference here is that instead of matrix B, A^T is going to be multiplied by matrix A as illustrated by Figure 5.1:

$$\begin{pmatrix} \gamma_{0,0} \cdots \gamma_{0,3} \\ \vdots \quad \ddots \quad \vdots \\ \gamma_{3,0} \cdots \gamma_{3,3} \end{pmatrix} \mathrel{+}= \begin{pmatrix} \alpha_{0,0} \\ \vdots \\ \alpha_{3,0} \end{pmatrix} \left(\alpha_{0,0} \cdots \alpha_{3,0}\right) + \begin{pmatrix} \alpha_{0,1} \\ \vdots \\ \alpha_{3,1} \end{pmatrix} \left(\alpha_{0,1} \cdots \alpha_{3,1}\right) + \cdots$$

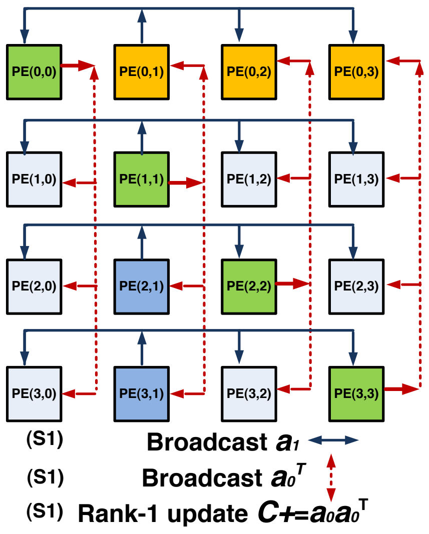


Figure 5.2: Second iteration of a 4 × 4 SYRK on LAC.

so that C is updated in the ith iteration with

$$\begin{pmatrix} \gamma_{0,0} + \alpha_{0,i}\alpha_{0,i} & \cdots & \gamma_{0,3} + \alpha_{0,i}\alpha_{3,i} \\ \vdots & \ddots & \vdots \\ \gamma_{3,0} + \alpha_{3,i}\alpha_{0,i} & \cdots & \gamma_{3,3} + \alpha_{3,i}\alpha_{3,i} \end{pmatrix}. \tag{5.1}$$

Let us assume that the $4 \times k_c$ matrix A is distributed to the PE array in a 2D cyclic round-robin fashion. We notice that the resulting matrix C in Equation 5.1 is symmetric. Also, to perform this operation, column vectors of A need to be transposed to perform each rank-1 update. This transpose operation can be overlapped with computation by taking advantage of the 2D arrangement of PEs and the broadcast buses. The diagonal PEs can receive columns of A from row buses and then broadcast them across the column buses to produce the transposed vector.

Thus, at the lowest level, the unblocked algorithm computes the SYRK of a $n_r \times n_r$ sub-matrix of C stored in the accumulators of the LAC from a

$n_r \times k_c$ sub-matrix of A. Three different operations take place in the same cycle in each iteration. Figure 5.2 illustrates the second ($i = 1$) iteration of a SYRK operation. The ith column of PEs broadcasts the values of the ith column of A, a_i, across the row busses, where the PEs in each row keep a copy of these values in their register file for use in the next iteration. At the same time, the values a_{i-1} from the previous iteration are transposed along the diagonal PEs by broadcasting them over the column busses. Hence, all PEs now have copies of elements of a_{i-1} and a_{i-1}^T, and a rank-1 update is performed to compute $C := C + a_{i-1} \times a_{i-1}^T$. The a_{i-1}^T is also kept in $(i-1)$th row of PEs to store A^T. This is repeated for $i = 0, \ldots, k_c$ cycles.

5.2.2 Blocked SYRK on LAC

A bigger SYRK for C of size $m_c \times m_c$ and A of size $m_c \times k_c$ can be blocked into smaller subproblems using the smaller SYRK (mentioned above) to update the diagonal $n_r \times n_r$ lower triangular blocks of C and produce the transpose of the corresponding $n_r \times k_c$ panels of A in a single iteration. Most of the computations are thereby cast into typical GEMM operations using the produced panel of A^T and the remaining panels of A.

The blocked algorithm that we will use can be derived by partitioning A and C as

$$C = \left(\begin{array}{c|c|c} C_{00} & 0 & 0 \\ \hline C_{10} & C_{11} & 0 \\ \hline C_{20} & C_{21} & C_{22} \end{array}\right) \quad , \quad A = \left(\begin{array}{c} A_0 \\ \hline A_1 \\ \hline A_2 \end{array}\right) \quad , \text{ and } \quad A^T = \left(\begin{array}{c|c|c} A_0^T & A_1^T & A_2^T \end{array}\right)$$

Then, $C = AA^T$ means:

$$C+=\left(\begin{array}{c|c|c} C_{00}+A_0A_0^T & * & * \\ \hline C_{10}+A_1A_0^T & C_{11}+A_1A_1^T & * \\ \hline C_{20}+A_2A_0^T & C_{21}+A_2A_1^T & C_{22}+A_2A_2^T \end{array}\right)$$

If the computation has progressed to the point where C_{00}, C_{10}, and C_{20} have already been updated with their final results and the rest of C has not yet been updated, then we can update C_{11} and C_{21} as:

$$C_{11}+=A_1A_1^T \quad \text{and} \quad C_{21}+=A_2A_1^T.$$

A blocked SYRK performs computations with matrices A and C that fit in the LAC local memory, while the computation $C_{11}+=A_1A_1^T$ is performed by an unblocked variant of SYRK on the LAC.

For simplicity, we assume that m_c and k_c are divisible by n_r, i.e. $m_c = mn_r$, and $k_c = kn_r$. The $m_c \times k_c$ block of matrix A is distributed among the local stores in a 2-D round-robin fashion much like it was for GEMM (as described previously). We will describe how a single iteration of the blocked down-looking SYRK algorithm is mapped onto the LAC. Figure 5.3 shows a case with $m_c = 8n_r$. Highlighted is the data involved in the fifth iteration. We describe the different operations to be performed:

(1a) $C_{11} := C_{11} + A_1A_1^T$: The block C_{11} of C is moved to the accumulators and $C_{11} := A_1A_1^T$ is performed as a smaller SYRK, which is computed by the LAC as described previously.

(1b) $A_1^T := (A_1)^T$: As mentioned before, as part of computing C_{11} in (1a),

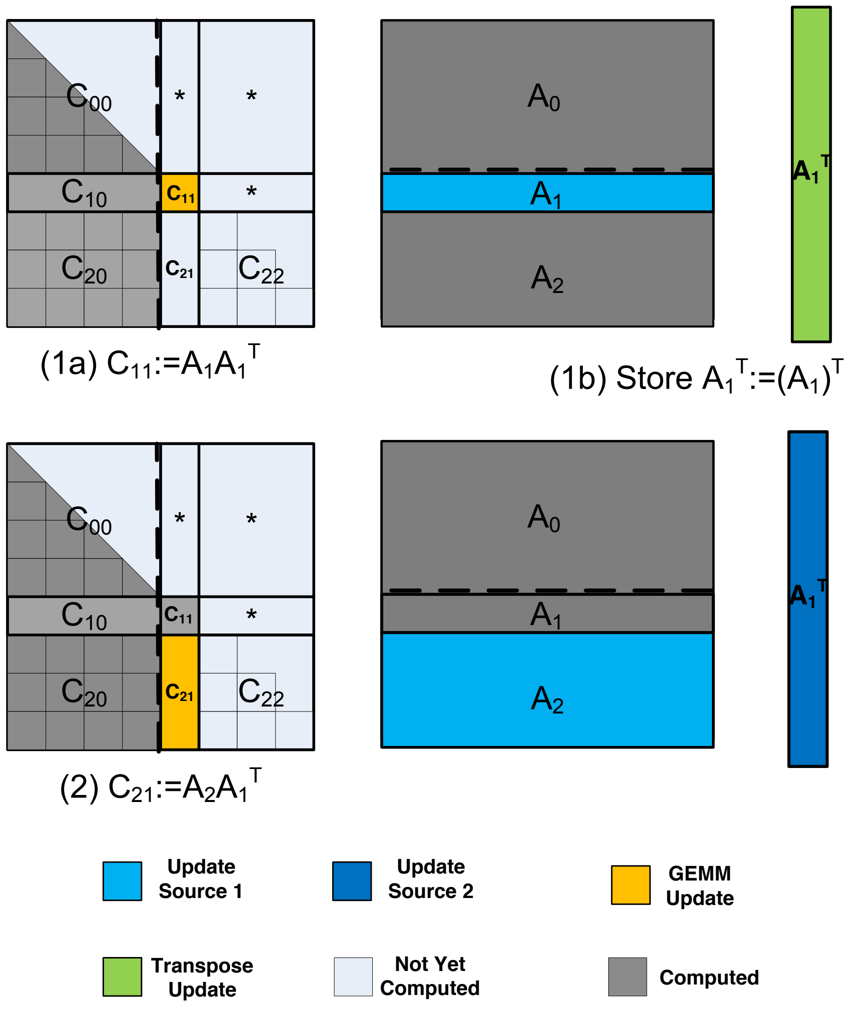


Figure 5.3: Blocked SYRK, fifth iteration.

the transpose A_1^T is formed and stored in the PE rows for future use in (2).

(2) $C_{21} := A_2 A_1^T$: In this stage, a matrix multiplication as described in Chapter 3 is performed. Successive $n_r \times n_r$ blocks of C_{21} are brought in and out of the accumulators of the PEs. A_1^T was already broadcasted and saved in the PEs as part of (1). Finally, successive $n_r \times m_c$ row panels of A_2 are multiplied by A_1^T to update the corresponding successive $n_r \times n_r$ blocks of C_{21}.

The down-looking algorithm has very good data locality for the core GEMM operations. No extra storage is needed for blocks of A_1^T and matrix A is resident in the LAC throughout all computations.

For even larger matrices that do not fit into the LAC local memories, a further hierarchically blocked SYRK is utilized. There, another level of blocking is used, which is composed out of two smaller SYRK and GEMM kernels that each fits in the LAC local store. Appropriate blocking of matrices thereby facilitates reuse of data, which reduces the need for high bandwidth between the memory banks of the PEs, the on-chip memory and the external storage. Details go beyond the scope of this document.

The LAC uses very similar principles as for SYRK to perform the SYR2K operation and its blocked versions. The SYR2K produces $C := C + AB^T + BA^T$ by cross-multiplying rectangular matrices $A, B \in \mathbb{R}^{n \times m}$ by their transpose to update the lower triangular part of the symmetric matrix $C \in \mathbb{R}^{n \times n}$. The amount of both communication and computation is doubled in this case.

5.3 TRSM

The TRSM operation solves a system of equations $LX = B$, with lower (or upper) triangular matrix $L \in \mathbb{R}^{n\times n}$ and rectangular matrix $B \in \mathbb{R}^{n\times m}$ for $X \in \mathbb{R}^{n\times m}$, such that upon completion $X = L^{-1}B$. As with SYRK, there are two of algorithms (one blocked and one unblocked) that will be utilized, depending on the level of the memory hierarchy in which the data is stored at.

5.3.1 Unblocked TRSM on LAC

The inner kernel of TRSM uses an unblocked algorithm which we first briefly describe.

Partition

$$X = \left(\begin{array}{c} x_1^T \\ \hline X_2 \end{array}\right) \quad , \quad B = \left(\begin{array}{c} b_1^T \\ \hline B_2 \end{array}\right) \quad , \text{ and } \quad L = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right),$$

where b_1^T is a row vector and λ_{11} is a scalar. Then $B = LX$ means that

$$\left(\begin{array}{c} b_1^T \\ \hline B_2 \end{array}\right) = \left(\begin{array}{c} \lambda_{11}x_1^T + 0 \\ \hline l_{21}x_1^T + L_{22}X_2 \end{array}\right),$$

which in turn means that

$$\begin{array}{c} b_1^T = \lambda_{11}x_1^T \\ \hline B_2 - l_{21}x_1^T = L_{22}X_2 \end{array}.$$

We can thus compute X for matrix B as follows

$$\begin{array}{c} b_1^T := x_1^T = b_1^T/\lambda_{11} \\ \hline B_2 := X_2 = L_{22}^{-1}(B_2 - l_{21}x_1^T) \end{array}.$$

In each iteration, i, the unblocked variant performs two operations: First, the corresponding row vector of B, b_i^T is replaced with the result of the same row

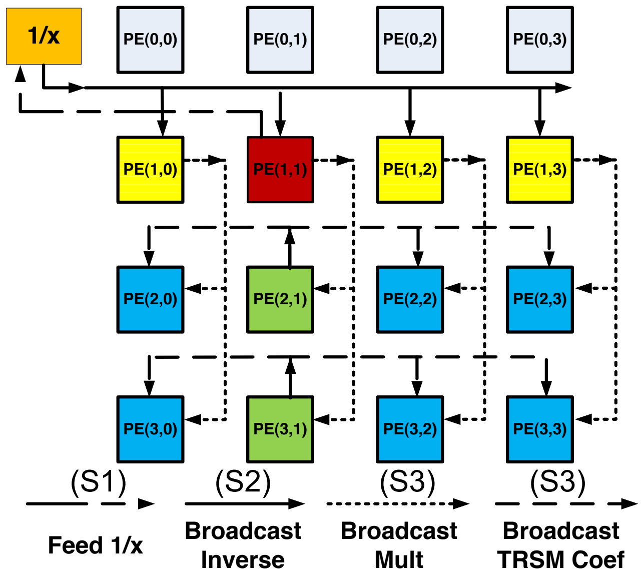


Figure 5.4: Second iteration of a 4×4 TRSM operation mapping on LAC.

in $X = L^{-1}B$ by performing a scale operation $b_i^T = b_i^T / \lambda_{ii}$. Then the same row is used to update the rest of the B by performing a rank-1 update.

Basic TRSM ($n_r \times n_r$): Figure 5.4 illustrates the mapping of an unblocked down-looking TRSM algorithm for a $n_r \times n_r$ sub-matrix of B and lower triangular $n_r \times n_r$ diagonal sub-matrix of L, both stored in the registers of the LAC (with $n_r \times n_r$ PEs). The LAC is augmented with a reciprocal unit $f(x) = 1/x$, implementation details of which are discussed in Section A.3.2. In each iteration $i = 0, \ldots, n_r - 1$, the algorithm performs three steps, $S1$ through $S3$, where the figure shows the second such iteration ($i = 1$). In $S1$ and $S2$, the element $\lambda_{i,i}$ of L in PE(i,i) is updated with its inverse. The result is broadcast within the ith PE row and used to multiply into the elements of the corresponding row of matrix B (effectively dividing row elements of B by $\lambda_{i,i}$). In $S3$, the results of those computations are broadcast within their respective columns to be multiplied by the corresponding column of L (which is broadcast within the respective rows) in order to perform a rank-1 update that subtracts the result of this multiplication from the remaining lower part of matrix B. This completes the current iteration, which is repeated for $i = 0, \ldots, n_r - 1$. Given a MAC unit with p pipeline stages, this $n_r \times n_r$ TRSM takes $2pn_r$ cycles. Due to the data dependencies between different PEs within and between iterations, each element has to go through p stages of MAC units while other stages are idle.

Stacked TRSM ($n_r \times pn_r$): Careful examination of the mapping of the $n_r \times n_r$ TRSM above exposes that a lot of cycles are wasted. Given current floating-point unit designs, fine grain data dependencies keep all but one of the stages of the FPU pipeline idle. To overcome this inefficiency, we can stack several successive $n_r \times n_r$ TRSM operations and thus fill the empty slots of the FPU pipeline. With a p stage pipelined FPU design, the LAC can finish the computation of p blocks in almost the same amount of time as a single $n_r \times n_r$ TRSM.

This solution is illustrated in Figure 5.5. In each iteration, p multiplication operations (colored yellow) and p rank-1 updates (colored blue) on the target elements of B are pipelined through each FPU. The stacked solution for TRSM of a $n_r \times pn_r$ panel of B (distributed among the local stores) takes

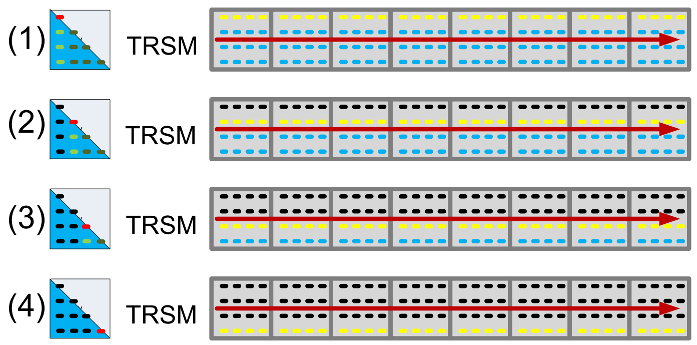


Figure 5.5: Overcoming the data dependency by pipelining TRSM operations. Eight blocks of 4×4 TRSMs are stacked in each of the four iterations to fill empty slots of an eight stage pipelined MAC unit.

approximately $2pn_r + p$ cycles. All the intermediate values are stored in actual pipeline registers of the MAC units and therefore no extra temporary registers are required. However, this solution still does not fully utilize the resources.

Software pipelined TRSM ($n_r \times gpn_r$): The previous solution filled the empty pipeline stages of the floating-point units. However, dependencies existing within each iteration of the stacked TRSM allow only one row out of the available n_r row of PEs to be utilized for computing $b_1^T := x_1^T = b_1^T / \lambda_{11}$ (step $S2$) while the other PEs are waiting for the result to perform the $B_2 := X_2 = B_2 - l_{21} b_1^T$ ($S3$) computation (rank-1 update). In cases where B panels are relatively large, we can adopt software pipelining techniques to overcome this inefficiency. The wide panel of B is blocked into smaller panels (sub-panels). The solution is shown in Figure 5.6, where the panel of B is blocked into four smaller sub-panels. Within each iteration, the result of a stacked TRSM for $b_1^T := b_1^T / \lambda_{11}$ in one sub-panel will be used to update $B_2 := B_2 - l_{21} b_1^T$ simulta-

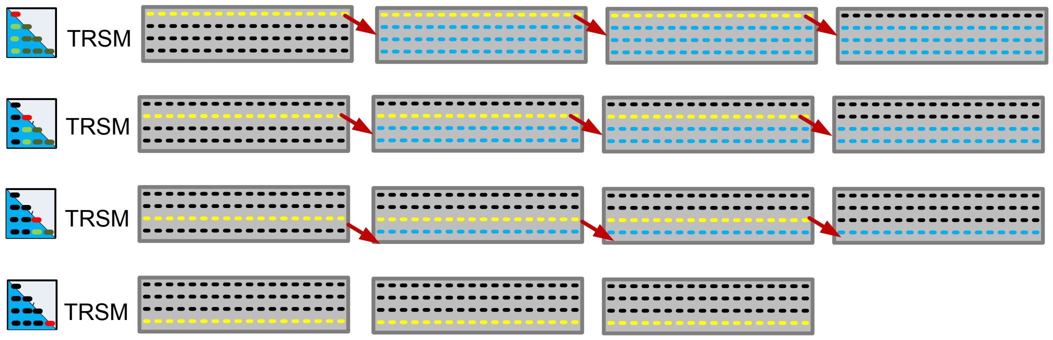


Figure 5.6: TRSM operation mapping on LAC, increasing utilization by software pipelining four stacked TRSM operations.

neously with computing the next set of $b_1^T := b_1^T/\lambda_{11}$ in the following sub-panel. Hence, within each iteration, we can overlap multiplication updates (yellow) with rank-1 updates (blue) by pipelining and hence simultaneously working on different sub-panels.

This solution further improves the utilization and almost doubles the speed of computation. The software pipelined solution for stacked TRSM takes $p(n_r(g+1))$ cycles for a $n_r \times gpn_r$ panel of B. This solution only reorders the operations between stacked TRSM calls and therefore does not need extra storage. The utilization of this operation can be estimated as $\frac{1}{n_r}(1 + \frac{(g-1)(n_r+1)}{2} + \frac{n_r}{2}\frac{n_r-1}{n_r})/(g+1) = \frac{g(n_r+1)}{2(g+1)n_r} \times 100\% \simeq 60\%$, where $n_r = 4$. However, what we see next is that this is not where most computation happens when performing a larger TRSM operation.

5.3.2 Blocked TRSM on LAC

For larger matrices and to get higher performance, a blocked algorithm is employed that casts most computation in terms of GEMM operations. Only the updates by diagonal blocks of L use the lower-order unblocked TRSM. We show how the blocked algorithm is derived. Partition

$$X = \left(\begin{array}{c} X_0 \\ \hline X_1 \\ \hline X_2 \end{array}\right) \quad , \quad B = \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right) \quad , \text{ and } \quad L = \left(\begin{array}{c|c|c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right).$$

Then $LX = B$ means that

$$B = \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right) = \left(\begin{array}{c} L_{00}X_0 \\ \hline L_{10}X_0 + L_{11}X_1 \\ \hline L_{20}X_0 + L_{21}X_1 + L_{22}X_2 \end{array}\right).$$

Now, if the computation has progressed so that X_0 has already been computed, then X_1 can be computed by solving $X_1 = L_{11}^{-1}(B_1 - L_{10}X_0)$, requiring a matrix-matrix multiply followed by a small triangular solve. This motivates the operations in the blocked variant. This blocked algorithm computes on a matrix that fits in the LAC local memory while the computation $B_1 := L_{11}^{-1}B_1$ is performed by the unblocked variants on the LAC.

Let us now assume that a larger matrices B of size $kn_r \times mn_r$ and L of size $kn_r \times kn_r$ are distributed among the PE local stores of the LAC in a 2-D round-robin fashion much like the matrix was for GEMM in Chapter 3. We now describe how a single iteration of the blocked TRSM algorithm is performed by the LAC. In Figure 5.7 we show the case where $k = 8$. Highlighted is the data involved in the fifth iteration. We describe the different operations to be performed:

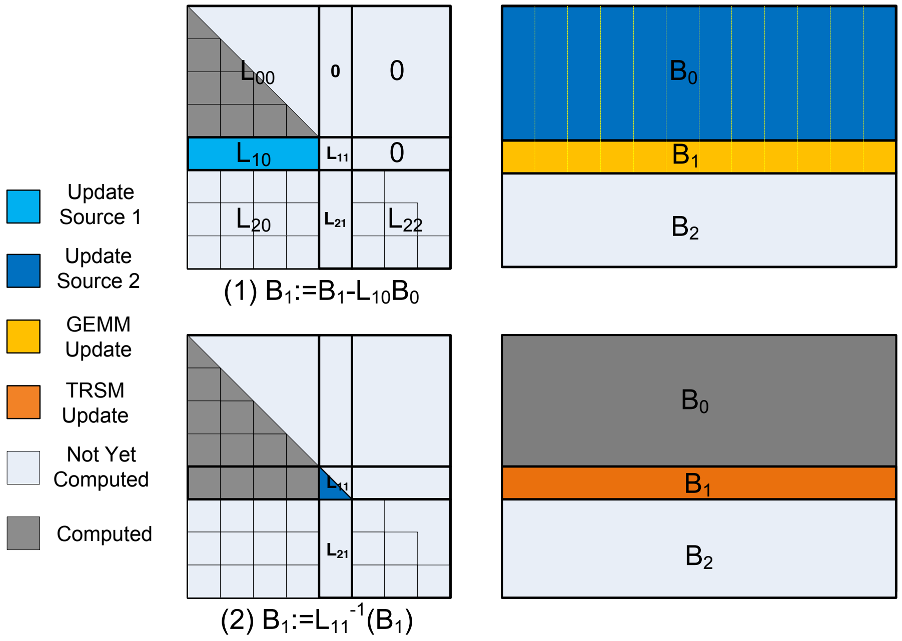


Figure 5.7: Blocked TRSM, fifth iteration.

(1) $B_1 := B_1 - L_{10}B_0$**:** Blocks of B_1 are moved in and out of the accumulators of the PEs. This is a matrix multiplication, which is orchestrated like similar operations were for GEMM.

(2) $B_1 := L_{11}^{-1}B_1$**:** The block of L_{11} is moved to the registers of the PEs. An unblocked (either stacked or software pipelined) TRSM operation is performed on B_1 as described before.

We chose this algorithmic variant primarily because it exhibits better data locality. In this algorithm, the data above the current row panel being computed, which represents the bulk of data being processed, is read but needs not be written. The alternative would be an algorithm in which the blocks below the current row panel are updated, meaning that they need to be read and written. In other words, in the chosen algorithm a block is repeatedly updated by data that can be streamed, enhancing both temporal and spatial locality. This is also of advantage when mapping even larger matrices to the LAC, where data must be brought in from higher levels of memory. The important insight is that by designing the algorithm and architecture hand-in-hand, the appropriate algorithm can guide the hardware design and vice versa.

Computations with large matrices do not fit in the aggregate PE local memories of the LAC. Hence, we need another level of blocking. This results in a hierarchical TRSM that is composed of two main smaller kernels, where each of them fits in the LAC local store. This is simply another level of blocking. The question of scheduling and locality needs to be answered again at this scale. Shared on-chip memory and multiple LACs pose more challenges in efficiency and utilization of the available resources and parallelism. Also, there are more options for the granularity with which we move blocks of data.

5.3.3 Performance Analysis

It is clear that if one considers dynamic bandwidth, the $kn_r \times gpn_r$ TRSM kernel needs more bandwidth in its earlier stages. We can compute the maximum bandwidth demand of TRSM by computing the bandwidth demands of a TRSM factorization in which computation and communication are overlapped. This includes fetching the first row panel of the input matrix to perform a $n_r \times mn_r$ $(m = gp)$ TRSM operations on it and then updating the next row panel with its result. Now, $n_r^2 \times gpn_r$ operations are performed that take $gpn_r + p$ cycles. Hence, the bandwidth demand for updating could reach up to $(gpn_r^2)/(gpn_r + p) \simeq n_r$ elements per cycle. This is the maximum bandwidth that TRSM requires in order to overlap computation and communication without prefetching while updating B.

The required average bandwidth demand is much lower since more calculations need to be performed per fetching of subsequent row panels of B. The average bandwidth demand can be calculated as the ratio of total computations to total communications. The total communication for B includes bringing B in and out of the LAC, i.e. $2kn_rgpn_r$ values. The total number of cycles for computation is $\sum_{i=0}^{k}(ig + g + 1)pn_r$. Hence, the average bandwidth is

$$\frac{2kn_rgpn_r}{\sum_{i=0}^{k}(ig + g + 1)pn_r} \simeq \frac{2kn_rgpn_r}{0.5k^2gpn_r} \leq \frac{4n_r}{k}.$$

To calculate the utilization, we have to measure the total computation time/cycles of all GEMM and TRSM suboperations in all iterations and divide them by the total amount of actual useful computations that are performed. The total number of cycles can be estimated as

$$\sum_{i=0}^{k} n_r(in_r)(gpn_r)/(n_r^2) + (g + 1)pn_r = \sum_{i=0}^{k}(ig + g + 1)pn_r.$$

The total number of MAC operations for this operation can be computed as $\sum_{i=0}^{k} n_r(gn_r)(gpn_r) + gpn_r(n_r^2)/2$, which, at 100% utilization, should take $\sum_{i=0}^{k}(i + 1/2)(gpn_r)$ cycles. Hence, the utilization can be calculated as:

$$\frac{\sum_{i=0}^{k}(i+1/2)(gpn_r)}{\sum_{i=0}^{k}(ig+g+1)pn_r} \simeq \frac{\sum_{i=0}^{k}(i+1/2)(gpn_r)}{\sum_{i=0}^{k}(i+1)(gpn_r)} = \frac{\sum_{i=0}^{k}(i+1/2)}{\sum_{i=0}^{k}(i+1)}.$$

In our analysis, we assumed (conservatively) that the utilization of the software pipelined TRSM operation is 50%. This is less than what we estimated previously in Section 5.3.1, especially for large gs. Still, according to the above estimation, the utilization number for a 32×128 TRSM operation becomes 90%.

5.4 Results

Details of analytical performance models and LAC operation for GEMM were described in Chapters 3 and 4. We derived similar models for the other level-3 BLAS operations. Figures 5.8, and 5.9 report performance results of a single core as a function of the size of the local memory and the bandwidth to the on-chip memory for TRSM and SYRK, respectively. Here, we use $n_r \in \{4, 8\}$, $m_c = k_c$ (this determines the size of the blocks that are mapped to the LAC), and $n = 512$. These graphs demonstrate the fundamental trade-off between bandwidth to the external memory and the size of the local memory, which in itself is a function of the kernel size (k_c, m_c, and n_r). Performance is either limited by under-utilization in some parts of an operation or by limitations of the off-core bandwidth.

The GEMM operation typically achieves the best utilization and hence

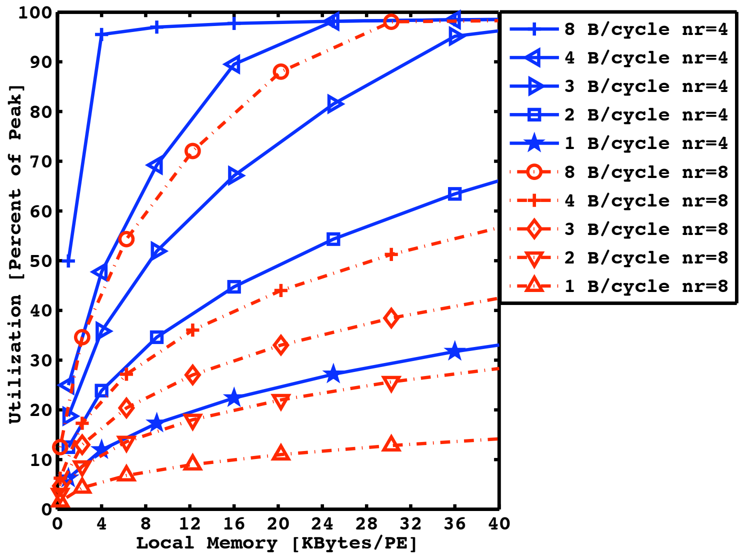


Figure 5.8: Estimated core performance for SYRK as a function of the bandwidth between LAC and on-chip memory, and the size of local memory with $n_r = 4$ and $n_r = 8$, $m_c = k_c = 256$.

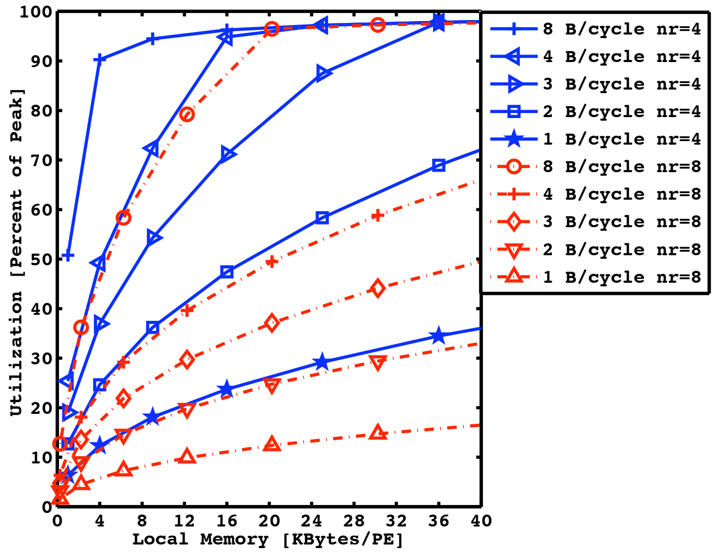


Figure 5.9: Estimated core performance for TRSM as a function of the bandwidth between LAC and on-chip memory, and the size of local memory with $n_r = 4$ and $n_r = 8$, $m_c = k_c = 256$.

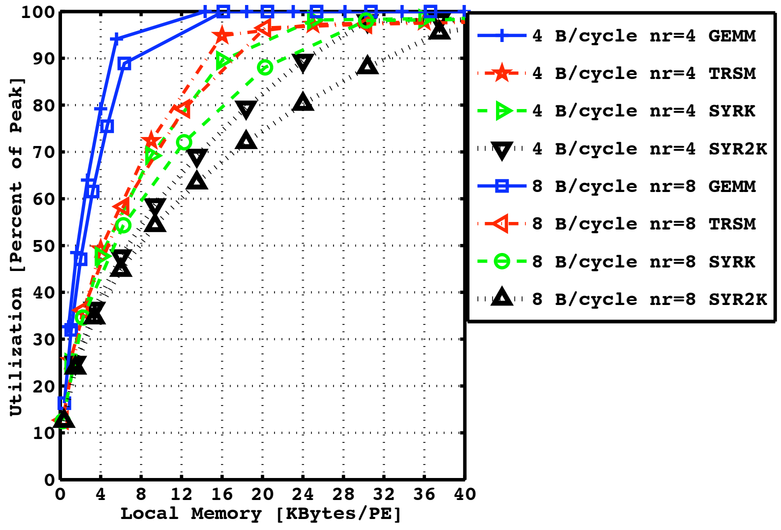


Figure 5.10: Utilizations for representative level-3 BLAS operations for $n_r = 4$.

performance among all other level-3 BLAS. Figure 5.10 shows a comparison between selected curves from Figures 5.8, 5.9, and 3.4 for $n_r \in \{4, 8\}$, $m_c = k_c$ and $n = 512$. We can observe that for a PE memory size of 20 KBytes and off-core memory bandwidth of 4 B/cycles, GEMM, TRSM, SYRK, and SYR2K achieve 100%, 95%, 90%, and 85% utilization, respectively.

The LAC shows utilizations for TRSM, SYRK, and SYR2K that are close to what GEMM can achieve. The reason why none of the other operations reach 100% utilization is that their basic operations do not fully utilize all the PEs. This is due to the triangular shape of the diagonal blocks in each of these cases. However, since lower-order terms only form a fraction of all computations, the overall performance approaches the peak as the size of problem grows.

TRSM achieves better performance for smaller problem sizes, even

Algorithm	$\frac{\text{W}}{\text{mm}^2}$	$\frac{\text{GFLOPS}}{\text{mm}^2}$	$\frac{\text{GFLOPS}}{\text{W}}$	Utilization
GEMM $n_r = 4$	0.397	21.61	54.4	100%
TRSM $n_r = 4$	0.377	20.53	51.7	95%
SYRK $n_r = 4$	0.357	19.45	49.0	90%
SYR2K $n_r = 4$	0.314	17.07	43.0	79%
GEMM $n_r = 8$	0.397	21.61	54.4	100%
TRSM $n_r = 8$	0.377	20.53	51.7	95%
SYRK $n_r = 8$	0.346	18.80	47.3	87%
SYR2K $n_r = 8$	0.290	15.77	39.7	73%

Table 5.1: LAC efficiency for level-3 BLAS algorithms at 1.1 GHz.

though the computation of the triangular part of the lower order term of TRSM is less efficient than SYRK. The difference between SYRK and TRSM is in the bandwidth demand. SYRK needs more bandwidth than TRSM for the same problem size. In small problems, the amount of bandwidth directly affects the performance and results in a higher utilization for TRSM. By contrast, SYRK has higher utilization of the lower order term and better performance in bigger problem sizes. For example, with 25 Kbytes of local memory per PE, SYRK with 98% utilization overtakes TRSM with 96% utilization.

SYR2K performs worse than SYRK as is expected for this operation. For the same PE memory size, only a smaller SYR2K operation can be mapped on the LAC. A typical level-3 BLAS has $O(n^2)$ communication and $O(n^3)$ computation complexity. The SYR2K operation doubles the amount of communication and computation, which is not bandwidth efficient compared to solving a bigger SYRK problem.

Since GEMM results in the highest utilization and load, we used access patterns of the GEMM algorithm obtained from our simulator to estimate

SRAM power consumption for the rest of level-3 BLAS operations. Table 5.4 summarizes detailed performance and area efficiencies of the LAC for all presented level-3 BLAS operations at 1.1 GHz.

5.5 Summary

In this chapter, we discussed generalization of the LAC architecture to support level-3 BLAS operations. We presented an overview of representative level-3 BLAS operations and picked SYRK and TRSM to show algorithm/architecture for them on LAC. We utilized the 2D architecture of the LAC to optimally map SRYK. Furthermore, we showed mapping of TRSM across layers of the memory hierarchy. The results that we presented conclude that with minimal modifications, this architecture can achieve high efficiency across all level-3 BLAS.

Chapter 6

Generalization Beyond Level-3 BLAS

The next step towards generalization is moving towards algorithms with more irregularities not only in the domain of linear algebra, but also in the signal processing domain. This allows us to further examine the capacities of our codesign methodology and the flexibility of our proposed architecture. In this chapter, we first discuss matrix factorization algorithms and their requirements. Details of each algorithm and its mapping are presented in Appendix A. Here, we only summarize the modifications to some components in the architecture and the resulting design metrics.

We then go beyond the linear algebra domain by exploring FFTs, a completely different set of algorithms with very different behavior. We show them to also be suitable for the baseline LAC architecture (details can be found in Appendix B). We exploit similarities between the original LAC and the FFT-optimized design to introduce a flexible, hybrid architecture that can perform both of these applications efficiently. Comparing both full-custom designs with our proposed hybrid core, we demonstrate the costs of flexibility versus efficiency.

6.1 Matrix Factorizations

Matrix factorizations are the natural next step towards making our architecture more flexible. We choose three matrix factorization algorithms: Cholesky, LU (with partial pivoting), and QR factorization [107]. These algorithms are typically the first (and most compute intensive) step towards the solution of linear systems of equations or linear least-squares problems [48], which have applicability in scientific computing applications. These operations also solve more complex kernels like Kalman filters [145], and updating and downdating algorithms [136].

Many current solutions use heterogeneous computing for more complicated algorithms like Cholesky, QR, and LU factorization [7, 143]. Often, only the most parallelizable and simplest parts of these algorithms, that exhibit ample parallelism, are performed on accelerators. Specifically, the part of the computations that can be cast in terms of level-3 BLAS operations is typically mapped to the accelerators. Other more complex parts, which are added to the algorithm to overcome floating-point limitations or which would require complex hardware to exploit fine grain parallelism, are off-loaded to a general-purpose host processor. Unlike traditional heterogeneous solutions, we aim to support such complex kernels directly on our LAP. To do so, we focus only on the inner kernels of these algorithms. Problems with larger sizes are cast into level-3 BLAS operations and these the inner kernels.

Quality implementations of these algorithms aim to ensure numerical stability and try to prevent spurious overflow and underflow errors. As a result, algorithms become more complex leading to inherent overhead. The question we pursue is how to accommodate such complexities when mapping these algorithms onto accelerators and/or into custom hardware. Our solution is to avoid the inefficiencies caused by limitations in current architectures. We show that by adding minimal logic, we can overcome corresponding complexities.

We show the challenges and limitations in current architectures to perform these matrix factorizations efficiently. We propose a new solution that tries to avoid all inefficiencies caused by limitations in current architectures and thereby overcomes the complexities in matrix factorization algorithms themselves. The problem is that architecture designers typically only have a high-level understanding of algorithms, while algorithm designers try to optimize for already existing architectures. We specifically focus on the design of the floating-point units to satisfy algorithm requirements. Our solution allows architectural changes to the design in order to reduce complexity directly in the algorithm whenever possible. Thus, the solution is to exploit algorithm/architecture codesign.

A linear system of equation, is represented as $Ax = b$, where $n \times n$ matrix A and vector b are known, and x is the solution vector that we wish to compute. This system is often solved by first factoring matrix A. As the most generic example, LU factorization can be exploited to compute L and U such that $A \rightarrow LU$, where L is a lower triangular matrix $L \in \mathbb{R}^{n \times n}$ and U is an upper triangular matrix $U \in \mathbb{R}^{n \times n}$. After computing L and U, two triangular solves with single right hand side (similar to TRSM in Chapter 5) are performed, which are known as forward substitution and backward substitution. Forward substitution solves $Ly = b$ for y and backward substitution then solves $Ux = y$ for x.

6.1.1 Cholesky Factorization

Cholesky factorization is the most straight-forward factorization operation. Given a Symmetric Positive Definite (SPD) matrix[1], $A \in \mathbb{R}^{n\times n}$, the Cholesky factorization produces a lower triangular matrix, $L \in \mathbb{R}^{n\times n}$ such that $A = LL^T$.

We start by deriving the algorithm. Partition

$$A = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) \quad \text{and} \quad L = \left(\begin{array}{c|c} \lambda_{11} & 0 \\ \hline l_{21} & L_{22} \end{array}\right),$$

where α_{11} and λ_{11} are scalars. Then $A = LL^T$ means that,

$$\left(\begin{array}{c|c} \alpha_{11} & \alpha_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} \lambda_{11}^2 & * \\ \hline \lambda_{11} l_{21} & L_{22}L_{22}^T + l_{21}l_{21}^T \end{array}\right)$$

which in turn means that

$$\begin{array}{c|c} \alpha_{11} = \lambda_{11}^2 & \star \\ \hline a_{21} = \lambda_{11} l_{21} & A_{22} - l_{21}l_{12}^T = L_{22}L_{22}^T \end{array}.$$

We can compute L from A via the operations

$$\begin{array}{c|c} \alpha_{11} := \lambda_{11} = \sqrt{\alpha_{11}} & \star \\ \hline a_{21} := l_{21} = (1/\lambda_{11}) a_{21} & A_{22} := L_{22} = \mathrm{Chol}(A_{22} - l_{21}l_{12}^T) \end{array},$$

[1]A condition required to ensure that the square root of a non-positive number is never encountered.

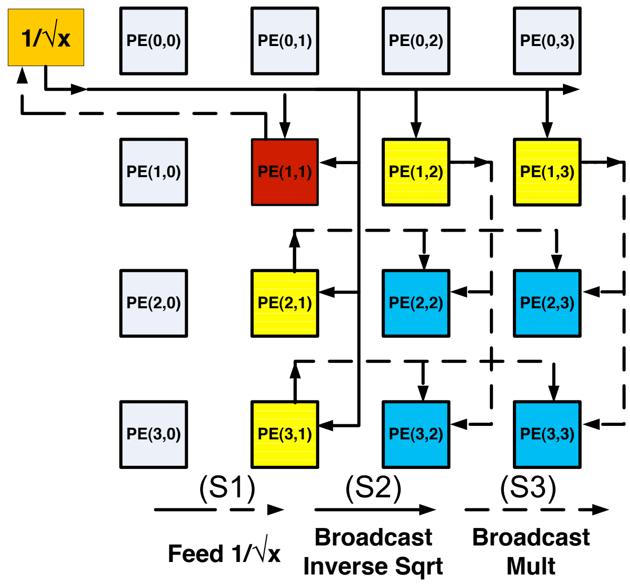


Figure 6.1: 4x4 Cholesky decomposition mapping on LAC, 2nd iteration.

overwriting A with L. For high performance, it is beneficial to also derive a blocked algorithm that casts most computations in terms of matrix-matrix operations, but we will not need these in our discussion. The observation is that the "square-root-and-reciprocal" operation $\alpha_{11} := \sqrt{\alpha_{11}};\ t = 1/\alpha_{11}$ is important, and that it should therefore be beneficial to augment the microarchitecture with a unit that computes $f(x) = 1/\sqrt{x}$ when mapping the Cholesky factorization onto the LAC.

We now focus on how to factor a $n_r \times n_r$ submatrix when stored in the registers of the LAC (with $n_r \times n_r$ PEs). In Figure 6.1, we show the second iteration of the algorithm. For this subproblem, the matrix has also been copied to the upper triangular part, which simplifies the design.

In each iteration, $i = 0, \ldots, n_r - 1$, the algorithm performs three steps

$S1$ through $S3$. In $S1$, the inverse-square-root is computed. In $S2$, the element in PE(i,i) is updated with its inverse square root. The result is broadcast within the ith PE row and ith PE column. It is then multiplied into all elements of the column and row which are below and to the right of PE(i,i). In $S3$, the results of these computations are broadcast within the columns and rows to be multiplied by each other as part of a rank-1 update of the remaining part of matrix A. This completes one iteration, which is repeated for $i = 0, \ldots, n_r - 1$.

Given a MAC unit with p pipeline stages and an inverse square root unit with q stages, this $n_r \times n_r$ Cholesky factorization takes $2p(n_r - 1) + q(n_r)$ cycles. Due to the data dependencies between different PEs within and between iterations, each element has to go through p stages of MAC units while other stages are idle. The last iteration only replaces the PE($n_r - 1$,$n_r - 1$) value by its square root, which only requires q additional cycles.

Clearly, there are a lot of dependencies and there will be wasted cycles. However, this smaller subproblem is not where most computations happen when performing a larger Cholesky factorization. For this reason, we do not discuss details of how to fully optimize the LAC for this operation here.

The important idea is that by introducing an inverse square-root unit, that operation needs not to be performed on a host nor in software or emulation on the LAC, which yields a substantial savings in cycles.

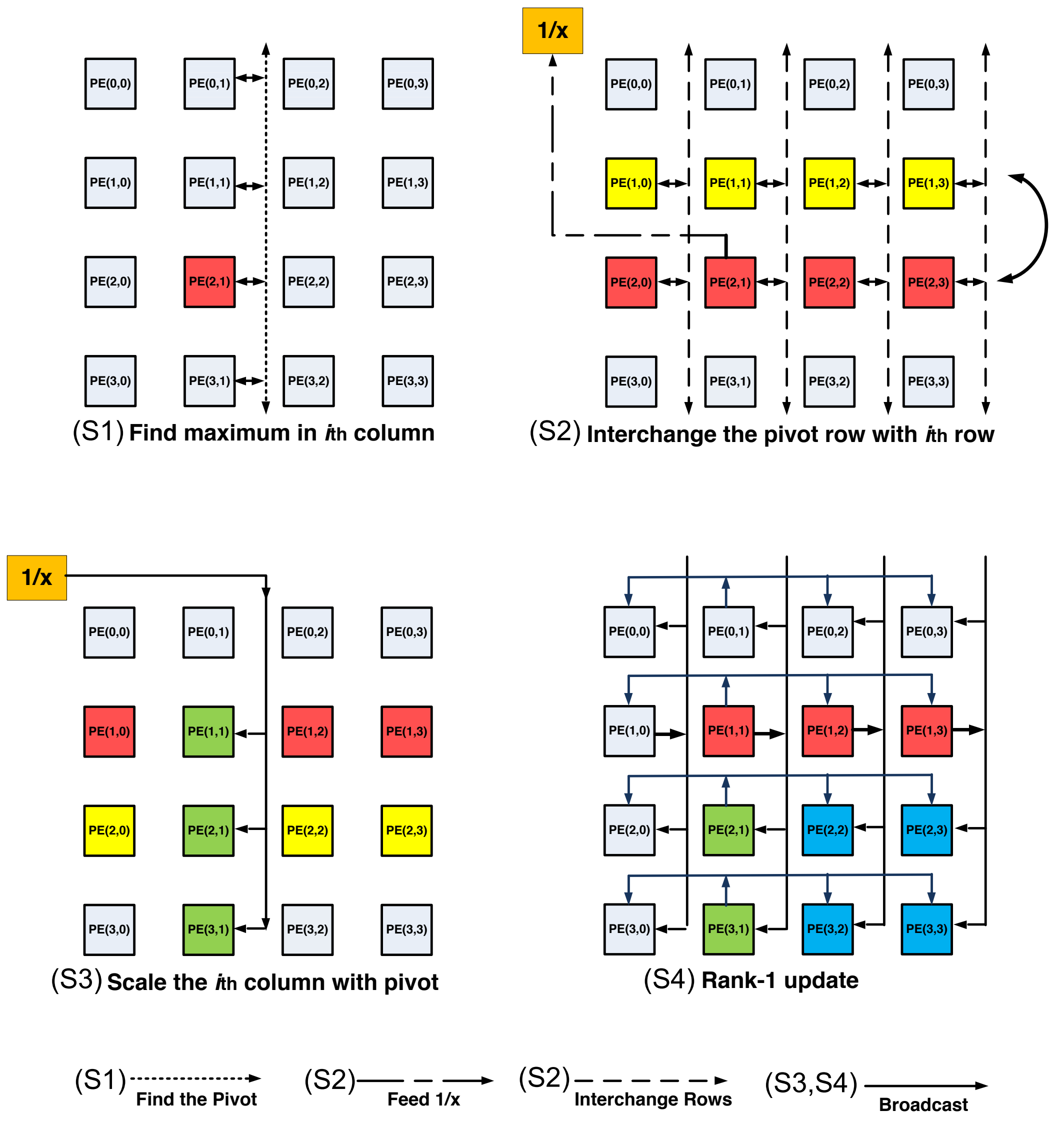


Figure 6.2: Second iteration of a $K \times n_r$ LU factorization with partial pivoting on the LAC.

6.1.2 LU Factorization with Partial Pivoting

LU factorization with partial pivoting is a more general solution for decomposing matrices. The LU factorization of a square matrix A is the first and most computationally intensive step towards solving $Ax = b$. It decomposes a matrix A into a unit lower-triangular matrix L and an upper-triangular matrix U such that $A = LU$.

We again briefly motivate the algorithm that we utilize: partition

$$A = \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right), L = \left(\begin{array}{c|c} 1 & 0 \\ \hline l_{21} & L_{22} \end{array}\right), U = \left(\begin{array}{c|c} \upsilon_{11} & u_{12}^T \\ \hline 0 & U_{22} \end{array}\right),$$

where α_{11}, and υ_{11} are scalars. Then $A = LU$ means that

$$\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} \upsilon_{11} & u_{12}^T \\ \hline l_{21}\upsilon_{11} & L_{22}U_{22} + l_{21}u_{12}^T \end{array}\right)$$

so that

$$\begin{array}{c|c} \alpha_{11} = \upsilon_{11} & a_{12}^T = u_{12}^T \\ \hline a_{21} = \upsilon_{11} l_{21} & A_{22} - l_{21}u_{12}^T = L_{22}U_{22} \end{array}.$$

We can thus compute L and U in place for matrix A. The diagonal elements of L are not stored (all of them are ones). The strictly lower triangular part of A is replaced by L. The upper triangular part of A, including its diagonal elements, is replaced by U as follows:

$$\begin{array}{c|c} \alpha_{11} := \upsilon_{11} \text{ (no-op)} & a_{12}^T := u_{12}^T \text{ (no-op)} \\ \hline a_{21} := l_{21} = a_{21}/\upsilon_{11} & A_{22} := \mathrm{LU}(A_{22} - l_{21}u_{12}^T) \end{array}.$$

Again, we do not need the blocked version of this algorithm for the discussion in this document.

In practice, the use of finite precision arithmetic yields this naive algorithm for numerical accuracy reasons: the update to matrix A in the first iteration is given by

$$\left(\begin{array}{c|ccc}
\alpha_{11} & \alpha_{12} & \cdots & \alpha_{1,n} \\ \hline
0 & \alpha_{22} - \lambda_{21}\alpha_{12} & \cdots & \alpha_{2,n} - \lambda_{21}\alpha_{1,n} \\
0 & \alpha_{32} - \lambda_{31}\alpha_{12} & \cdots & \alpha_{3,n} - \lambda_{31}\alpha_{1,n} \\
\vdots & \vdots & \ddots & \vdots \\
0 & \alpha_{n,2} - \lambda_{n,1}\alpha_{12} & \cdots & \alpha_{n,n} - \lambda_{n,1}\alpha_{1,n}
\end{array}\right),$$

where $\lambda_{i,1} = \alpha_{i,1}/\alpha_{11}$, $2 \leq i \leq n$. The algorithm clearly fails if $\alpha_{11} = 0$. If $\alpha_{11} \neq 0$ and $|\alpha_{i,1}| \gg |\alpha_{11}|$, then $\lambda_{i,1}$ will be large in magnitude and it can happen that for some i and j the value $|\alpha_{i,j} - \lambda_{i,1}\alpha_{i,j}| \gg |\alpha_{i,j}|$, $2 \leq j \leq n$; that is, the update greatly increases the magnitude of $\alpha_{i,j}$. This is a phenomenon known as large element growth and leads to numerical instability. The problem of element growth can be solved by rearranging (pivoting) the rows of the matrix (as the computation unfolds). Specifically, the first column of matrix A is searched for the largest element in magnitude. The row that contains such element, the *pivot row*, is swapped with the first row, after which the current step of the LU factorization proceeds. The net effect is that $|\lambda_{i,1}| \leq 1$ so that $|\alpha_{i,j} - \lambda_{i,1}\alpha_{1,j}|$ is of a magnitude comparable to the largest of $|\alpha_{i,j}|$ and $|\alpha_{1,j}|$, thus keeping element growth bounded. This is known as the *LU factorization with partial pivoting*. The observation is that finding the (index of the) largest value in magnitude in a vector is important for this operation. For practical purposes, LU factorization with partial pivoting is numerically stable.

To study opportunities for corresponding architecture extensions, we focus on how to factor a $kn_r \times n_r$ submatrix (see Figure 6.3) stored in a 2D round-robin fashion in the local store and registers of the LAC (with $n_r \times$

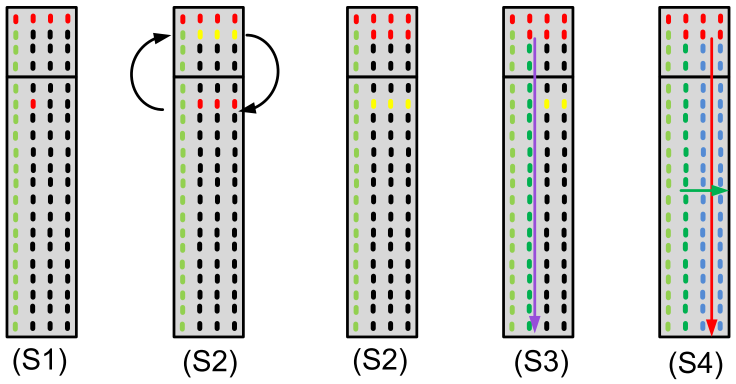


Figure 6.3: Operations and data manipulation in the second iteration of a $k \times n_r$ LU factorization inner kernel.

n_r PEs). In Figure 6.2, we show the second iteration of the right-looking unblocked algorithm ($i = 1$).

In each iteration, $i = 0, \ldots, n_r - 1$, the algorithm performs four steps, S1 through S4. In S1, the elements in the ith column below the diagonal are searched for the maximum element in magnitude. Note that this element can be in any of the ith column's PEs. Here, we just assume that it is in the row with $j = 2$. After the row with maximum value (the pivot row) is found, in S2, the pivot value is sent to the reciprocal ($1/x$) unit and the pivot row is swapped with the diagonal (ith) row concurrently. In S3, the reciprocal ($1/x$) is broadcast within the ith column and multiplied into the elements below PE(i,i). In S4, the results of the division (in the ith column) are broadcast within the rows. Simultaneously, the values in the ith (pivot) row to the right of the ith column are broadcast within the columns. These values are multiplied as part of a rank-1 update of the remaining part of matrix A. This completes the current iteration, which is repeated for $i = 0, \ldots, n_r - 1$.

According to the above mapping, most of the operations are cast as rank-1 updates and multiplications that are already provided in the existing LAC architecture. In addition to these operations, two other essential computations are required: first, a series of floating-point comparisons to find the maximal value in a vector (column); and second, a reciprocal ($1/x$) operation needed to scale the values in the ith column by the pivot. Due to these extra complexities, most existing accelerators send the whole $kn_r \times n_r$ block to a host processor to avoid performing the factorization themselves [7, 143]. By contrast, we will discuss a small set of extensions that will allow us to efficiently perform all needed operations and hence the complete LU factorization within dedicated hardware.

6.1.3 QR Factorization and Vector Norm

Householder QR factorization is often used when solving a linear least-squares problem. QR factorization decomposes a matrix $A \in \mathbb{R}^{m\times n}(m \geq n)$ into a orthonormal matrix matrix $Q \in \mathbb{R}^{m\times n}$ and an upper-triangular matrix $R \in \mathbb{R}^{n\times n}$ such that $A = QR$. The key to practical QR factorization algorithms is the Householder transformation. Given $u \neq 0 \in \mathbb{R}^n$, the matrix $H = I - uu^T/\tau$ is a reflector or Householder transformation if $\tau = u^T u/2$. In practice, u is scaled so that its first element is “1”. We will now show how to compute $A \rightarrow QR$, the QR factorization, of $m \times n$ matrix A as a sequence of Householder transformations applied to A.

In the first iteration, we partition $A \rightarrow \left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right)$. Let $\begin{pmatrix} 1 \\ u_1 \end{pmatrix}$ and

Algorithm: $\left[\begin{pmatrix}\rho_1\\ \hline u_2\end{pmatrix}, \tau_1\right] = \text{HOUSEV}\begin{pmatrix}\alpha_1\\ \hline a_{21}\end{pmatrix}$	
	$\chi_2 := \|a_{21}\|_2$
	$\alpha := \left\|\begin{pmatrix}\alpha_1\\ \chi_2\end{pmatrix}\right\|_2 (= \|x\|_2)$
$\rho_1 = -\text{sign}(\alpha_1)\|x\|_2$	$\rho_1 := -\text{sign}(\alpha_1)\alpha$
$\nu_1 = \alpha_1 + \text{sign}(\alpha_1)\|x\|_2$	$\nu_1 := \alpha_1 - \rho_1$
$u_2 = a_{21}/\nu_1$	$u_2 := a_{21}/\nu_1$
	$\chi_2 = \chi_2/\|\nu_1\|(= \|u_2\|_2)$
$\tau_1 = (1 + u_2^T u_2)/2$	$\tau_1 = (1 + \chi_2^2)/2$

Table 6.1: Computing the Householder transformation. Left: simple formulation. Right: efficient computation.

τ_1 define the Householder transformation that zeroes a_{21} when applied to the first column. Then, applying this Householder transform to A yields:

$$\left(\begin{array}{c|c}\alpha_{11} & a_{12}^T\\ \hline a_{21} & A_{22}\end{array}\right) := \left(I - \begin{pmatrix}1\\ \hline u_2\end{pmatrix}\begin{pmatrix}1\\ \hline u_2\end{pmatrix}^T/\tau_1\right)\left(\begin{array}{c|c}\alpha_{11} & a_{12}^T\\ \hline a_{21} & A_{22}\end{array}\right)$$
$$= \left(\begin{array}{c|c}\rho_{11} & a_{12}^T - w_{12}^T\\ \hline 0 & A_{22} - u_{21}w_{12}^T\end{array}\right),$$

where $w_{12}^T = (a_{12}^T + u_{21}^T A_{22})/\tau_1$. Computation of a full QR factorization of A will now proceed with submatrix A_{22}.

The new complexity introduced in this algorithm is in the computation of u_2, τ_1, and ρ_1 from α_{11} and a_{21}, captured in Table 6.1, which require a vector-norm computation and scaling (division). This is referred to as the computation of the Householder vector. We first focus on the computation of the vector norm.

The 2-norm of a vector x with elements $\chi_0, \cdots, \chi_{n-1}$ is given by $\|x\| := (\sum_{i=0}^{n} |\chi_i|^2)^{1/2} = \sqrt{\chi_0^2 + \chi_2^2 + \ldots + \chi_{n-1}^2}$. The problem is that intermediate values can overflow or underflow. This is avoided by normalizing x and performing the following operations instead.

$$t = \max_{i=0}^{n-1} |x_i|\,; \quad y = x/t; \quad \|x\|_2 := t \times \|y\|_2.$$

If not for overflow and underflow, the operation would be no more complex than an inner product followed by a square root. To avoid overflow and underflow, the maximum of all inputs must be found, the vector be normalized, and an extra multiplication is needed to scale the result back. As such, with the exception of the mentioned normalization and the introduction of a matrix-vector multiplication, the overall mapping of QR factorization to the LAC is similar to that of LU factorization. Due to space reasons, we focus our following discussions on the computation of this Householder vector and vector norm only.

To compute the vector norm, two passes over the data should be performed: a first pass to search and find the largest value in magnitude followed by a second pass to scale the vector elements and accumulate the inner-product. In addition to being slow, "this algorithm also involves more rounding errors than the unscaled evaluation, which could be obviated by scaling by a power of the machine base [58]". A one-pass algorithm has been presented in [19]. It uses three accumulators for different value sizes. This algorithm avoids overflow and underflow. However, it still needs to perform division. More details about how this is computed in software are discussed in [58, 85].

We now focus on how to perform a vector norm of a scaled $kn_r \times 1$ vector (see Figure 6.4) when stored in the local store and registers of the LAC

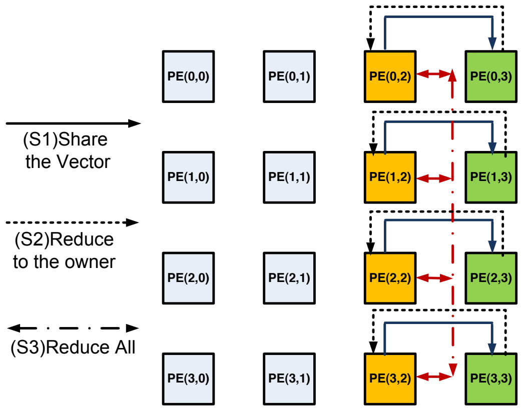


Figure 6.4: Mapping of the Vector Norm operation of a single vector stored in the third column of the LAC.

(with $n_r \times n_r$ PEs). Recall that such a column is only stored in one column of the LAC. In Figure 6.4, we show the iterations for calculating a vector norm that is stored in the 3rd column of PEs. The algorithm performs three steps, $S1$ through $S3$.

In $S1$, the third column of PEs starts computing the inner product with half of the vector elements. Simultaneously, the PEs in this row share the elements of the other half of the vector with the adjacent PEs in the next column (fourth column in Figure 6.4). PEs in the adjacent column also start performing inner products. After all the PEs in both columns have computed their parts, in $S2$ the partial inner products are reduced back into the original LAC column, leaving that column with n_r partial results. In $S3$, a reduce-all operation that requires n_r broadcast operations across the corresponding column bus produces the final vector norm result in all the PEs of the owner column. Thus, performing a vector norm in the LAC is straightforward. The real challenge is the extra complexity to find the maximum value and to scale the vector by it, which is introduced for avoiding overflow and underflow. This will be discussed in the next section.

6.1.4 Hardware Extensions

In the previous sections, we discussed how the LAC architecture requires extensions to support matrix factorizations. We categorize these extensions into two groups. The first group extends each MAC unit to remove complexity from operations like vector-norm and LU with pivoting. The second group supports special functions like reciprocal and inverse square-root that are used in TRSM, Cholesky, and LU factorization. In the following, we summarize possible extensions. Further details of each extension are described in Section A.2.

Floating-Point Unit Extensions We add a comparator to each floating-point unit of each PE to find the pivot while performing dot-product computations. We also add support for an extra exponent bit in the MAC unit to avoid overflow or underflow. In Section A.2 we discuss how this extra bit eliminates the required normalization part in the vector-norm computation.

Special Functions Support We add a special function unit that can compute reciprocal, inverse square-root, square-root, and division functions for the LAC. An alternative option is to extend the PEs on the LAC so they can

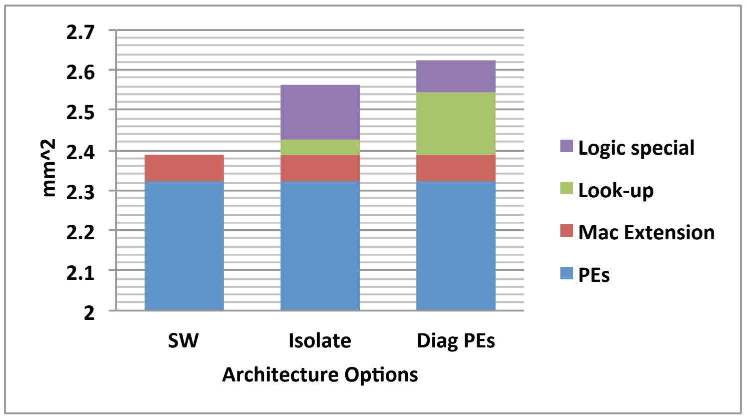


Figure 6.5: LAC area break-down with different divide/square-root extensions.

directly support such operations themselves. We use multiplicative methods, which use iterative MAC operations and table look-ups for approximation. Such multiplicative methods allow us to use the existing MAC units to perform the computations.

6.1.5 Results

We study the performance and efficiency behavior of our extensions for these algorithms and different inner kernel problem sizes. A very important point is that even larger problems sizes are usually blocked into smaller subproblems that cast most of the operations into a combination of highly efficient level-3 BLAS operations and the complex inner kernels that we discuss here.

For our study, we first assumed three different LAC architectures with three options for divide/square-root extensions: first, a software-like implementation that uses a micro-programmed state machine to perform Goldschmidt's [123] operation on the MAC unit in the PE; second, an isolated

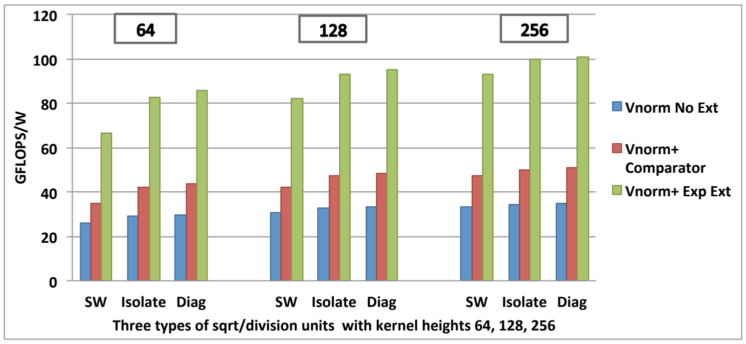


Figure 6.6: The effect of hardware extensions and problem sizes on the power efficiency of vector norm inner kernel.

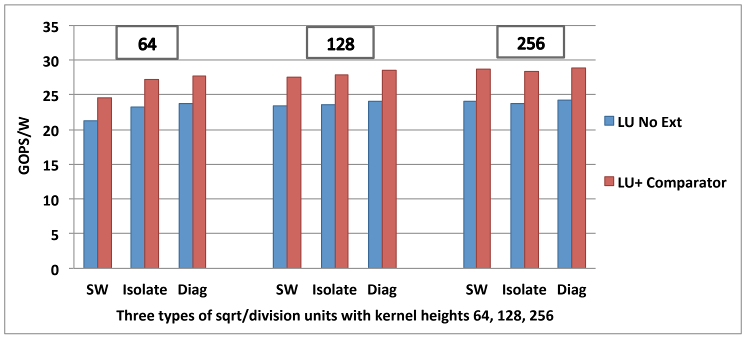


Figure 6.7: The effect of hardware extensions and problem sizes on the power efficiency of LU factorization with partial pivoting inner kernel.

divide/square-root unit; and third, a hardware extension to the PEs that adds extra logic around the available MAC units in the diagonal PEs. Area overhead for each of these options is shown in Figure 6.5. We can observe that in case of a 4×4 LAC, the overhead for these extensions is around 10% if an isolated unit is added to the LAC. If the extensions are added to all the diagonal PEs, more area is used.

We further assumed two types of extensions for the MAC units in the LAC, which include the maximum finder comparator and the extra exponent bit. Resulting power efficiencies for vector norm and LU operations are presented in Figures 6.6, and 6.7, respectively. We can observe that for the LU factorization, there is a 20% speed and 15% energy improvement with the comparator added to the MAC units. The exponent extension halves the total cycles of the vector-norm, and the divide/square-root unit saves up to 30% cycles compared to the baseline. Energy savings reach up to 60% with the exponent bit extension.

In summary, with limited logic extensions for these algorithms save cycles and consume less power. However, the bigger impact is the fact that the LAC does not need to waste cycles and energy to send such inner kernels to a host processor for computation.

6.2 Fast Fourier Transform

To investigate fundamental tradeoffs between flexibility and efficiency in our architecture, we further studied applications that go beyond the traditional linear algebra domain. Specifically, we explored the mapping of FFTs, which are an important signal processing kernel [110]. While GEMM is a straightforward kernel with simple, predictable data access patterns, the FFT provides more challenges to obtaining high performance. First, the increased ratio of data movement per computation (even with perfect caches) will cause the algorithm to be memory bandwidth limited on most current computer systems. Second, memory access patterns include strides of $2, 4, 8, ...N/2$, which interfere pathologically with the cache indexing and the cache and memory banking for standard processor designs. Third, the butterfly operation contains more additions than multiplications, so the “balanced” FPUs on most current architectures will be under-utilized.

In this section, we briefly analyze the similarities between algorithms and show how one might transform an optimized GEMM core into an FFT core. We consider whether a combined core that can perform either operation efficiently is practical, and we analyze the loss in efficiency required to achieve this flexibility. Complete details of mapping an FFT on the LAC along with the required modifications that need to be made to the existing core architecture are discussed in Appendix B.

6.2.1 FFT Algorithm and Mapping

At the lowest level, FFT algorithms are based on combining a small number of complex input operands via sum, difference, and complex multiplications to produce an equal number of complex output operands. These are referred to as “butterfly” operations because of the shape of the dataflow diagram (e.g., as shown later in Figure B.2). In this section, we briefly give the mathematical description of Radix-2 and Radix-4 FFT butterfly operations as optimized for execution on Fused Multiply-Add (FMA) units. Then, we discuss the data communication patterns that are needed when applying these operations to compute FFTs of longer sequences.

The Radix-2 Butterfly operation can be written as the following matrix operation, where w_L^j are constant values (usually referred to as “twiddle factors") that we store in memory:

$$\begin{pmatrix} x(j) \\ x(j+L/2) \end{pmatrix} := \begin{pmatrix} 1 & \omega_L^j \\ 1 & -\omega_L^j \end{pmatrix} \begin{pmatrix} x(j) \\ x(j+L/2) \end{pmatrix}.$$

This operation contains a complex multiplication operation and two complex additions, corresponding to 10 real floating-point operations. Using a floating-point MAC unit, this operation takes six Multiply-ADD operations that yields into 83% utilization.

A modified, FMA-optimized butterfly is introduced in [69], where the multiplier matrix in the normal butterfly is factored and replaced by:

$$\begin{pmatrix} 1 & \omega_L^j \\ 1 & -\omega_L^j \end{pmatrix} = \begin{pmatrix} 2 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & -\omega_L^j \end{pmatrix}.$$

This algorithm requires 12 floating point operations represented in six multiply-adds. Although the total number of floating-point operations is increased, they all utilize a fused multiply-add unit and the total number of FMAs remains six.

A Radix-4 FFT butterfly is typically represented as the following matrix operation:

$$\begin{pmatrix} x(j) \\ x(j+L/4) \\ x(j+L/2) \\ x(j+3L/4) \end{pmatrix} \times = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{pmatrix} \mathbf{diag}(1, \omega_L^j, \omega_L^{2j}, \omega_L^{3j}).$$

This contains three complex multiplications and eight complex additions that sum up to 34 real floating-point operations. The number of complex additions is much larger than the number of multiplications. Hence, there is a clear computation imbalance between multiplications and additions. Note also that three complex twiddle factors ω_L^j, $\omega_L^2 j$, and $\omega_L^3 j$ all have to be brought into the butterfly unit.

Alternately, the Radix-4 matrix above can be permuted and factored to give the following representation ($\omega = \omega_L^j$):

$$\begin{pmatrix} x(j) \\ x(j+L/4) \\ x(j+L/2) \\ x(j+3L/4) \end{pmatrix} \times = \left(\begin{array}{cc|cc} 1 & 0 & \omega & 1 \\ 0 & 1 & 0 & -i\omega \\ \hline 1 & 0 & -\omega & 0 \\ 0 & 1 & 0 & -i\omega \end{array}\right) \left(\begin{array}{cc|cc} 1 & \omega^2 & 0 & 0 \\ 1 & -\omega^2 & 0 & 0 \\ \hline 0 & 0 & 1 & \omega^2 \\ 0 & 0 & 1 & -\omega^2 \end{array}\right).$$

This can be further divided recursively using the same factorization as in the radix-2 FMA-adapted version. The result generates 24 FMA operations (as depicted later in Figure B.1). The FMAC utilization for the Radix-4 butterfly is 34/48=70.83%, but this corresponds to 40/48=83.33% if using the nominal $5NLog_2^N$ operation count from the Radix-2 algorithm that is traditionally used in computing the FLOP rate. Further details about this algorithm will be presented in Section B.2. The number of loads also drops because only two of the three twiddle factors (ω_L^j and $\omega_L^2 j$) are required to perform the computations.

The two implementations of an L-point Radix-4 FFT are shown below. The pseudo-code for the standard implementation is shown on the left and the pseudo-code for the FMA optimized version is shown on the right:

```
for $j = 0 : L/4 - 1$
    $a := x(j);$
    $b := \omega_L^j x(j + L/4)$
    $c := \omega_L^{2j} x(j + L/2)$
    $d := \omega_L^{3j} x(j + 3L/4)$
    $\tau_0 := a + c$
    $\tau_1 := a - c$
    $\tau_2 := b + d$
    $\tau_3 := b - d$
    $x(j) := \tau_0 + \tau_2;$
    $x(j + L/4) := \tau_1 - i\tau_3;$
    $x(j + L/2) := \tau_0 - \tau_2;$
    $x(j + 3L/4) := \tau_1 + i\tau_3;$
end for
```

```
for $j = 0 : L/4 - 1$
    $a := x(j);$
    $b := x(j + L/4)$
    $c := x(j + L/2)$
    $d := x(j + 3L/4)$
    $b := a - \omega_L^{2j} b$
    $a := 2a - b$
    $d := c - \omega_L^{2j} d$
    $c := 2c - d$
    $x(j + L/2) := c = a - \omega_L^j c$
    $x(j) := 2a - c$
    $x(j + L/4) := d := b - i\omega_l^j d$
    $x(j + 3L/4) := 2b - d$
end for
```

The broadcast bus topology in the LAC allows a PE to communicate with other PEs in the same row and with other PEs in the same column simultaneously. This can be effectively exploited for mapping of FFT algorithms. To maximize locality, we consider only designs in which each butterfly operation is computed by a single PE, with communication taking place between the butterfly computational steps. We note that if the LAC dimensions are selected as powers of two, the communication across PEs between both Radix-2 or Radix-4 butterfly operations will be limited to the neighbors on the same row or column, which naturally maps to our broadcast bus architecture.

6.2.2 Hardware Extensions

Details of core and PE configuration tradeoffs and design choices are presented in Appendix B. Here we briefly mention the architectural modification for the core and the PEs.

Core Extensions The off-core bandwidth needs to be double that of the original LAC design. Furthermore, the PEs must be able to overlap the prefetching of input data and the post-storing of output data from/to off-core memory concurrently with the computations. Doubling the memory bandwidth can be implemented by expanding the memory interface so that both row and column buses can transfer data to/from PEs.

The PE Extensisons The PE micro-architecture must perform the three tasks of Radix-4 butterfly computation, FFT communication, and off-core communication concurrently. Some extra logic and storage is needed to facilitate data movements and locality. These options are described with the help of Figure 6.8. An 8-byte register file is needed to store the four complex input, temporary, and output values of the FMA-optimized Radix-4 butterfly. The twiddle factors take an extra four registers. The larger PE SRAM is divided into two halves and an extra bus is added to provide enough data bandwidth for both Radix-4 computations and off-core communications as in shown in Figure 6.8 (right).

6.2.3 Results

In this section, we present area, power and performance estimates for the LAC with the modifications introduced in previous sections.

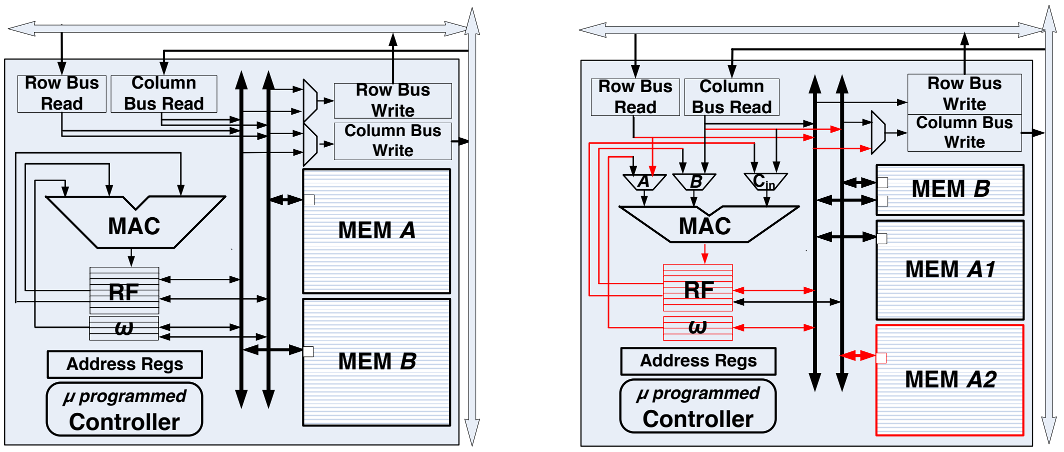


Figure 6.8: New PE configurations for full-overlap FMA-optimized Radix-4 FFT: (left) FFT-optimized PE with two 8-byte, single-ported SRAMs, and (right) Hybrid PE with two 8-byte, single-ported SRAMs to contain matrix A.

Figure 6.9 demonstrates the normalized efficiency metrics of the three different PE designs based on the original LAC design. The first option is the basic LAC, the second option is an FFT optimized core based on 2D arrangement of PEs with MAC units, and the third option is a hybrid core that can perform both GEMM and FFT operations. We can observe that the hybrid design has lower efficiency when considering maximum power and area. Note that in all cases, the efficiency numbers are already scaled by achievable utilization.

Finally, Table 6.2 provides comparisons of estimated performance, area, and power consumption between our proposed design and several alternative processors for which performance, area, and power estimates were available [30, 72, 150, 153]. In each case, we limit the comparison to double-precision 1D

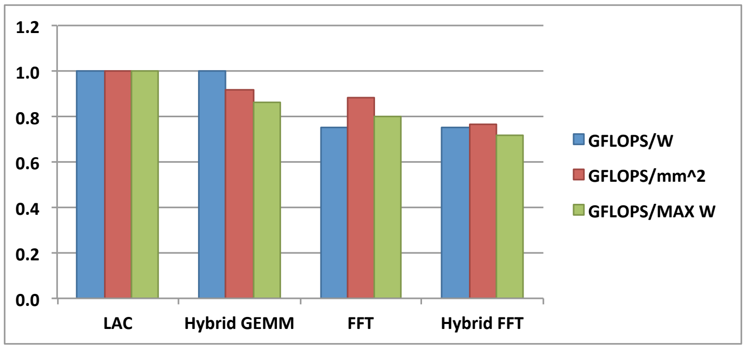


Figure 6.9: Efficiency of different designs normalized to the original LAC design at 1 GHz.

Platform Running FFT	Problem size fits in	KBytes	FFT GFLOPS	Power (Watt)	GFLOPS/ Watt	GFLOPS/ mm^2	Utilization
Hybrid Core	Core SRAM	288	26.7	0.66	40.50	12.12	83%
Hybrid Core+SRAM	Off-core	2336	26.7	1.02	26.30	1.71	83%
Xeon E3-1270 core	L2 $	288	12.0	28	0.43	0.33	44%
ARM Cortex A9	L1 $	32	0.6	0.28	2.13	0.45	60%
PowerXCell 8i SPE	SPE local	2048	12.0	64	0.19	0.12	11%
NVIDIA C2050	L1+L2 $	1728	110.0	150.00	0.73	0.21	21%

Table 6.2: Comparison between the proposed hybrid core and several alternatives for cache-contained double-precision FFTs scaled to 45nm.

FFT performance for problem sizes that fit into either the first and/or second levels of SRAM or cache. All area and power estimates are scaled to 45nm technology and include only the cores and SRAM. In each case, the proposed hybrid linear algebra and FFT engine provides at least an order of magnitude advantage in performance per watt and unit area.

6.3 Summary

In this chapter, we discussed the opportunities for extending the basic LAC architecture to support matrix factorizations and FFTs. We show how adding moderate complexity to the architecture of the LAC greatly alleviates complexities in the matrix factorization algorithms. We also explored mapping of FFTs on the core and showed how the 2D arrangement of PEs helps removing unnecessary communication between PEs. With less than 10% loss in efficiency, a hybrid core can perform FFT with orders of magnitude higher efficiency compared to other architectures. All of these opportunities are achieved by rethinking the process of design. The architecture is relaxed and therefore adapted to better support the algorithm requirements.

Chapter 7

Summary and Future Work

This chapter briefly reviews the dissertation Then, we discuss ongoing and future research opportunities.

7.1 Summary

This dissertation provides initial evidence regarding the benefits of custom hardware for dense linear algebra computations. We proposed the design and architecture of a specialized linear algebra processor (LAP), consisting of a number of linear algebra cores (LACs). Our Analysis show that a prototype LAP can achieve up to 600 GFLOPS for DGEMM while consuming less than 25 Watts in standard 45 nm technology, which is orders of magnitude more energy efficient than cutting-edge CPUs. We studied the multi-dimensional design trade-off space for our single- and multi-core design. Some of the axes of this space include the number of cores, the different sizes of cores, and the features of the different layers of the memory hierarchy, each layer with its own storage size and bandwidth to the next level. We developed an analytical framework to evaluate associated performance tradeoffs, which provides a powerful tool for designers to assess their design balance and its utilization.

We also studied the different factors in power consumption of such systems by coupling our analytical performance models with a power model. We modeled the power consumption for our design and its competitors and presented a power breakdown for different components of the architecture. The basic conclusion is that, as had been postulated, one to two orders of magnitude improvement in power and performance density can be achieved. Further, we showed how this architecture can support the full range of level-3 BLAS operations if small changes are introduced in the micro-architecture.

We examined how generally applicable the LAC/LAP architecture is by mapping more complex problems like matrix factorizations. Algorithms like LU and QR factorization introduce extra complexities to ensure numerical stability. We proposed modifications to the micro-architectural design of the LAC and its floating-point units to decrease the complexity of these algorithms. We also showed how an existing PE can be enhanced to support special functions for divide and square-root operations. This demonstrates the potential of this architecture for achieving high efficiency while being flexible enough to support a broad class of operations. The conclusion is that adding moderate complexity to the architecture greatly alleviates complexities in the algorithm.

To push the envelope, we studied the feasibility of mapping FFT, which is an algorithm from a different domain of applications. FFT has far more communications per operation, and more additions that multiplications. Current architectures achieve less than 50% utilization for this operation. Careful algorithm analysis for the target architecture, combined with judiciously chosen data-path modications, allowed us to produce a highly efficient accelerator for FFT operations with minor changes to the original linear algebra core. The proposed FFT engine provides at least an order of magnitude advantage in performance per watt and unit area compared to other processors.

In summary, this dissertation provides evidence that a flexible yet highly power efficient accelerator could be designed for the class of linear algebra operations. This was achieved by careful analyses of possible algorithms and existing architectures as a codesign process. We further showed how such algorithm/architecture codesign we could expand the flexibility of our architecture beyond linear algebra operations while maintaining its power efficiency.

7.2 Future Work

In the following, we will point to some of the future directions that could expand this multi-dimensional algorithm/architecture codesign space. We briefly cover each category of potential future research.

Micro-Architecture Level. PE and LAC designs may need further modifications in their logic and architecture to provide facilities for supporting more applications. An example is the design of floating-point units that can operate at variable precision or extending capabilities of the PEs to provide functionality for more special functions like Cordic. Furthermore, we have to design the logic for the core interface to on-chip memory and study its design tradeoffs.

System-Level Explorations. System-level integration is an important direction that opens up multiple research topics. The host interface for integration of one or more LAPs (or LACs) with one or more on-chip or off-chip host processors is part of system level development. We will try to clarify more design space details of the LAP when it is placed in heterogeneous systems. To achieve this, we plan to extend our cycle accurate simulator and integrate it into other multi-core simulators like MARSSx86 [102] or GEM5 [18] to study detailed design tradeoffs both at the core and chip level. These details include invocation, completion, memory addressing and task granularity (see Section 2.2.4). In a heterogeneous system, tasks have computational cost, and there is communication cost as data is moved between resources. A research direction can be to investigate how to best perform course-grain task scheduling and load balancing to exploit heterogeneous multicore architectures.

Software Techniques and Programming Interface. Future research directions more on the software side includes integration with existing libraries and using software techniques to optimize performance. We plan to collaborate with members of the FLAME research group in order to integrate our proposed LAP with libflame [138], a modern alternative to the widely used LAPACK [12] library. Advanced software techniques like loop fusion could be used in our codesign process to further optimize kernels and take advantage of data locality on target architectures.

Generalization. The goal of generalization is to map more algorithms on the LAC and analyze the associated cost in power and efficiency. In the end, a design space spectrum of flexibility and performance versus efficiency can be derived from this study. We plan to implement the collective communication routines for the hardware interconnect between PEs and add necessary hardware if needed. Furthermore, it becomes worthwhile to investigate widely used operations like Singular Value Decomposition (SVD) in the domain of linear algebra. We could try to go beyond FFT and codesign the LAC to map a wider class of signal processing applications as well. Finally, algorithms like Multi-Layer Perceptron (MLP), and Local Linear Model Tree (LOLIMOT) are based on computations on huge data sets that are processed as matrices [109]. We aim to study trade-offs and costs of adding such functionalities to the LAC.

Appendices

Appendix A

Core Level Extensions for Matrix Factorizations

Within the dense linear algebra domain, a typical computation can be blocked into sub-problems that expose highly parallelizable parts like GEneral Matrix-matrix Multiplication (GEMM). These can be mapped very efficiently to accelerators. However, many current solutions use heterogeneous computing for more complicated algorithms like Cholesky, QR, and LU factorization [7, 143]. Often, only the most parallelizable and simplest parts of these algorithms, which exhibit ample parallelism, are performed on the accelerator. Other more complex parts, which are added to the algorithm to overcome floating point limitations or which would require complex hardware to exploit fine grain parallelism, are offloaded to a general-purpose processor.

The problem with heterogeneous solutions is the overhead for communication back and forth with a general-purpose processor. In the case of current GPUs, data has to be copied to the device memory and then back to the host memory through slow off-chip buses. Even when GPUs are integrated on the chip, data has to be moved all the way to off-chip memory in order to perform transfers between (typically) incoherent CPU and GPU address spaces.

While the CPU could be used to perform other tasks efficiently, it is wasting cycles synchronizing with the accelerator and copying data. Often times the accelerator remains idle waiting for the data to be processed by the CPU, also wasting cycles. This is particularly noticeable for computation with small matrices.

In this appendix, we propose a new solutions that try to avoid all inefficiencies caused by limitations in current architectures and thereby overcome the complexities in matrix factorization algorithms. The problem is that architecture designers typically only have a high-level understanding of algorithms, while algorithm designers try to optimize for already existing architectures. Our solution is to revisit the whole system design by relaxing the architecture design space. By this we mean allowing architectural changes to the design in order to reduce complexity directly in the algorithm whenever possible. Thus, the solution is to exploit algorithm/architecture co-design. We add minimal, necessary but sufficient logic to the LAC design to avoid the need for running complex computations on a general-purpose core.

A.1 Related Work

Implementation of matrix factorizations on both conventional high performance platforms and accelerators has been widely studied. Many existing solutions perform more complex kernels on a more general-purpose (host) processor while the high-performance engine only computes paralellizable blocks of the problem [7, 143].

The typical solution for LU factorization on GPUs is presented in [143]. The details of multi-core, multi-GPU QR factorization scheduling are discussed in [7]. A solution for QR factorization that can be entirely run on the GPU is presented in [71]. For LU factorization on GPUs, a technique to reduce matrix decomposition and row operations to a series of rasterization problems is used [44]. There, pointer swapping is used instead of data swapping for pivoting operations.

On FPGAs, [151] discusses LU factorization without pivoting. However, when pivoting is needed, the algorithm mapping becomes more challenging and less efficient due to complexities of the pivoting process and wasted cycles. LAPACKrc [49] is a FPGA library with functionality that includes Cholesky, LU and QR factorizations. The architecture has similarities to the LAP. However, due to limitations of FPGAs, it does not have enough local memory. Similar concepts as in this document for FPGA implementation and design of a unified, area-efficient unit that can perform the necessary computations (division, square root and inverse square root operations that will be discussed later) for calculating Householder QR factorization is presented in [13]. Finally, a tiled matrix decomposition based on blocking principles is presented in [130].

A.2 Hardware Extensions

In this section, we discuss how to overcome the challenges that are discussed in Section 6.1 with regards to the mapping of factorization algorithms on the LAC. These extensions allow an architecture to perform more complex operations more efficiently. We will introduce architecture extensions that provide such improvements specifically for factorizations. However, such extensions also introduce a base overhead in all operations, since they add extra logic and cause more power and area consumption. Corresponding trade-offs will be analyzed in the results section.

Here, we focus on small problems that fit in the LAC memory. Bigger problem sizes can be blocked into smaller problems that are mainly composed of Level-3 BLAS operations (discussed in [135]) and algorithms for smaller problems discussed here. We briefly review the relevant algorithms and their micro-architecture mapping in Section 6.1. The purpose is to expose specialized operations, utilized by these algorithms, that can be supported in hardware. We start by analyzing opportunities for extensions targeting Cholesky and LU factorization, followed by solutions to complexities in vector norm operations.

A.2.1 Cholesky Factorization

We observe that the key complexity when performing Cholesky factorization is the inverse square-root operation. If we add this ability to the core's diagonal PEs, the LAC can perform the inner kernel of the Cholesky factorization natively. The last state of the $n_r \times n_r$ Cholesky factorization will save even more cycles if a square-root function is available. The $n_r \times n_r$ Cholesky factorization is purely sequential with minimal parallelism in rank-1 updates.

However, it is a very small part of a bigger, blocked Cholesky factorization. Again, the goal here is to avoid sending data back and forth to a general purpose processor or performing this operation in emulation on the existing MAC units, which would keep the rest of the core largely idle.

A.2.2 LU Factorization with Partial Pivoting

For LU factorization with partial pivoting, PEs in the LAC must be able to compare floating-point numbers to find the pivot (S1 in Section 6.1.2). In the blocked LU factorization, we have used the left-looking algorithm, which is the most efficient variant with regards to data locality [17]. In the left-looking LU factorization, the PEs themselves are computing the temporary values that they will compare in the next iteration of the algorithm. Knowing this fact, the compare operation and its latency could be done implicitly without any extra latency and delay penalty.

The next operation that is needed for LU factorization is the reciprocal ($1/x$). The reciprocal of the pivot needs to be computed for scaling the elements by the pivot (S2 in Section 6.1.2). This way, we avoid multiple division operations and simply multiply all the values by the reciprocal of the pivot and scale them.

A.2.3 QR Factorization and Vector Norm

In Section 6.1.3, we showed how the vector norm operation is performed in conventional computers to avoid overflow and underflow. The extra operations that are needed to perform vector norm in a conventional fashion are the following: a floating-point comparator to find the maximum value in the vector just as in LU factorization, a reciprocal function to scale the vector by the maximum value, again just as in LU factorization, and a square-root unit to compute the length of the scaled vector just as what is needed to optimize the last iteration of a $n_r \times n_r$ Cholesky factorization. However, we can observe that all these extra operations are only necessary due to limitations in hardware representations of real numbers.

Consider a floating number f that, according to the IEEE floating-point standard, is represented as $1.m_1 \times 2^{e_1}$, where $1 \leq 1.m_1 < 2$. Lets investigate the case of an overflow for $p = f^2$, and as a result $p = (1.m_2) \times 2^{e_2} = (1.m_1)^2 \times 2^{2e_1}$, where $1 \leq (1.m_1)^2 < 4$. If $(1.m_1)^2 \leq 2$, then $e_2 = 2e_1$. But, if $2 \leq (1.m_1)^2$, then $2 \leq (1.m_1)^2 = 2 \times 1.m_2 \leq 2$ and therefore $e_2 = 2e_1 + 1$. In both cases, a single extra exponent bit suffices for avoiding overflow and underflow in computations of the square of a floating-point number.

Still, there might be the possibility of overflow/underflow due to accumulation of big/small numbers that could be avoided by adding a second exponent bit. However, the square-root of such inner product is still out of the bounds of a standard floating-point number. Therefore, only a single additional bit suffices. Hence, what is needed is a floating-point unit that has the ability to add one exponent bit for computing the vector norm to avoid overflows and corresponding algorithm complexities.

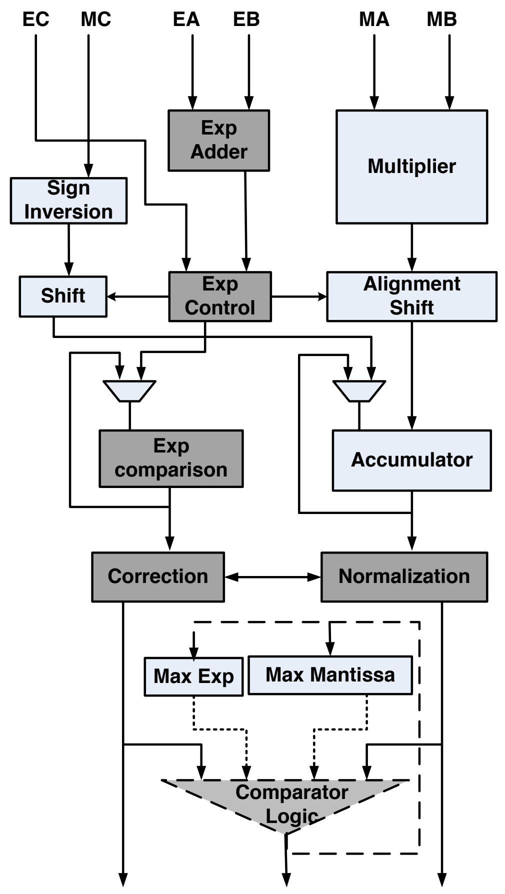


Figure A.1: Extended reconfigurable single-cycle accumulation MAC unit [63] with addition of a comparator and extended exponent bit-width, where shaded blocks show which logic should change for exponent bit extension.

A.3 Architecture

In this section, we describe the proposed architecture for our floating-point MAC unit and the extensions made to it for matrix factorization applications. We start from a single-cycle accumulating MAC unit and explain the modifications for LU and vector norm operations. Then, we describe the extensions for reciprocal, inverse square-root, and square-root operations.

A.3.1 Floating-Point MAC Unit

A floating-point MAC unit with single-cycle accumulation is presented in [142]. Using the same design principles, [63] presents a reconfigurable floating-point MAC that is also able to perform multiplication, addition and multiply-add operations. This design does not support operations on denormalized numbers [142]. We describe our modifications to the same design as shown in Figure A.1.

The first extension is for LU factorization with partial pivoting, where the LAC has to find the pivot by comparing all the elements in a single column. We noted that PEs in the same column have produced temporary results by performing rank-1 updates. To find the pivot, we add a comparator after the normalization stage in the floating-point unit of each PE. There is also a register that keeps the maximum value produced by the corresponding PE. If the new normalized result is greater than the maximum, it replaces the maximum and its index is saved by the external controller. An extra comparator is a simple logic in terms of area/power overhead [129]. It is also not a part of the critical path of the MAC unit and does not add any delay to the original design. With this extension, finding the pivot is simplified to a search among only n_r elements that are the maximum values produced by each PE in the same column.

The second extension is for vector norm operations in the Householder QR factorization. Previously, we have shown how adding an extra exponent bit can overcome overflow/underflow problems in computing the vector norm without the need for performing extra operations to find the biggest value and scale the vector by it. In Figure A.1, the shaded blocks show where the architecture has to change. These changes are minimal and their cost is negligible. Specifically, with the architecture in [142], the same shifting logic for a base-32 shifter can be used. The only difference here is that the logic decides between four exponent input bits instead of three.

A.3.2 Reciprocal and (Inverse) Square-root Units

In Cholesky factorization, we observed that the LAC needs a way to compute the inverse square-root of the diagonal elements and scale the corresponding column with the result. Adding a square-root unit can also save more cycles in the last iteration of a $n_r \times n_r$ Cholesky factorization. Furthermore, LU factorization needs a reciprocal operation to scale the elements by the pivot. As discussed in [108], a reciprocal unit is also mandatory for TRiangular Solve with Multiple right-hand side (TRSM) operations to support the complete Level-3 BLAS. In this section, we will give details and design options for such a unit.

Division, reciprocal, square-root, and inverse square-root functions are used in many applications in the domain of signal processing, computer graphics, and scientific computing [99, 127]. Several floating-point divide and square-root units have been introduced and studied in the literature [35, 98, 113]. There are mainly two categories of implementations in modern architectures: multiplicative (iterative) and subtractive methods. An extensive presentation of these methods and their hardware implementations are presented in [127].

Two main multiplicative methods for calculating divide and square-root functions are Newton-Raphson and Goldschmidt's. These algorithms work iteratively to refine an initial approximation. They utilize a look-up table for initial approximation and the number of result digits doubles after each iteration (converging at a quadratic rate). In each iteration, a series of multiplication, subtraction, and shifts are performed, which means a multiply-add unit could be utilized for these operations. Hence, they can be implemented as an enhancement on top of such existing units. Goldschmidt's method, which is based on a Taylor series with two independent multiplication operations, is more suitable for pipelined floating-point units than the Newton-Raphson method.

Subtractive (digit recurrence), which are also known as SRT methods, directly calculate (multiple) digits of the desired result. They have high latency and generally are implemented as a dedicated, complex component. However, there are redundancies between the division and square-root units that allow a single unit to perform both operations. For the higher radix implementations with lower latencies, these designs become complex and area consuming.

In [127, 128], it is concluded that a separate SRT-based subtractive divide and square-root unit is more efficient for a Givens rotation application. This is because multiplicative methods occupy the Multiply-Add (MAD) unit and prevent it to do anything else, while subtractive methods work in parallel with an existing MAC unit, resulting into a faster design.

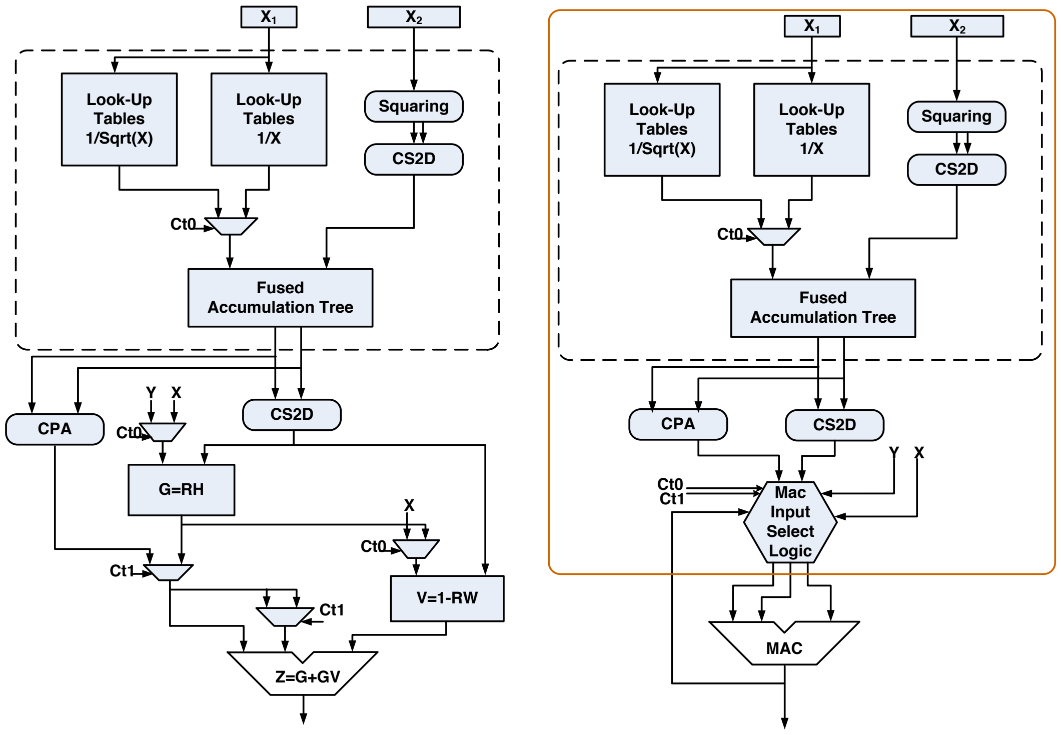


Figure A.2: Floating-point unit extensions: (left) original divide, reciprocal, square-root and inverse square-root design with the Minimax logic [113] used for the isolate unit; (right) a single MAC unit design to support special functions. The overheads on top of an existing MAC unit are encapsulated in the big rounded rectangle. PEs in the LAC with that overhead can perform special functions.

Operation	$G = RH$	$V = 1 - RW$	$Z = G + GV$	Ct0 Ct1
Division	$G = RY$	$V = 1 - RX$	$Z = G + GV$	00
Reciprocal	–	$V = 1 - RX$	$Z = R + RV$	01
Squar-root	$G = RX$	$V = 1 - GS$	$Z = G + GV/2$	10
Inv Sqrt	$G = RX$	$V = 1 - GS$	$Z = R + RV/2$	11

Table A.1: Operations of the divide and square-root unit with control signals [113].

Given the nature of linear algebra operations and the mapping of algorithms on the LAC, a multiplicative method is chosen. The reason lies within the fact that there are many MAC units in the core, and exploiting one of them for divide or square-root will not harm performance. In our class of applications, a divide and square-root operation is often performed when other PEs are waiting in idle mode for the its result. As the iterations of Cholesky and LU factorization go forward, only a part of the LAC is utilized, and the top left parts are idle. Therefore, a diagonal PE is the best candidate for such extensions on top of its MAC unit.

The design we are considering for this work is the architecture presented in [113]. It uses a 29-30 bit approximation with a second-degree minimax polynomial approximation that is known as the optimal approximation of a function [114]. This approximation is performed by using table look-ups. Then, a single iteration of a modified Goldschmidt's method is applied. This architecture, which is shown in Figure A.2(left), guarantees the computation of exactly rounded IEEE double-precision results [113]. It can perform all four operations: divide Y/X, reciprocal 1/X, square-root $\sqrt{X}$, and inverse square-root $1/\sqrt{X}$. While utilizing the same architecture for all operations, the divison/reciprocal operations take less time to be computed, since computing G and V can be done in parallel. In case of square-root/inverse square-root, all operations are sequential and, as a result, the latency is higher. Figure A.3.2 shows the type of operations and control signals that are performed for all four functions.

The design in Figure A.2(left) could be reduced to use a single reconfigurable MAC unit, which performs all the computations itself. This strategy reduces the design area and overhead. This reduction does not increase the latency, but reduces the throughput. However, as indicated before, for our class of linear algebra operations, there is no need for a high-throughput division/square root unit. Therefore, the design with a single reconfigurable MAC unit as shown in Figure A.2(right) is preferred. The extra overhead on top of an unmodified MAC unit includes the approximation logic and its look-up tables. A simple control logic performs the signal selection for the MAC inputs.

In summary, the changes we apply to the PEs in the LAC are as follows: all PEs in the LAC design will get the extra-exponent bit and the comparator logic for vector norm and LU with partial pivoting operations, respectively. There are three options for the divide and square-root unit implementation in the LAC: first, a separate unit can be used to be shared by all of PEs, or the top-left PE can be modified to hold the extra logic on top of its MAC unit. A third option is to add the divide and square-root logic to all diagonal PEs. We will evaluate these options and their trade-offs for our applications in the next section.

A.4 Experimental Results and Implementations

In this section, we present area, power and performance estimates for the LAC with the modifications introduced in previous sections. We will compare the performance to a pure software-like (micro-coded) implementation of additional complex operations using existing components and micro-programmed state machines. We chose three different problem sizes and we perform an area, power, and efficiency study to evaluate the benefits of these architectural extensions.

A.4.1 Area and Power Estimation

We use the power and area data from [43] and combine it with complexity and delay reports from [113] for floating-point units. We assumed two types of extensions for the MAC units in the LAC, which include the maximum finder comparator and the extra exponent bit (Figure A.1). We also assumed three different LAC architectures with three options for divide/square-root extensions: first, a software-like implementation that uses a micro-programmed state machine to perform Goldschmidt's operation on the MAC unit in the PE; second, an isolated divide/square-root unit that performs the operation with the architecture in Figure A.2(left); and third, an extension to the PEs that adds extra logic and uses the available MAC units in the diagonal PEs (Figure A.2(right)).

The comparator is not on the critical path of the MAC pipeline and the extensions for the extra exponent bit are negligible. Therefore, we assume that there is no extra latency added to the existing MAC units with these extensions. The divide and square-root unit's timing, area, and power estimations are calculated using the results in [113]. For a software solution with multiple Godlschmidt iterations, we assume no extra power or area overhead for the micro-programmed state machine.

The area overhead for diagonal PEs includes the selection logic and the

Problem Size	Total Cycles			Dynamic Energy		
	SW	Isolated	Diagonal	SW	Isolated	Diagonal
Cholesky						
4	496	192	176	4 nJ	1 nJ	1 nJ
LU Factorization						
64	524	340	340	62 nJ	60 nJ	60 nJ
128	700	644	644	121 nJ	119 nJ	119 nJ
256	1252	1252	1252	239 nJ	236 nJ	236 nJ
LU Factorization With Comparator						
64	500	316	316	53 nJ	51 nJ	51 nJ
128	612	556	556	103 nJ	101 nJ	101 nJ
256	1036	1036	1036	202 nJ	200 nJ	200 nJ
Vector norm						
64	282	158	150	32 nJ	29 nJ	29 nJ
128	338	214	206	59 nJ	56 nJ	56 nJ
256	418	294	286	114 nJ	111 nJ	111 nJ
Vector norm With Comparator						
64	276	152	144	23 nJ	20 nJ	20 nJ
128	308	184	176	41 nJ	38 nJ	38 nJ
256	372	248	240	78 nJ	75 nJ	75 nJ
Vector norm With Exponent bit extension						
64	154	80	76	12 nJ	10 nJ	10 nJ
128	170	96	92	21 nJ	19 nJ	19 nJ
256	202	128	124	39 nJ	37 nJ	38 nJ

Table A.2: Total cycle counts and dynamic energy consumption for different architecture options (columns for divide/square-root options, and row sets for MAC unit extension options), algorithms and problem sizes.

minimax function computation. In case of a 4×4 LAC, we observe that the overhead for these extensions is around 10% if an isolated unit is added to the LAC (see Figure 6.5 in Section 6.1). If the extensions are added to all the diagonal PEs, more area is used. However, with an isolated unit more multipliers and multiply-add unit logic is required. The benefit of using the diagonal PEs is in avoiding the extra control logic and in less bus overhead for sending and receiving data.

A.4.2 Performance and Efficiency Analysis

In this part, we analyze the unblocked inner kernels of the three factorization algorithms. We study the performance and efficiency behavior of our extensions for these algorithms and different inner kernel problem sizes. A very important point is that even larger problems sizes are usually blocked into smaller subproblems that cast most of the operations into a combination of highly efficient level-3 BLAS operations and the complex inner kernels that we discuss here. Many accelerators only support level-3 BLAS and perform more complex kernels on the host processor. The overhead of sending the data associated with these computations back and forth is significant and affects the performance by wasting cycles. However, such issues are out of the scope of this document. What we want to show here is how effective our proposed extensions are in achieving high performance for the inner kernels compared to the baseline architecture with a micro-coded software solution.

Cholesky factorization can be blocked in a 2D fashion by breaking the problem down to a few level-3 BLAS operations and a Cholesky inner kernel. For our experiment, we evaluate a 4×4 unblocked Cholesky. We study the effects of different divide/square-root schemes on the performance of this inner kernel. The kernel performance and utilization is low because of the dependencies and the latency of the inverse square-root operation. We observe (Table A.2) that the number of cycles drops by a third by switching from a software solution to hardware extensions on the LAC.

LU factorization with partial pivoting is not a 2D-scalable algorithm.

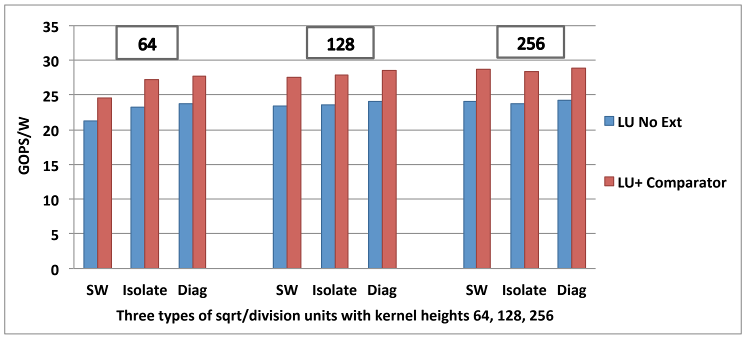


Figure A.3: The effect of hardware extensions and problem sizes on the power efficiency of LU factorization with partial pivoting inner kernel.

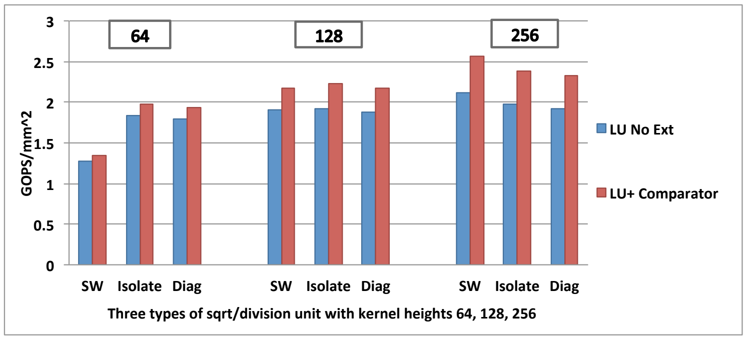


Figure A.4: The effect of hardware extensions and problem sizes on the area efficiency of LU factorization with partial pivoting inner kernel.

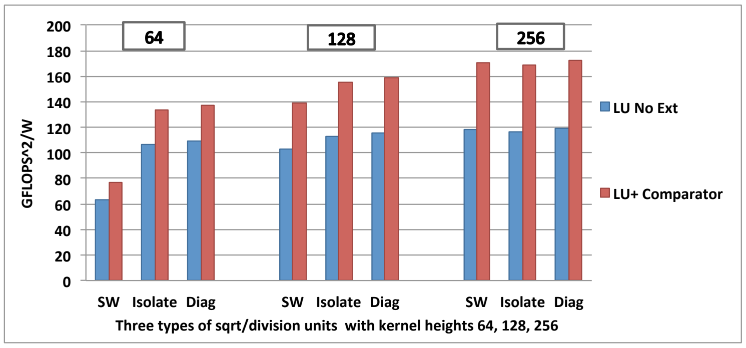


Figure A.5: The effect of hardware extensions and problem sizes on the inverse E-D metric of LU factorization with partial pivoting inner kernel.

The pivoting operation and scaling needs to be done for all rows of a given problem size. Hence, for a problem size of $k \times k$, the inner kernel that should be implemented on the LAC is a LU factorization of a $k \times n_r$ block of the original problem. For our studies, we use problems with different $k = 64, 128, 256$, which are typical problem sizes that fit on the LAC. We compare the performance of a LAC with different divide/square-root unit extensions in different columns and with/without the built-in comparator to find the pivot. As we have shown in Section 6.1, the reciprocal operation and pivoting (switching the rows) can be performed concurrently in the LAC owing to the column broadcast buses. The pivoting delay is the dominating term. Hence, bigger problem sizes are not sensitive to the latency of the reciprocal unit architecture. However, there is a 20% speed and 15% energy improvement with the comparator added to the MAC units.

Vector norm as part of a Householder transformation only utilizes a single column of PEs for the inner product and reduce. To measure the maximum achievable efficiency, we assume that there are four different vector norms completing concurrently one in each column. Note that the baseline is the original normalizing vector norm. We have three options for divide/square-root operations, and three options for MAC unit extensions. The first option is a micro-coded software solution, the second option is utilizing the comparator in the MAC unit without an exponent extension, and the last is a MAC unit with an extra exponent bit. The problem sizes are again $k = 64, 128, 256$ different vector lengths. As shown in Table A.2, we can observe that the exponent extension halves the total cycles, and the divide/square-root unit saves up to 30% cycles compared to the baseline. Energy savings reach up to 60% with the exponent bit extension. By contrast, different divide/square-root units do not differ in terms of dynamic energy consumption.

We assume a clock frequency of 1GHz for the LAC. Utilization and efficiency can be calculated from the number of total cycles the hardware needs to perform an operation and the number of operations in each factorization. Power efficiency for vector norm and LU are presented in Figures A.6, A.3 respectively. Figures A.7, A.4 also represent the area efficiency respectively. Another metric that we use is the inverse energy-delay. It shows how extensions reduce both latency and energy consumption. Note that for LU factorization, the pivoting operation is also taken into account. Therefore, we used GOPS instead of GFLOPS as performance metric. For LU factorization problems with $k = 64, 128, 256$, we estimated the corresponding total number of operations to be 1560, 3096 and 6168, respectively. For the vector norm, we use the original algorithm as the baseline, which requires 257, 769 or 1025 operations per corresponding vector norm of size $k = 64, 128, 256$. Since our implementation will result in an effective reduction in the number of actually required computations, the extensions have higher GOPS/W than what is reported as peak GFLOPS/W for the LAC in [105].

Results for LU factorization confirm that there is no improvement in efficiency with different reciprocal architectures when solving big problem sizes. Given this fact, isolated unit seems to be a better option for LU factorization.

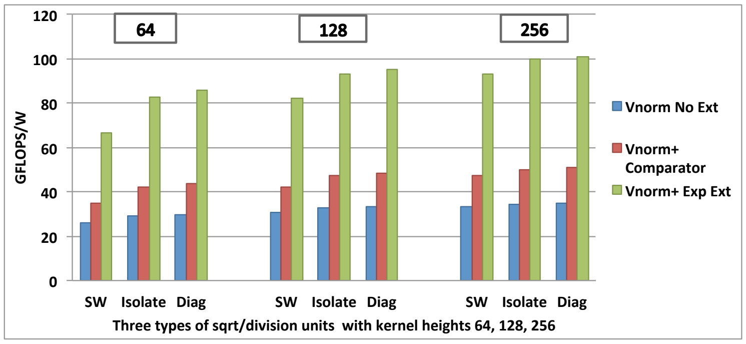


Figure A.6: The effect of hardware extensions and problem sizes on the power efficiency of vector norm inner kernel.

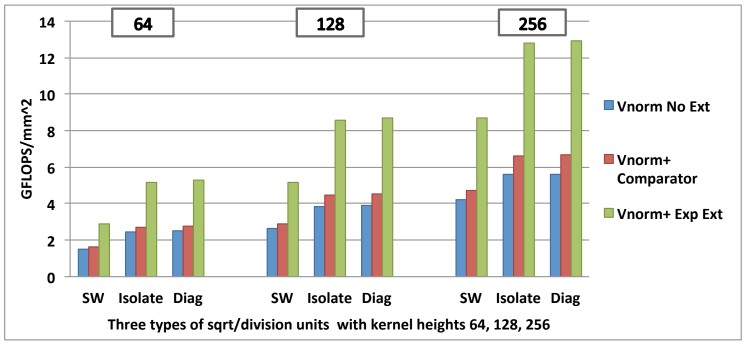


Figure A.7: The effect of hardware extensions and problem sizes on the area efficiency of vector norm inner kernel.

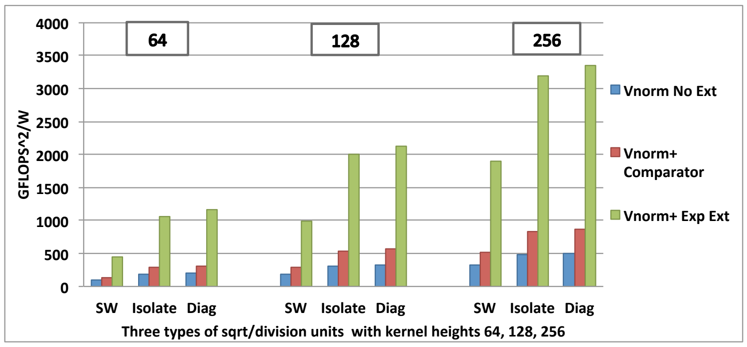


Figure A.8: The effect of hardware extensions and problem sizes on the inverse E-D metric of vector norm inner kernel.

By contrast, vector norm benefits from all types of extension. However, the exponent bit is what brings significant improvements in efficiency.

Since there are not many options for Cholesky, we only summarize the numbers here in the text. The number of operations in a 4×4 Cholesky kernel is 30. For different divide/square unit architectures (software, isolated, and on diagonal PEs), the achieved efficiencies are as follows: 1.95, 4.67 and 5.75 GFLOPS/W; 0.52, 4.95, and 5.15 GFLOPS^2/W; and 0.03, 0.06, 0.07 GFLOPS/mm^2. The reason for the very poor efficiency (less than 5 GFLOPS/W) is the small size of the kernel and limited available parallelism. Still, adding the special function unit improves efficiency around ten times, while reducing dynamic energy consumption by 75%.

A.5 Summary

In this appendix, we propose two modifications to the MAC unit designs to decrease the complexity of factorization algorithms. We also show how existing processing elements can be enhanced to perform special functions such as divide and square-root operations. To demonstrate the effectiveness of our proposed extensions, we applied them to the mapping of Cholesky, LU and QR factorizations on such an improved architecture. Results show that our extensions significantly increase efficiency and performance.

Future work includes comparison and mapping of big, tiled matrix factorization problems onto the LAC, including its integration into a heterogeneous system architecture next to general-purpose CPUs and a heterogeneous shared memory systems, which will allow comparisons between the trade-offs of complexity and flexibility.

Appendix B

Core Level Extensions for Fast Fourier Transform

FFTs are fundamentally linked to the underlying mathematics of many areas of computational science. They are perhaps the most important single tool in “signal processing” and analysis, and play a fundamental role in indirect imaging technologies, such as synthetic aperture radar [24] and computerized tomographic imaging [67]. FFTs are a widely-used tool for the fast solution of partial differential equations, and support fast algorithms for the multiplication of very large integers. Unlike GEMM, the FFT has a more modest number of computations per data element (this is one of the main reasons that it is “fast”), so that performance of FFT algorithms is typically limited by the data motion requirements rather than by the arithmetic computations.

For both the GEMM and FFT algorithms, application-specific designs have been proposed that promise orders of magnitude improvements in power/area efficiency relative to general-purpose processors [92, 105]. However, each of these have been isolated and dedicated design instances limited to one algorithm. With full-custom design increasingly becoming cost-prohibitive, there is a need for solutions that have enough flexibility to run a range of operations at the efficiency of full-custom designs. In this appendix, we analyze the similarities between algorithms and show how one might transform an optimized GEMM core to an FFT core. We consider whether a combined core that can perform either operation efficiently is practical, and analyze the loss in efficiency required to achieve this flexibility.

We begin by exploring FFT algorithms that may be suitable for the baseline LAC architecture. After evaluating LAC limitations and trade-offs for possible solutions, we introduce an "FFT core" that we have optimized for FFTs over a wide range of vector lengths. While optimized for performing FFTs, this core is based on a minimal set of modifications to the existing LAC architecture. We then take similarities between the original LAC and the FFT-optimized design to introduce a flexible, hybrid design that can perform both of these applications efficiently. Comparing both full-custom designs with our proposed hybrid core, we demonstrate the costs of flexibility versus efficiency.

B.1 Related Work

The literature related to fixed-point FFT hardware in the digital signal processing domain is immense. Literature reviews of hardware implementations date back to 1969 [16] – only four years after the publication of the foundational Cooley-Tukey algorithm [29].

The literature related to floating-point FFT hardware is considerably more sparse, especially for double-precision implementations. Important recent work includes the automatic generation of hardware FFT designs from high-level specifications [92]. These hardware designs can be used in either ASIC or FPGA implementations [27], but the published double-precision results for these designs are currently limited to FPGAs [9]. Hemmert and Underwood [56] provide performance comparisons between CPU and FPGA implementations of double-precision FFTs, and include projections of anticipated performance. Finally, a broad survey of the power, performance, and area characteristics of single-precision FFT performance on general-purpose processors, GPUs, FPGAs and ASICs is provided by Chung [27].

Performance of FFT algorithms varies dramatically across hardware platforms and software implementations, depending largely on the effort expended on optimizing data motion. General-purpose, microprocessor-based systems typically deliver poor performance, even with highly optimized implementations, because the power-of-2 strides of the FFT algorithms interact badly with set-associative caches, with set-associative address translation mechanisms, and with power-of-2-banked memory subsystems.

We compare the performance, area, and power of our proposed designs with a sampling of floating-point FFT performance results on general-purpose processors, specialized computational accelerators, and GPUs.

B.2 FFT Algorithm Mapping

In this section, we show the details of mapping an FFT on the LAC along with the required modifications that need to be made to the existing core architecture. We start by focusing on small problems that fit in the local core memory. Then, we present solutions for bigger problems that do not fit in the local store.

B.2.1 Radix-4 FFT Algorithms on the PEs

In Section 6.2.1 we gave a description of regular and FMA optimized versions of the Radix-2 and Radix-4 butterfly operations. Here, we show the details of mapping such operations on the PEs. A Radix-2 operation takes six FMA operations. Performing Radix-2 operations in each PE, the LAC can perform 32-point FFTs, but can only hide the latency of FMA pipeline for FFT transforms with 64 or more points. The Radix-4 butterfly on the PE is more complicated due to data dependencies within the butterfly operation. Figure B.1 shows the DAG of the Radix-4 butterfly. Solid ellipse nodes take 4 FMA operations and dashed nodes take 2 FMA operations. A pipelined FMAC unit has q pipeline stages with $q = 5 \sim 9$. The nodes in the DAG should be scheduled in a way that data dependency hazards do not occur due to pipeline latency. However, the FMAC units have single cycle accumulation capabilities. Hence, no data dependency hazards can occur among addition/accumulations (dashed arrows). For the multiplication dependencies (solid arrows), there should be at least q cycles between start of a child node and the last cycle of its parent. The start-finish cycle numbers next to each node show an execution schedule that tolerates pipeline latencies of up to 9 cycles with no stalls, thus providing 100% FMA utilization.

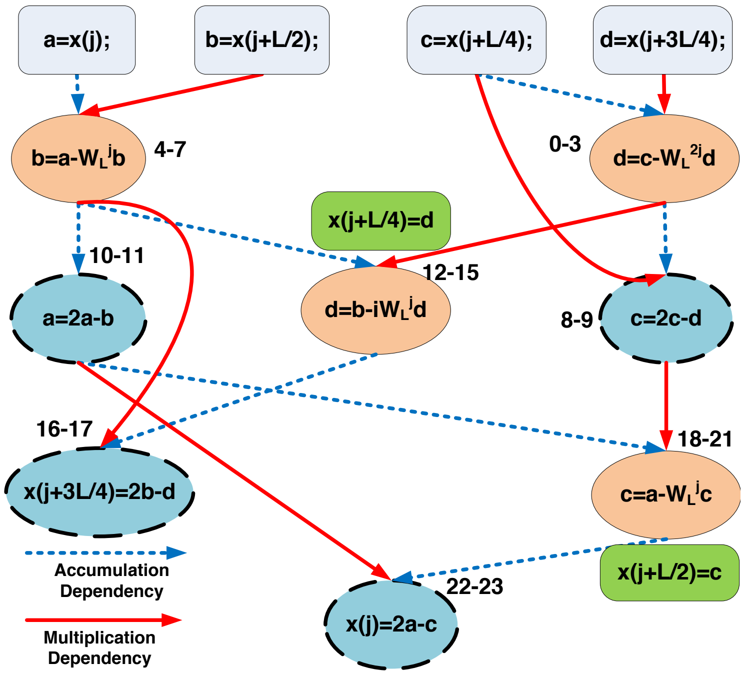


Figure B.1: DAG of the optimized Radix4 Butterfly using a fused multiply-add unit. Rectangles on top indicate the input data, solid nodes show complex computations with four FMA operations each, nodes with dashed lines show complex computations with two FMA operations each. The nodes are executed in an order that avoids data dependency hazards due to pipeline latencies, as shown by the start-finish cycle numbers next to each node.

B.2.2 FFT on the Core

Here, we describe both Radix-2 and Radix-4 based FFTs on the LAC. We compare the computation and communication of these two options, including the bus access behavior.

Radix-2 based FFT When PEs perform Radix-2 butterfly operations, each PE has to exchange one of its outputs with its neighbor of distance 2^0 (one) after the first stage. All PEs on the same row perform communication between PE_{2n} and PE_{2n+1}. After the second stage, PEs exchange outputs with those of neighbors at a distances of 2^1 (two). These PEs also fall on the same row of the 4×4 arrangement of the LAC. After the third stage, each PE exchanges its output with a PE that has a distance of 2^2 (four). In our architecture, with $n_r = 4$, this translates to adjacent neighbors on the same column. Finally, after the fourth stage, each PE switches its outputs with the PE that has a distance of 2^3 (eight). This also requires a column bus communication. In subsequent stages, the distances are multiples of $4^2 = 16$. In a 4×4 arrangement, these are mapped to the same PE. Therefore, there is no communication between PEs for these stages.

The shortcoming of performing Radix-2 butterflies on the PEs comes from a computation/communication imbalance. In stages two through four, broadcast buses are being used for exchanging data. For each exchange, n_r complex numbers are transferred on the bus, which takes $2n_r$ (eight) cycles. Since computation requires only six cycles, this imbalance decreases utilization by an undesirable 25%.

Radix-4 based FFT The Radix-4 algorithm is similar to the Radix-2 algorithm, but with more work done per step and with communication performed over larger distances in each step. Figure B.2 shows a 64-point FFT where each PE performs Radix-4 butterfly operations. This transform contains three stages. The communication pattern for the first PE in the second and third stages is shown with highlighted lines in the figure. In the second stage, PE_0=PE(0,0) has to exchange its last three outputs with the first outputs of its three neighboring $\text{PEs}_{(1,2,3)\times 4^0}$, or PE_1=PE(0,1), PE_2=PE(0,2), and PE_3=PE(0,3) (See figure B.3). Similarly, in the third stage, PE(0,0) has to exchange its last three outputs with the first outputs of PEs that have distance with multiples of 4 or $\text{PEs}_{(4,8,12)} = PE_{(1,2,3)\times 4^1}$, or PE_4=PE(1,0), PE_8=PE(2,0), and PE_{12}=PE(3,0). Since there are only 16 PEs in a core, PEs that have distances of multiples of $4^2 = 16$ fall onto the same PE, and there is no PE-to-PE communication. When communication is required, all the PEs on the same row or column have to send and receive a complex number to/from each of their neighbors. The amount of data that needs to be transferred between PEs is $2n_r(n_r - 1)$. For the case of $n_r = 4$, the communication takes 24 cycles, which exactly matches the required cycle count for the radix-4 computations. As such, the remainder of the appendix will focus on the Radix-4 solution only.

The approach used for the 64-point FFT can be generalized to any (power of 4) size for which the data and twiddle factors fit into the local memory of the PEs. Consider an $N = 4^m$ point FFT using the Radix-4 butterfly implementation described above. The transform includes $log_4^N = m$ stages. Out of these m stages, only two use broadcast buses for data transfer – one stage using the row buses and one stage using the column buses. The rest of data reordering is done by address replacement locally in each PE. Therefore as discussed in the next section, as the transform size increases, the

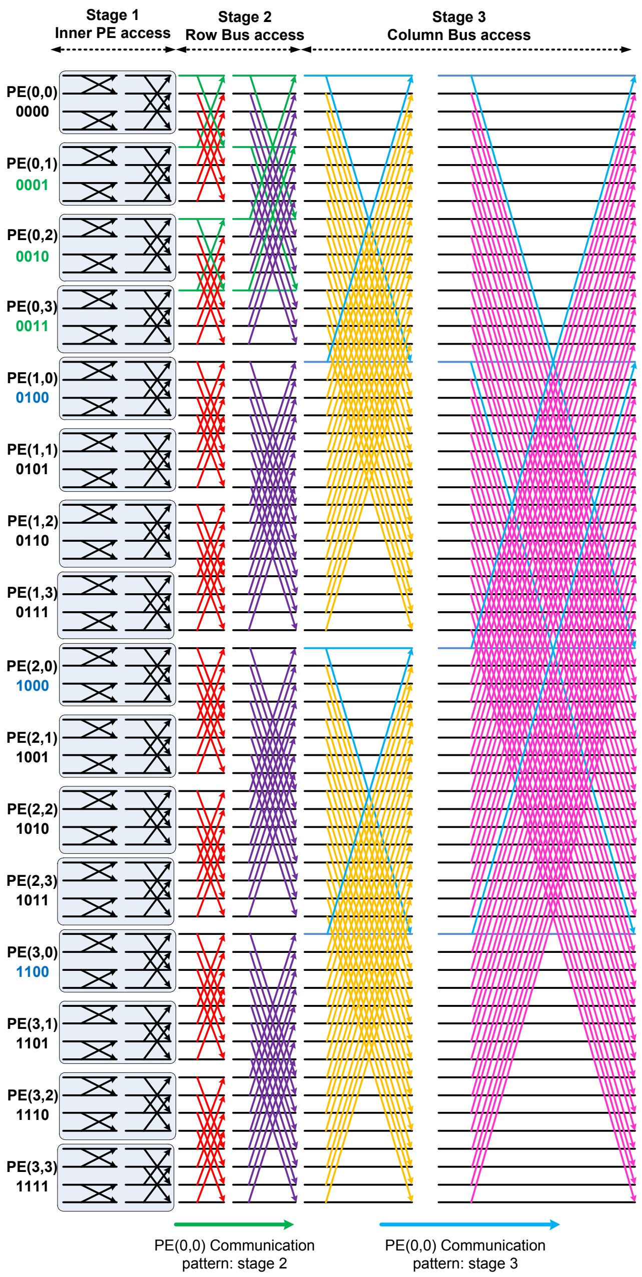


Figure B.2: 64 point FFT performed by 16 PEs in the core. Each PE is performing Radix-4 Butterfly operations. The access patterns for PE(0,0) are highlighted. Stage 2 only utilizes row-buses to perform data communications. Stage 3 only utilizes column-buses to perform data communications.

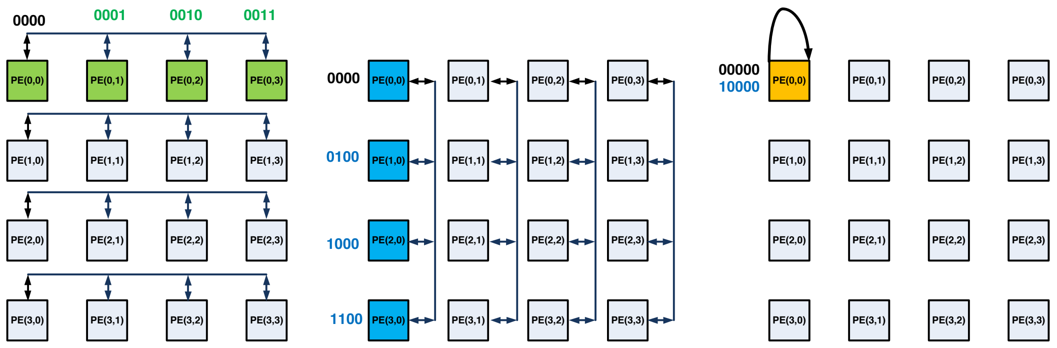


Figure B.3: Data communication access pattern between PEs of the LAC for Radix-4 FFT.

broadcast buses are available for bringing data in and out of the LAC for an increasing percentage of the total time.

For larger problem sizes, the radix computations can be performed in a depth-first or breadth-first order (or in some combination). We choose the breadth-first approach due to its greater symmetry and simpler control. In this approach, all the butterflies for each stage are performed before beginning the butterflies for the next stage.

B.2.3 FFT Memory Hierarchy for Larger Transform Sizes

The local memory in the PEs will allow storage of input data, output data, and twiddle factors for problems significantly larger than the 64-element example above, but the local memory size will still be limited. We will use 4096 as a “typical” value for the maximum size that can be transformed in PE-local memory, but we note that this is a configurable parameter.

Given a core capable of computing FFTs for vectors of length $64, \ldots, 4096$, it is of interest to explore the off-core memory requirements to support the data access patterns required by these small FFTs as well as those of more general transforms, such as larger 1D FFTs or multidimensional FFTs. This analysis is limited to on-chip (but off-core) memory. Considerations for off-chip memory are out of scope of this document and are deferred to future work.

First, we note that the butterfly computations shown in Figure B.2 produce results in bit-reversed order. Although some algorithms are capable of working with transformed results in permuted orders, in general it is necessary to invert this permutation to restore the results to their natural order. Converting from bit-reversed to natural order (or the converse) generates many power-of-two address strides, which are problematic for memory systems based on power-of-two banking with multi-cycle bank cycle times. The most straightforward solutions are based on high-speed, multi-port SRAM arrays, capable of sourcing or sinking contiguous, strided, or random addresses at a rate matching or exceeding the bandwidth requirement of the core. Each of the solutions discussed below will be capable of handling the bit-reversal transformation, as well as any other data access patterns required.

Algorithm for Larger 1D FFTs Support for larger one-dimensional FFTs is provided through the generalized Cooley-Tukey factorization, commonly referred to as the “four-step” algorithm [15]. For an FFT of length N, we split the length into the product of two integer factors, $N = N_1 N_2$. The 1D discrete Fourier transform can then be computed by the sequence: (1) Perform N_1 DFTs of size N_2; (2) Multiply the result by an array of complex

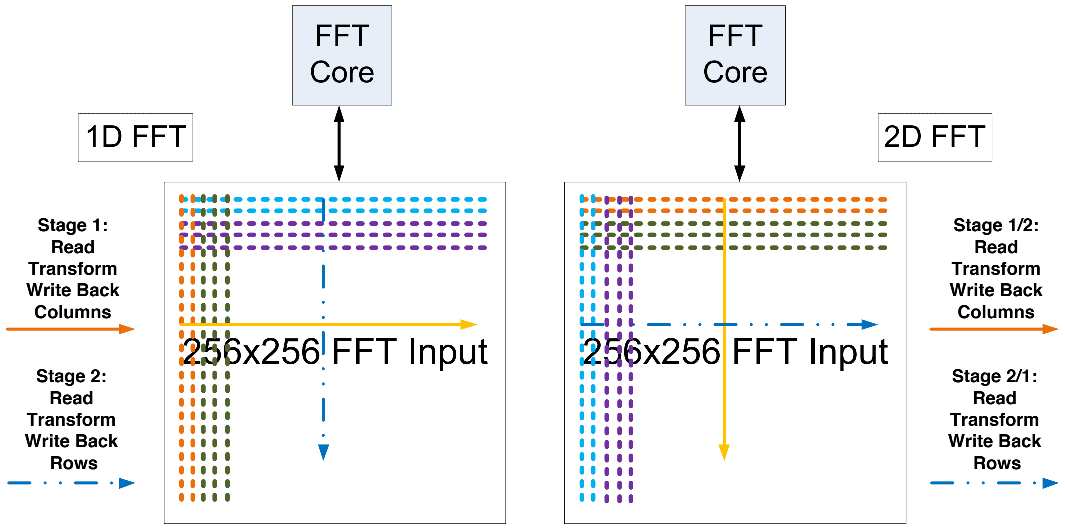

FFT Core
1D FFT
Stage 1: Read Transform Write Back Columns
Stage 2: Read Transform Write Back Rows
256x256 FFT Input
FFT Core
2D FFT
Stage 1/2: Read Transform Write Back Columns
Stage 2/1: Read Transform Write Back Rows
256x256 FFT Input

Figure B.4: Overview of data motion to/from the core for performing a 64K 1D FFT (left), and for a 256×256 2D FFT (right).

roots of unity (called "twiddle factors"); (3) Perform N_2 DFTs of size N_1. For a core capable of performing transforms of up to N=4096, this algorithm allows computing a 1D transform for lengths of up to $4096^2 = 2^{24} \simeq 16$ million elements. (On-chip memory capacity will not be adequate for the largest sizes, but the algorithm suffices for this full range of sizes.)

The overall data motion for the 1D and 2D FFTs is shown in Figure B.4. For the 1D FFT, the first set of DFTs must operate on non-contiguous data – essentially the "columns" of a row-major array. In our design, the data is loaded from these non-contiguous locations in the on-chip memory into the core using a stride of N_2 complex elements, as indicated in the left panel of Figure B.4.

After each column is loaded, the core transforms the data in its local memory as described in the previous section. Note that since the columns are all of the same length, the twiddle factors for these transforms can be held in PE-local memory and re-used for every column. The results of the transform are written back to their original locations in the SRAM array while applying a bit-reversal permutation to restore them to natural order.

After the first set of transforms, the 1D FFT requires multiplication by an additional set of twiddle factors, which are loaded from a second SRAM array. Next, the 1D FFT requires a second set of DFTs to be performed along the “rows”. For this second set of transforms the data is loaded from contiguous locations in SRAM to the cores. It is then transformed and written back to its original location in the SRAM after applying a bit-reversal permutation.

This completes computations for the 1D FFT, but the results are, at this point, stored in the transpose of the natural order. Given the ability of the SRAM to source data in arbitrary order, it is assumed that subsequent computational steps will simply load the data using transposed addressing. Note that this requires that the subsequent processing step knows how the original N was decomposed into the product of N_1 and N_2.

Algorithm for 2D FFTs For a core capable of computing 1D FFTs of lengths $64, \ldots, 4096$, two-dimensional FFTs of sizes up to 4096×4096 are straightforward. These transforms are similar to large 1D FFTs, but are simpler to implement since there are no additional “twiddle factors” required. The data motion for the 2D FFT is also shown in Figure B.4. The row and

FFT $N \times N$	2D No-Ov	2D Ov	1D No-Ov	1D Ov
Core Local Store	$4N$	$6N$	$6N$	$8N$
Radix-4 Cycles	$6Nlog_4^N/n_r^2$			
Twiddle Mult Cycles	-	-	$6N/n_r^2$	$4N/n_r^2$
Communication	$4N = 2N$(R)+$2N$(W)		$6N = 4N$(R)+$2N$(W)	

Table B.1: Different FFT core requirements for both overlapped and non-overlapped versions of $N \times N$ 2D and N^2 1D FFTs.

column transforms can be performed in either order, but choosing to perform the column transforms first emphasizes the similarity with a 1D FFT that is decomposed into the same 2D layout. The column data is read into the cores using a stride of N_2 elements, then transformed and written back to its original location in the SRAM (using bit-reversal to obtain natural ordering). Then the rows are processed in a similar fashion and written back to their original locations in the SRAM. In this case the output contains the transform in the natural ordering, so subsequent processing steps can read the data contiguously.

B.3 Architecture Trade-offs and Configurations

In previous sections, we provided the fundamentals for mapping a Radix-4 based FFT transform to a modified LAC. In this section, we describe the necessary modifications to the PEs, the core, and the off-core SRAM to support the efficient mapping of FFTs. We first describe analytical models before demonstrating the tradeoff analysis using them.

B.3.1 Analytical models

The number of PEs in each row/column is denoted with $n_r(=4)$ and problem sizes are chosen in the range of $N = 64, \ldots, 4096$. Each FMA-optimized Radix-4 butterfly takes 24 cycles as presented in Section 6.2.1. Therefore, an N-point FFT requires a cycle count of $Total_{Cycles} = N/4 \times 24 \times log_4^N/n_r^2$.

We consider two cases in our analysis for FFT on the core: no or full overlap of communication with computation. Note that the FFT operation has a much higher ratio of communication/computation $(O(N)/O(N \log N))$ compared to a typical level-3 BLAS operation like matrix multiplication $(O(N^2)/O(N^3))$. Therefore, the non-overlap FFT solution suffers significantly resulting in low utilization. The different cases of the core requirements are presented in Table B.1.

Core constraints for 2D FFTs For both stages of the 2D FFT and the first stage of the 1D FFT, each core is performing independent FFT operations on rows and columns. The twiddle factors remain the same and therefore the core bandwidth and local store size can be calculated as follows. The amount of data transfer for a problem of size N includes N complex inputs and N complex transform outputs resulting in a total of $4N$ real number transfers. In case of no overlap, data transfer and computation are performed sequentially. For the case of full overlap, the average bandwidth for a FFT of size N can be derived from the division of the total data transfers by the total computation cycles as

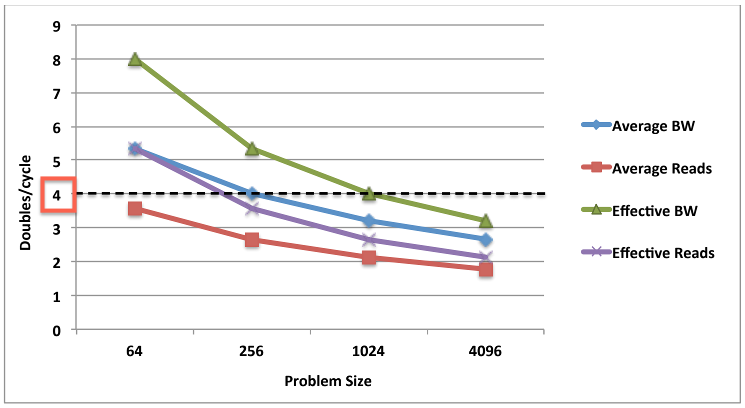


Figure B.5: required bandwidth to support full overlap in the worst case for different problems. Note that four doubles/cycle is the maximum capacity of a core with column buses used for external transfers.

$BW_{Avg} = 2n_r^2/3\log_4^N$. However, out of log_4^N stages, stage 2 utilizes row buses and stage 3 uses column buses for inter-PE communications. If column buses are used to bring data in and out of the PEs, the effective required bandwidth is increased to $BW_{eff} = 2n_r^2/3(\log_4^N - 1)$.

The aggregate local store of PEs includes the N complex input points and N complex twiddle factors. In the no-overlap case, this amount of storage suffices since there is no need for extra buffering capacity. However, the overlapped case requires an extra N point buffer to hold the prefetched input values for the next transform. Therefore, the aggregate PE local stores in a core should be $6N$ floating-point values.

Core constraints for 1D FFTs The second set of FFTs in the “four-step” 1D FFT, require more input bandwidth to the cores. Each core is performing independent FFT operations on rows. The twiddle factors are changing with

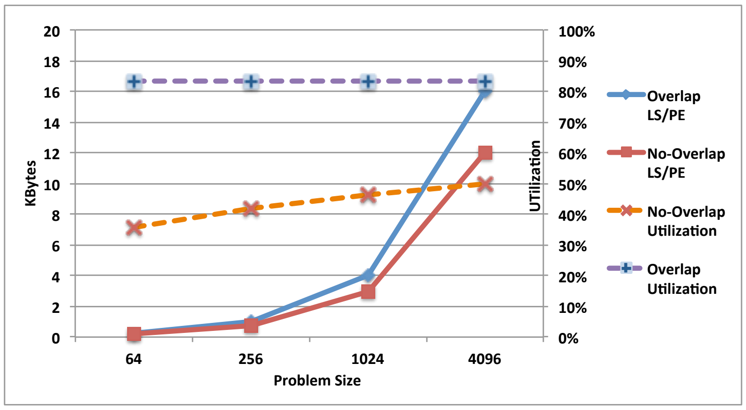


Figure B.6: Local store/PE and respective utilization for both cases of non-overlap and overlapped solutions.

each new N point input vector. However, each twiddle factor is going to be multiplied with the corresponding input before the FFT computation gets started. An extra $4N$ real multiplications are added to the total computations of this transform. Therefore the total cycle count is $Total_{Cycles} = (6Nlog_4^N + 4N)/n_r^2$. The amount of data transfer for a problem of size N includes $2N$ complex inputs (transform inputs and twiddle factors), and N complex outputs resulting in a total of $6N$ real number transfers. In case of no overlap, data transfer and computation are performed sequentially. For the case of full overlap, the average bandwidth for an FFT of size N can be derived from the division of the total data transfers by the total computation cycles as $BW_{Avg} = 3n_r^2/(3\log_4^N + 2)$. However, If column buses are used to bring data in and out of the PEs, the effective required bandwidth is increased (as in the 2D case described above) to $BW_{eff} = 3n_r^2/(3\log_4^N - 1)$ (see Figure B.5).

Each N-point input to the core has to be pre-multiplied by a different

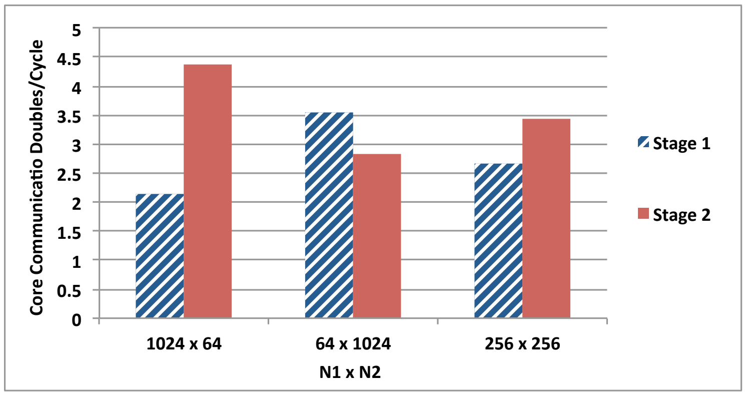


Figure B.7: Average communication load on core for 64K 1D FFT.

set of twiddle factors, so another buffer is needed for the corresponding twiddle factors.

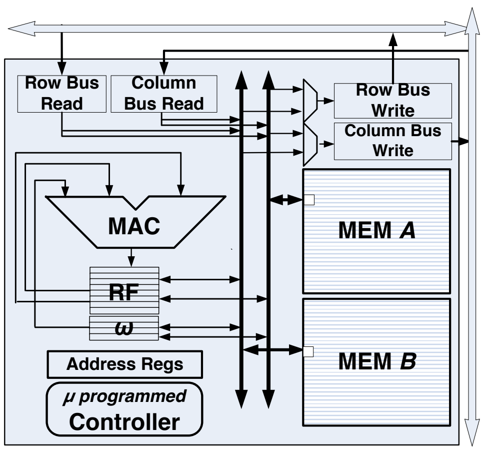
Finally, as described earlier, one can compute the 1D discrete Fourier transform by splitting N into the product of two integer factors, $N = N_1 \times N_2$. Earlier we noted that the fully-overlapped solution has lower communication load for larger transform lengths. Noting also that the second set of FFTs put more communication load on the core/external memory, we expect that ordering the factors so that the larger factor corresponds to the length of the second set of transforms will provide more balanced memory transfer requirements. Figure B.7 demonstrates this effect for the case of a 64K point 1D FFT with three different options for $64K = N_1 \times N_2$.

B.3.2 Core Configuration

Figure B.6 shows the core bandwidth and local store requirements for the overlapped and non-overlapped algorithms. The utilization of the non-overlapped version increases from 35% to 50% as the size of the transform increases. The overlapped algorithm can fully utilize the FMA units for all these sizes, maintaining its optimum utilization of slightly over 83.3%. Depending on the FFT type (1D or 2D), the overlapped algorithm requires 33%$\sim$50% extra local storage.

Note that the non-overlapped bandwidth is assumed to be at a fixed rate of four doubles/cycles, which is the maximum capacity of the LAC. However, for the overlapped algorithm at problem sizes $N <= 1024$, extra off-core bandwidth is required to attain the peak achievable efficiency. The chart on the left side of Figure B.5 shows that the maximum required off-core bandwidth does not exceed eight doubles/cycle. Therefore, the off-core bandwidth needs to double that of the original LAC design. Furthermore, the PE must be able to overlap the prefetching of input data and the post-storing of output data from/to off-core memory concurrently with the computations.

Doubling the memory bandwidth could be implemented in three ways: doubling the width of the column buses, doubling the number of column buses, or connecting the row buses to the off-core memory. The first choice would be complex to implement, since the original column bus bandwidth is matched to the PE-local SRAM bandwidth. The second choice is not quite as complex, but still requires an expansion of the PE local SRAM bandwidth. The best solution is therefore to expand the memory interface so that both row and column buses can transfer data to/from PEs. This solution does not impose any area overhead for additional broadcast buses and provides an interface



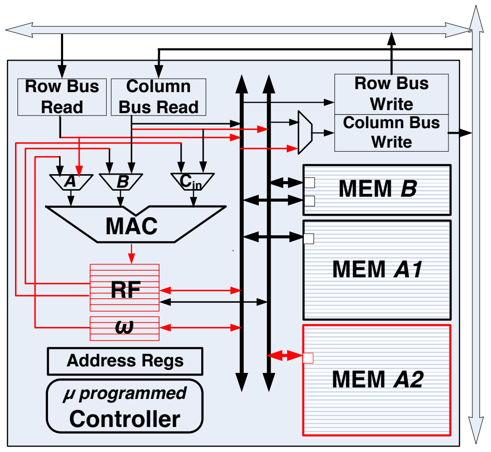


Figure B.8: New PE configurations for full-overlap FMA-optimized Radix-4 FFT: (left) FFT-optimized PE with two 8-byte, single-ported SRAMs (right) Modified linear algebra PE with two 8-byte, single-ported SRAMs to contain matrix A ("Hybrid").

to the memory that is always free of inter-PE use during phases in which the column buses are busy with inter-PE transfers. Further, this design is symmetric and natively supports transposition.

B.3.3 PE Configuration

The PE micro-architecture must perform the three tasks of Radix-4 butterfly computation, FFT communication, and off-core communication concurrently. Some extra logic and storage is needed to facilitate data movements and locality. These options are described with the help of Figure B.9.

An 8-byte register file is needed to store the four complex input, temporary, and output values of the FMA-optimized Radix-4 butterfly (Figure B.1). The twiddle factors take an extra four registers. We separate these two reg-

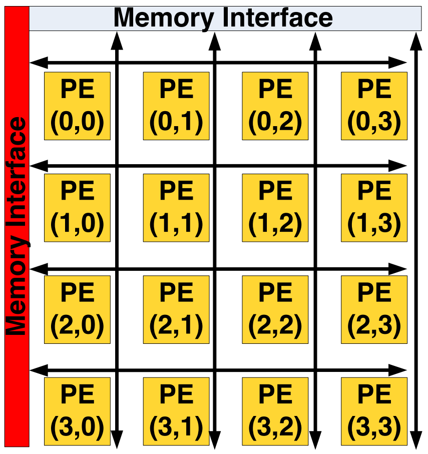


Figure B.9: New core configurations with extended external row bus interface for full-overlap FMA-optimized Radix-4 FFT.

ister files to avoid adding extra ports to the existing (large) register file and hence save energy and area. The PE SRAM needs enough bandwidth to provide data for both Radix-4 computations and off-core communications. Each butterfly has six complex inputs and produces four complex outputs. This data transfer would require 20 cycles from a typical single-ported 8-byte wide SRAM. The remaining four cycles of the 24-cycle radix-4 compute phase do not provide enough time to implement the required off-core communications. There are three solutions to provide the required bandwidth to the PE-local stores: an extra port to the same PE SRAM could be added, a wider (16 byte wide) port could replace the existing port, or a separate SRAM block with its own 8-byte port can be added.

A simple study of memory power and area consumption of these options is presented in Table B.2. The dual ported solution consumes much more power and area than the other two. Hence, the wide solution needs extra buffering and a more complicated control to transmit data to/from other components. The two SRAM solution is the best one with the simplest control. This FFT PE is presented in Figure B.8(left). It has a symmetric design with two separate buses – each is connected to all the components in the PE and to one of the SRAMs.

So far, we have described the options for an FFT PE that is based on the baseline architecture but is specifically designed for FFT operation. If one starts with an existing linear algebra PE to make a hybrid FFT/Linear Algebra architecture, the register file design has to be extended with more ports and more capacity to match the requirements of the FFT. There are two options for extending this micro-architecture to facilitate FFT bandwidth for the hybrid design. The original linear algebra PE has one larger, single-ported SRAM and one smaller, dual-ported SRAM. Since the smaller SRAM is already dual ported, we must modify the larger SRAM to provide extra bandwidth. As discussed above, the best solution is to divide the larger SRAM into two halves and adding an extra bus to the PE (see Figure B.8(right)).

B.3.4 Off-core Memory Configuration

As noted in Section B.3, the maximum core bandwidth required for the non-overlapping case is four double-precision elements per cycle. The non-overlapped configuration requires an effective bandwidth of up to eight double-precision elements per cycle for problems sizes smaller than N=1024.

16Kbyte SRAM	**Wide**	**Dual-port**	**Separate**
# SRAMs, # ports x bus-width	1,1x16	1, 2x16	2, 1x8
Cycle time (nS)	0.73	0.79	0.67
Energy per access (nJ)	0.010	0.009	0.005
Total area (mm^2)	0.054	0.141	0.054
Max Power at Target Freq (mW)	0.010	0.017	0.010
Worst case FFT Access/Cycle	0.613	1.227	1.227
Worst Case FFT total dynamic energy (J)	0.006	0.011	0.006

Table B.2: PE SRAM options and their area, performance, and energy consumption report by CACTI [93].

Core changes are required to support external bandwidths above four double-precision values per cycle, with the addition of memory interfaces on the row buses providing the most symmetric solution. The effective bandwidth required for pre-fetch/post-store is decreased by opening up more cycles in which at least one of the buses is not used.

For the case of double-precision complex data, the natural data size is $2 \times 64 = 128$ bits, so we will assume 128-bit interfaces. As shown in section 6.2.1, the first step of a large 1D FFT requires less memory traffic than the second stage which includes loading an additional set of twiddle factors, so we focus on the second stage here. We consider whether the instantaneous read and write requirements of the algorithm can be satisfied by two separate memories, one for data (SRAM0) and one for twiddle factors (SRAM1), each with a single 128-bit-wide port operating at twice the frequency of the core, giving each a bandwidth of 4 double-precision elements per cycle.

The worst case occurs for $N = 64$, where full overlap of data transfers with computation requires that the external memory be able to provide

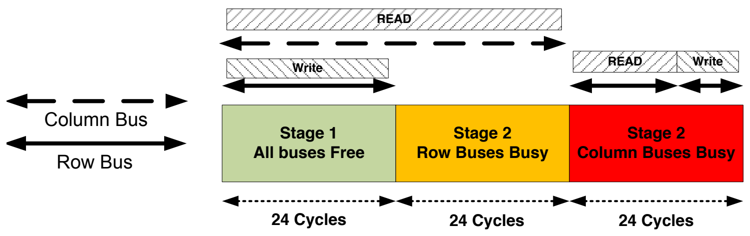


Figure B.10: Schematic of data bus usage for fully overlapped pre-fetch/post-store for the worst case of a 64-element FFT.

256 double-precision elements (64 complex input elements plus 64 complex twiddle factors) and receive 128 double-precision elements (64 complex output elements) during the 72 cycles required to perform the three radix-4 computations. The proposed memory interface bandwidth is clearly adequate to provide for the average traffic – the SRAMs require 64 cycles (of the 72 available) to provide the 256 words prefetched by the core for its next iteration. The writes require only 32 of the 72 cycles, and these can be overlapped with the reads.

The detailed scheduling is not particularly complex, but does require careful design, as shown in Figure B.10. Recall (Figure B.2) that during the first radix-4 step (24 cycles) of the 64-point FFT neither row nor column buses are in use, while the row buses are in use during the second (24-cycle) radix-4 step and the column buses are in use during the third 24-cycle radix-4 step. The SRAMs requires 64 cycles to source the input data and twiddle factors, so reads must occur during all three of these phases, with reads on the column buses during phase 2 (while the row buses are busy) and on the row buses during phase 3 (while the column buses are busy). Similarly, the writes require 32 cycles at the SRAM, so they must occur during at least two of the three phases. Since the data is both read from and written to SRAM0, the reads during the 24 cycles of "stage 1" of Figure B.10 must be reads of twiddle factors from SRAM1. The remaining 8 cycles of twiddle factor reads can occur during either stage 2 or stage 3.

If we further assume that only a single SRAM bank within each PE is available for this pre-fetch/post-store communication (with the other bank being used for the concurrent computation step), then a PE can read or write to a row or column bus, but cannot use both row and column buses in the same cycle without additional buffering. Fortunately, due to the shared bus architecture, each PE can only write to the column bus on 1/4 of the cycles and can only write to the row bus on 1/4 of the cycles, so it is straightforward to swizzle the active PEs so that no PE is both reading and writing on the same cycle. For all cases with $N > 64$, there are additional radix-4 stages with no use of the row and column buses, making full overlap of communication and computation easier to schedule.

B.4 Experimental Results and Implementations

In this section, we present area, power and performance estimates for the LAC with the modifications introduced in previous sections.

Table B.3 reports the projected power and area consumption of the components of the PE for the three different designs, along with the corresponding design metrics. The power consumption of the FFT design is con-

PE Design	LAC	FFT	Hybrid
SRAM			
Total SRAM Area (mm^2)	0.070	0.054	0.073
Total SRAMs MAX Power (W)	0.013	0.010	0.015
Total SRAM Actual Dynamic Power (W)	0.005	0.006	0.006
Floating-Point Unit			
FP Area (mm^2)	0.042	0.042	0.042
FP Power (W)	0.031	0.031	0.031
Register File			
RF Area (mm^2)	0.000	0.008	0.008
RF MAx Power (W)	0.000	0.004	0.004
RF Actual Power (W)	0.000	0.003	0.003
Broad-cast Buses			
Bus Area /PE (mm^2)	0.014	0.014	0.014
Max Bus Power (W)	0.001	0.001	0.001
PE Total			
Total PE Area (mm^2)	0.126	0.119	0.138
Total PE MAx Power (W)	0.045	0.047	0.052
Total PE Real Power (W) (GEMM,FFT)	0.037	0.041	(0.037, 0.041)
GFLOPS/W (GEMM, FFT)	53.82	40.53	(53.80, 40.50)
GFLOPS/MAX W (GEMM, FFT)	44.59	35.80	(38.55, 32.12)
GFLOPS/mm^2 (GEMM, FFT)	15.84	14.00	(14.54, 12.11)
W/mm^2	0.334	0.391	0.377

Table B.3: PE designs for dedicated LAC, dedicated FFT, and a hybrid design that can perform both operations.

sidered for the worst case and highest possible number of accesses. For the hybrid design, we report a pair of numbers, one for GEMM and one for FFT.

Figures B.11, and B.12 summarize the actual and maximum power consumption breakdown of the three proposed designs respectively. For the pure FFT and hybrid cores, the "actual" power considers the worst case power consumption when running an FFT. The maximum power breakdown shows the "maximum power" that is used by the three different PE designs. We observe that the power consumption is dominated by the FMAC unit, with secondary contributions from the PE-local SRAMs. Since the leakage power consumption of the SRAM blocks are negligible, the actual power efficiency is maintained in the hybrid PE.

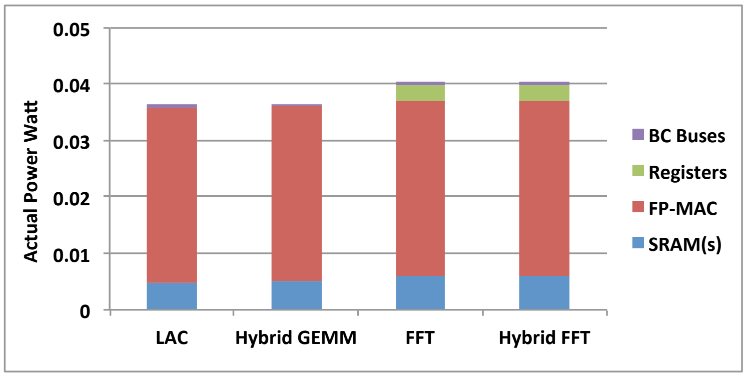


Figure B.11: Actual PE power consumption of each design for target applications at 1GHz.

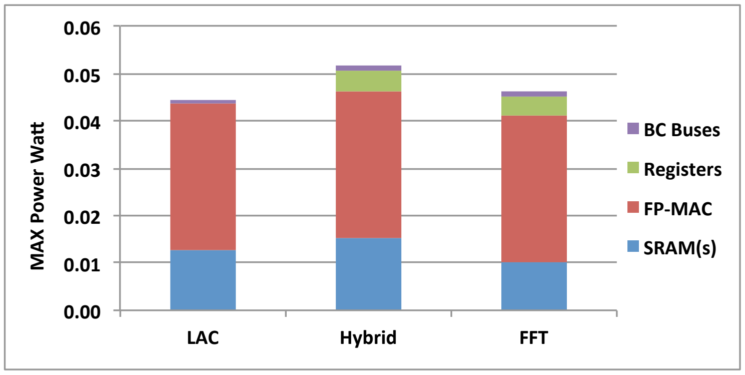


Figure B.12: Maximum PE power consumption of each target design at 1GHz.

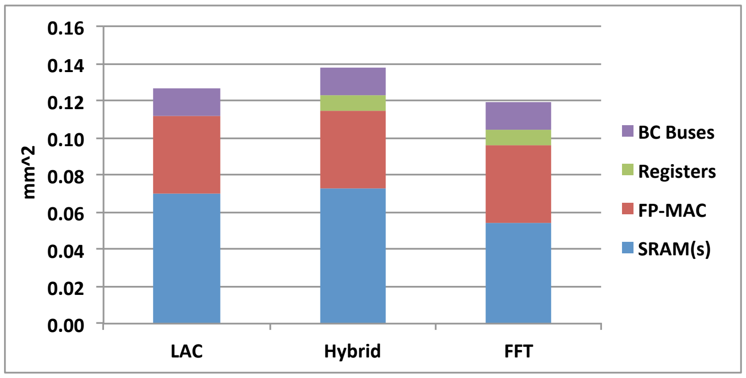


Figure B.13: Total area breakdown of the PE for each design.

Finally, the area breakdown in Figure B.13 emphasizes that most of the PE area is occupied by the memory blocks. The hybrid design has the largest aggregate PE SRAM capacity.

B.5 Summary

Starting with a baseline linear algebra architecture, this appendix presents analysis and modication of the design to efficiently support 1D and 2D complex FFT algorithms. We presented a hybrid core that can perform both algorithms while maintaining the efficiency characteristic of the original application-specc design. Our results show that this hybrid design can achieve up to 40 GFLOPS/W power efficiency for double-precision complex FFTs with 83% effective utilization of the FMAC units.

Bibliography

[1] Pentium® III Processor die photo fact sheet. Technical report.

[2] Fermi computer architecture white paper. Technical report, NVIDIA, 2009.

[3] Intel® Math Kernel Library. User's Guide 314774-009US, Intel, 2009.

[4] Samsung DDR3 SDRAM:High-Performance, Energy-Efficient Memory for Todays Green Computing Platforms. Technical Report BRO-10-DRAM-001, SAMSUNG Green Memory, March 2009.

[5] CSX700 Floating Point Processor. Datasheet 06-PD-1425 Rev 1, ClearSpeed Technology Ltd, 2011.

[6] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM Journal of Research and Development*, 1994.

[7] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. Qr factorization on a multicore node enhanced with multiple gpu accelerators. In *2011 IEEE International Parallel Distributed Processing Symposium (IPDPS)*, pages 932–943, 2011.

[8] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, volume 180, 2009.

[9] B. Akin, P. A. Milder, F. Franchetti, and J. C. Hoe. Memory bandwidth efficient two-dimensional fast Fourier transform algorithm and implementation for large problem sizes. In *Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, FCCM '12, pages 188–191. IEEE, 2012.

[10] V. Allada, T. Benjegerdes, and B. Bode. Performance analysis of memory transfers and GEMM subroutines on NVIDIA Tesla GPU cluster. *Proceedings of the 2009 IEEE International Conference on Cluster Computing (CLUSTER '09)*, pages 1 – 9, 2009.

[11] D. Altavilla and M. Chiappetta. VIA's Glenn Henry Speaks On New Low Power Isaiah Processor, January 2008.

[12] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. D. Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. SIAM, Philadelphia, PA, USA, 1999.

[13] S. Aslan, E. Oruklu, and J. Saniie. Realization of area efficient QR factorization using unified division, square root, and inverse square root

hardware. In *IEEE International Conference on Electro/Information Technology, 2009. (EIT '09).*, 2009.

[14] D. A. Bader and V. Agarwal. FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine. In *Proceedings of the 14th IEEE International Conference on High Performance Computing (HiPC), Lecture Notes in Computer Science 4873*, pages 18–21, 2007.

[15] D. H. Bailey. FFTs in external of hierarchical memory. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, pages 234–242. ACM, 1989.

[16] G. Bergland. Fast Fourier transform hardware implementations–an overview. *IEEE Transactions on Audio and Electroacoustics*, 17(2):104 – 108, jun 1969.

[17] P. Bientinesi and R. van de Geijn. Representing dense linear algebra algorithms: A farewell to indices. FLAME Working Note #17 TR-2006-10, The University of Texas at Austin, Department of Computer Sciences, 2006.

[18] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[19] J. L. Blue. A portable Fortran program to find the euclidean norm of a vector. *ACM Transactions on Mathematical Software.*, 4(1), 1978.

[20] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 83–94, New York, NY, USA, 2000. ACM.

[21] C. Caşcaval, S. Chatterjee, H. Franke, K. J. Gildea, and P. Pattnaik. A taxonomy of accelerator architectures and their programming models. *IBM Journal of Research and Development.*, 54:473–482, September 2010.

[22] L. E. Cannon. *A cellular computer to implement the Kalman filter algorithm.* PhD thesis, Bozeman, MT, USA, 1969.

[23] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: a performance view. *IBM Journal of Research and Development.*, 51:559–572, September 2007.

[24] M. Cheney, B. Borden, C. B. of the Mathematical Sciences, and N. S. F. (U.S.). *Fundamentals of radar imaging.* CBMS-NSF regional Conference series in applied mathematics. SIAM, Philadelphia, PA, USA, 2009.

[25] J. Choi, J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers.

In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE, 1992.

[26] J. Choi, J. Dongarra, and D. W. Walker. PUMMA: Parallel Universal Matrix Multiplication Algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994.

[27] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-43, pages 225–236, Washington, DC, USA, 2010. IEEE Computer Society.

[28] Clearspeed, Inc. CSX processor architecture. Technical Report PN-1110-0702, Feb 2007.

[29] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[30] J. Demmel and A. Gearhart. Instrumenting linear algebra energy consumption via on-chip energy counters. Technical Report UCB/EECS-2012-168, UC at Berkeley, 2012.

[31] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Dideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very

small physical dimensions. *IEEE Journal of Solid State Ciruits*, 9(5), October 1974.

[32] J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1), 1990.

[33] J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1), 1988.

[34] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, FPGA '05, pages 86–95, New York, NY, USA, 2005. ACM.

[35] M. D. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand. Reciprocation, square root, inverse square root, and some elementary functions using small multipliers. *IEEE Transactions on Computers*, 49(7), July 2000.

[36] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[37] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: a vector extension to the alpha architecture. *Proceedings of 29th Annual International Symposium on Computer Architecture (ISCA'29)*, pages 281 – 292, 2002.

[38] R. Espasa, M. Valero, and J. E. Smith. Out-of-order vector architectures. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO-30, pages 160–170, Washington, DC, USA, 1997. IEEE Computer Society.

[39] R. Espasa, M. Valero, and J. E. Smith. Vector architectures: past, present and future. In *Proceedings of the 12th international Conference on Supercomputing*, ICS '98, pages 425–432, New York, NY, USA, 1998. ACM.

[40] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics hardware*, HWWS '04, pages 133–137, New York, NY, USA, 2004. ACM.

[41] B. Flachs, S. Asano, S. H. Dhong, H. P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. S. Liberty, B. Michael, H.-J. Oh, S. M. Mueller, O. Takahashi, K. Hirairi, A. Kawasumii, H. Murakami, H. Noro, S. Onishi, J. Pille, J. Silberman, S. Yong, A. Hatakeyama, Y. Watanabe,

N. Yano, D. A. Brokenshire, M. Peyravian, V. To, and E. Iwata. Microarchitecture and implementation of the synergistic processor in 65-nm and 90-nm soi. *IBM Journal of Research and Development*, 51:529–543, September 2007.

[42] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving problems on concurrent processors. Vol. 1: General techniques and regular problems*. Prentice-Hall, Inc., 1988.

[43] S. Galal and M. Horowitz. Energy-efficient floating point unit design. *IEEE Transactions on Computers*, PP(99), 2010.

[44] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, page 3, 2005.

[45] O. Garreau and J. Lo. Scaling up to teraflops performance with the virtex-7 family and high-level synthesis. *Xilinx White Paper: Virtex-7 FPGA*, (WP387), February 2011.

[46] P. Gelsinger. Microprocessors for the new millennium: Challenges, opportunities, and new frontiers. In *2001 IEEE International Solid-State Circuits Conference, 2001. Digest of Technical Papers. ISSCC.*, pages 22 –25, 2001.

[47] V. George, S. Jahagirdar, C. Tong, K. Smits, S. Damaraju, S. Siers, V. Naydenov, T. Khondker, S. Sarkar, and P. Singh. Penryn: 45-nm next generation Intel® core$^{\mathrm{TM}}$ 2 processor. *IEEE Asian Solid-State Circuits Conference, 2007. ASSCC '07.*, Jan 2008.

[48] G. H. Golub and C. Van Loan. An analysis of the total least squares problem. *SIAM Journal on Numerical Analysis*, 17(1):883–893, December 1980.

[49] J. Gonzalez and R. C. Nunez. LAPACKrc: fast linear algebra kernels/solvers for FPGA accelerators. *Journal of Physics: Conference Series, Scientific Discovery through Advanced Computing, SciDAC 2009*, (180), 2009.

[50] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277 – 1284, sep 1996.

[51] K. Goto and R. van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software*, 35(1):1–14, 2008.

[52] K. Goto and R. A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):12, May 2008. Article 12, 25 pages.

[53] J. Greene and R. Cooper. A parallel 64k complex FFT algorithm for the IBM/Sony/Toshiba Cell Broadband Engine processor. In *Laboratory, University of Tennessee*, 2005.

[54] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26:10–24, March 2006.

[55] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. ACM.

[56] K. S. Hemmert and K. D. Underwood. An analysis of the double-precision floating-point FFT on FPGAs. In *Proceedings of the 2005 IEEE 13th International Symposium on Field-Programmable Custom Computing Machines*, FCCM '05, pages 171–180, 2005.

[57] B. A. Hendrickson and D. E. Womble. The Torus-Wrap mapping for dense matrix calculations on massively parallel computers. *SIAM Journal on Scientific and Statistical Computing*, 15(5), 1994.

[58] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, USA, second edition, 2002.

[59] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA'11, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society.

[60] S. Hong and H. Kim. An integrated gpu power and performance model. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 280–289, New York, NY, USA, 2010. ACM.

[61] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein. Scaling, power and the future of CMOS. In *Proceedings IEE International Electron Devices Meeting (IEDM)*, Washington, DC, December 2005.

[62] H. V. Jagadish and T. Kailath. A family of new efficient arrays for matrix multiplication. *IEEE Transactions on Computers*, 38(1):149 – 155, 1989.

[63] S. Jain, V. Erraguntla, S. Vangal, Y. Hoskote, N. Borkar, T. Mandepudi, and V. Karthik. A 90mW/GFlop 3.4GHz reconfigurable fused/continuous multiply-accumulator for floating-point and integer operands in 65nm. In *23rd International Conference on VLSI Design, 2010. VLSID '10.*, pages 252–257, 2010.

[64] J.-W. Jang, S. Choi, and V. Prasanna. Energy- and time-efficient matrix multiplication on FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(11):1305 – 1319, 2005.

[65] K. T. Johnson, A. R. Hurson, and B. Shirazi. General-purpose systolic arrays. *Computer*, 26(11):20 – 31, 1993.

[66] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49:589–604, July 2005.

[67] A. Kak and M. Slaney. *Principles of computerized tomographic imaging*. Classics In Applied Mathematics. SIAM, Philadelphia, PA, USA, 2001.

[68] D. Kanter. Inside Fermi: Nvidia's HPC push. Technical report, Real World Technologies, September 2009.

[69] H. Karner, M. Auer, and C. W. Ueberhuber. Top speed FFTs for FMA architectures, 1998.

[70] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 140–151, New York, NY, USA, 2009. ACM.

[71] A. Kerr, D. Campbell, and M. Richards. QR decomposition on GPUs. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, 2009.

[72] M. Kistler, J. Gunnels, D. Brokenshire, and B. Benton. Petascale computing with accelerators. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 241–250, New York, NY, USA, 2009. ACM.

[73] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26:10–23, May 2006.

[74] C. Kozyrakis and D. Patterson. Vector vs. superscalar and vliw architectures for embedded multimedia benchmarks. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pages 283–293, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[75] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. In *Proceedings of the 30th annual international symposium on Computer architecture*, ISCA '03, pages 399–409, New York, NY, USA, 2003. ACM.

[76] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 52–, Washington, DC, USA, 2004. IEEE Computer Society.

[77] V. B. Y. Kumar, S. Joshi, S. B. Patkar, and H. Narayanan. FPGA based high performance double-precision matrix multiplication. In *Proceedings*

of the 22nd International Conference on VLSI Design, VLSID '09, pages 341–346, Washington, DC, USA, 2009. IEEE Computer Society.

[78] H. Kung. Why systolic architectures? *Computer*, 15(1):37 – 46, 1982.

[79] S. Kung. VLSI array processors. *IEEE ASSP Magazine*, 2(3):4 – 22, July 1985.

[80] J. Kurzak, A. Buttari, and J. Dongarra. Solving systems of linear equations on the Cell processor using cholesky factorization. *IEEE Transactions on Parallel and Distributed Systems*, 19:1175–1186, September 2008.

[81] J. Kurzak and J. Dongarra. Qr factorization for the Cell broadband engine. *Journal of Scientific Programming*, 17(1-2):31–42, January 2009.

[82] G. Kuzmanov and W. van Oijen. Floating-point matrix multiplication in a polymorphic processor. *International Conference on Field-Programmable Technology, 2007. ICFPT 2007.*, pages 249 – 252, 2007.

[83] M. Langhammer. High performance matrix multiply using fused datapath operators. *2008 42nd Asilomar Conference on Signals, Systems and Computers*, pages 153 – 159, 2008.

[84] F. Lauginiger, R. Cooper, J. Greene, M. Pepe, M. Prelle, and Y. Steinsaltz. Performance of a multicore matrix multiplication library. *Second Workshop on Software Tools for MultiCore Systems (STMCS 2008)*, Jan 2007.

[85] 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*, 5(3):308–323, Sept. 1979.

[86] C. Lemuet, J. Sampson, J.-F. Collard, and N. Jouppi. The potential energy efficiency of vector acceleration. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[87] J. Li, J. Li, A. Skjellum, I. Banicescu, D. S. Reese, S. M. Bridges, and R. D. Koshel. A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. *Concurrency: Practice and Experience*, Jan 1996.

[88] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-42, pages 469–480, New York, NY, USA, 2009. ACM.

[89] T. Lippert, N. Petkov, P. Palazzari, and K. Schilling. Hyper-systolic matrix multiplication. *Parallel Computing*, 2001.

[90] T. M. Low and R. van de Geijn. An API for manipulating matrices stored by blocks. FLAME Working Note #12 TR-2004-15, The University of Texas at Austin, Department of Computer Sciences, May 2004.

[91] K. K. Mathur and S. L. Johnsson. Multiplication of matrices of arbitrary shape on a data parallel computer. *Parallel Computing*, 20(7):919 – 951, 1994.

[92] P. Milder, F. Franchetti, J. C. Hoe, and M. Püschel. Computer generation of hardware for linear digital signal processing transforms. *ACM Transactions on Design Automation of Electronic Systems*, 17(2), 2012.

[93] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Architecting efficient interconnects for large caches with CACTI 6.0. *IEEE Micro*, 28, 2008.

[94] R. Nath, S. Tomov, and J. Dongarra. An improved MAGMA GEMM for Fermi GPUs. Technical report, LAPACK WN #227, 2010.

[95] Y. Nishikawa, M. Koibuchi, M. Yoshimi, K. Miura, and H. Amano. Performance improvement methodology for clearspeed's csx600. *International Conference on Parallel Processing, 2007. ICPP 2007.*, pages 77 – 77, 2007.

[96] Y. Nishikawa, M. Koibuchi, M. Yoshimi, K. Miura, and H. Amano. An analytical network performance model for SIMD processor CSX600 interconnects. *Journal of Systems Architecture*, 57:146–159, January 2011.

[97] Y. Nishikawa, M. Koibuchi, M. Yoshimi, A. Shitara, K. Miura, and H. Amano. Performance analysis of ClearSpeed's CSX600 intercon-

nects. *Parallel and Distributed Processing with Applications, International Symposium on*, 0:203–210, 2009.

[98] S. F. Oberman. Floating point division and square root algorithms and implementation in the AMD-K7TM microprocessor. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, ARITH'14, 1999.

[99] S. F. Oberman and M. J. Flynn. Design issues in division and other floating-point operations. *IEEE Transactions on Computers*, 46(2), February 1997.

[100] M. Parker. High-performance floating-point implementation using FPGAs. In *Proceedings of the 28th IEEE Conference on Military communications*, MILCOM'09, pages 323–327, Piscataway, NJ, USA, 2009. IEEE Press.

[101] M. Parker. Achieving TeraFLOPS performance with 28nm FPGAs. *EDA Tech Forum*, December 2010.

[102] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSS: a full system simulator for multicore x86 CPUs. In *48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1050–1055, 2011.

[103] S. Patel and W.-M. W. Hwu. Accelerator architectures. *IEEE Micro*, 28(4):4 –12, 2008.

[104] A. Pedram, M. Daneshtalab, N. Sedaghati-Mokhtari, and S. Fakhraie. A high-performance memory-efficient parallel hardware for matrix com-

putation in signal processing applications. In *International Symposium on Communications and Information Technologies, 2006. ISCIT '06.*, pages 473–478, 2006.

[105] A. Pedram, A. Gerstlauer, and R. A. Geijn. A high-performance, low-power linear algebra core. In *Proceedings of the 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, ASAP '11, pages 35–42, Washington, DC, USA, 2011. IEEE Computer Society.

[106] A. Pedram, A. Gerstlauer, and R. van de Geijn. On the efficiency of register file versus broadcast interconnect for collective communications in data-parallel hardware accelerators. In *Proceedings of the 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 19–26, 2012.

[107] A. Pedram, A. Gerstlauer, and R. A. van de Geijn. Floating point architecture extensions for optimized matrix factorization. In *Proceedings of the 2013 IEEE 21st Symposium on Computer Arithmetic*, ARITH '13. IEEE, 2013.

[108] A. Pedram, S. Z. Gilani, N. S. Kim, R. v. d. Geijn, M. Schulte, and A. Gerstlauer. A linear algebra core design for efficient level-3 BLAS. In *Proceedings of the 2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, ASAP '12, pages 149–152, Washington, DC, USA, 2012. IEEE Computer Society.

[109] A. Pedram, M.-R. Jamali, S. Fakhraie, and C. Lucas. Reconfigurable parallel hardware for computing local linear neuro-fuzzy model. In *International Symposium on Parallel Computing in Electrical Engineering, 2006. (PARELEC 2006).*, pages 198–201, 2006.

[110] A. Pedram, J. McCalpin, and A. Gerstlauer. Transforming a linear algebra core to an fft accelerator. In *Proceedings of the 2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 175–184, 2013.

[111] A. Pedram, R. van de Geijn, and A. Gerstlauer. Codesign tradeoffs for high-performance, low-power linear algebra architectures. *IEEE Transactions on Computers, Special Issue on Power efficient computing*, 61(12):1724–1736, 2012.

[112] G. Pfister. The perils of parallel: Why accelerators now? July 2009.

[113] J.-A. Piñeiro and J. D. Bruguera. High-speed double-precision computation of reciprocal, division, square root and inverse square root. *IEEE Transactions on Computers*, 51(12):1377–1388, December 2002.

[114] J.-A. Piñeiro, S. F. Oberman, J.-M. Muller, and J. D. Bruguera. High-speed function approximation using a minimax quadratic interpolator. *IEEE Tranactions on Computers*, 54(3):304–318, March 2005.

[115] V. K. Prasanna Kumar and Y.-C. Tsai. On synthesizing optimal family of linear systolic arrays for matrix multiplication. *IEEE Transactions*

on Computers, 40(6):770–774, June 1991.

[116] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[117] E. Quinnell, E. Swartzlander, and C. Lemonds. Floating-point fused multiply-add architectures. In *Conference Record of the Forty-First Asilomar Conference on Signals, Systems and Computers, 2007. ACSSC 2007.*, pages 331–337, 2007.

[118] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. van de Geijn. Solving dense linear algebra problems on platforms with multiple hardware accelerators. In *ACM SIGPLAN 2009 symposium on Principles and practices of parallel programming (PPoPP'08)*, 2009.

[119] S. K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming simd extensions on the pentium iii processor. *IEEE Micro*, 20:47–57, July 2000.

[120] D. A. Reed, R. Bajcsy, M. A. Fernandez, J.-M. Griffiths, R. D. Mott, J. Dongarra, C. R. Johnson, A. S. Inouye, W. Miner, M. K. Matzke, and T. L. Ponick. Computational science: Ensuring America's competitiveness. Technical report, President's Information Technology Advisory Committee, Arlington, VA, June 2005.

[121] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi, and J. Owens. Register organization for media processing. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture, 2000. HPCA-6.*, pages 375 – 386, 2000.

[122] V. Saxena, P. Agrawal, Y. Sabharwal, V. K. Garg, V. A. Kuruvilla, and J. A. Gunnels. Optimization of BLAS on the Cell processor. In *Proceedings of the 15th international Conference on High performance computing*, HiPC'08, pages 18–29, Berlin, Heidelberg, 2008. Springer-Verlag.

[123] N. R. Scott. *Computer Number Systems and Arithmetic.* Prentice Hall, 1985.

[124] R. Scrofano, S. Choi, and V. Prasanna. Energy efficiency of FPGAs and programmable processors for matrix multiplication. In *Proceedings of the First IEEE International Conference on Field Programmable Technology (FPT)*, pages 422–425, 2002.

[125] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[126] SEMATECH Inc. International technology roadmap for semiconductors (ITRS), 2008 update. `http://www.itrs.net/`, 2008.

[127] P. Soderquist and M. Leeser. Area and performance tradeoffs in floating-point divide and square-root implementations. *ACM Computer Survey*, 28(3), 1996.

[128] P. Soderquist and M. Leeser. Division and square root: choosing the right implementation. *IEEE Micro*, 1997.

[129] J. Stine and M. Schulte. A combined two's complement and floating-point comparator. In *Proceedings of the 2005 IEEE International Symposium on Circuits and Systems, ISCAS 2005.*, pages 89–92 Vol. 1, 2005.

[130] Y.-G. Tai, K. Psarris, and C.-T. D. Lo. Synthesizing tiled matrix decomposition on FPGAs. In *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications*, FPL '11, pages 464–469, Washington, DC, USA, 2011. IEEE Computer Society.

[131] O. Takahashi, C. Adams, D. Ault, E. Behnen, O. Chiang, S. Cottier, P. Coulman, J. Culp, G. Gervais, M. S. Gray, Y. Itaka, C. J. Johnson, F. Kono, L. Maurice, K. McCullen, L. Nguyen, Y. Nishino, H. Noro, J. Pille, M. Riley, M. Shen, C. Takano, S. Tokito, T. Wagner, and H. Yoshihara. Migration of Cell broadband engine from 65nm soi to 45nm soi. In *Proceedings of the IEEE International Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers.*, pages 86 –597, 2008.

[132] D. Tan, C. Lemonds, and M. Schulte. Low-power multiple-precision iterative floating-point multiplier with SIMD support. *IEEE Transactions on Computers*, 58(2), 2009.

[133] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, 2011.

[134] R. Urquhart and D. Wood. Systolic matrix and vector multiplication methods for signal processing. *IEEE Communications, Radar and Signal Processing,*, 131(6):623 – 631, 1984.

[135] R. A. van de Geijn and E. S. Quintana-Ortí. *The Science of Programming Matrix Computations.* `www.lulu.com/contents/contents/1911788/`, 2008.

[136] R. A. van de Geijn and F. G. Van Zee. High-performance up-and-downdating via Householder-like transformations. *ACM Transactions on Mathematical Software.*

[137] R. A. van de Geijn and J. Watts. SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience*, 9(4), 1997.

[138] F. Van Zee. `libflame`*: The Complete Reference.* `www.lulu.com`, 2009.

[139] F. G. Van Zee, E. Chan, R. A. van de Geijn, E. S. Quintana-Ortí, and G. Quintana-Ortí. The libflame library for dense matrix computations. *IEEE Computation in Science & Engineering*, 11(6):56–62, 2009.

[140] F. G. Van Zee and R. A. van de Geijn. FLAME Working Note #66. BLIS: A framework for generating BLAS-like libraries. Technical Report TR-12-30, The University of Texas at Austin, Department of Computer Sciences, November 2012.

[141] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w TeraFLOPS processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1), 2008.

[142] S. R. Vangal, Y. V. Hoskote, N. Y. Borkar, and A. Alvandpour. A 6.2-GFlops floating-point multiply-accumulator with conditional normalization. *IEEE Journal of Solid-State Circuits*, 41(10), 2006.

[143] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[144] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. Rubio, F. Rawson, and J. Carter. Architecting for power management: The IBM POWER7 approach. In *Proceedings of the 2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*, 2010.

[145] G. Welch and G. Bishop. An introduction to the Kalman filter. Technical Report TR 95-041, Chapel Hill, NC, USA, 1995.

[146] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, Supercomputing '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[147] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the Cell processor for scientific computing. In *Proceedings of the 3rd Conference on Computing frontiers*, CF '06, pages 9–20, New York, NY, USA, 2006. ACM.

[148] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. Scientific computing kernels on the Cell processor. *International Journal of Parallel Programming*, 35:263–298, June 2007.

[149] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 235–246, 2010.

[150] D. Wu, X. Zou, K. Dai, J. Rao, P. Chen, and Z. Zheng. Implementation and evaluation of parallel FFT on engineering and scientific computation accelerator (esca) architecture. *Journal of Zhejiang University-Science C*, 12(12), 2011.

[151] G. Wu, Y. Dou, J. Sun, and G. D. Peterson. A high performance and memory efficient lu decomposer on FPGAs. *IEEE Transactions on Computers*, 61(3):366–378, March 2012.

[152] Y. Yamamoto, T. Fukaya, T. Uneyama, M. Takata, K. Kimura, M. Iwasaki, and Y. Nakamura. Accelerating the singular value decomposition of rectangular matrices with the CSX600 and the integrable svd. In *Proceedings of the 9th international Conference on Parallel Computing Technologies*, PaCT'07, pages 340–345, Berlin, Heidelberg, 2007. Springer-Verlag.

[153] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A fully integrated multi-CPU, GPU and memory controller 32nm processor. In *Proceedings of the 2011 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2011.

[154] N. Zhang and R. W. Broderson. The cost of flexibility in systems on a chip design for signal processing applications. Technical report, University of California, Berkeley, 2002.

[155] L. Zhuo and V. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(4), 2007.

[156] P. Zicari, P. Corsonello, S. Perri, and G. Cocorullo. A matrix product accelerator for field programmable systems on chip. *Microprocessors and Microsystems 32*, 2008.