LACore: A Large-Format Vector Accelerator for Linear Algebra Applications

Samuel P. Steffl

Submitted on: May 1st, 2017

Submitted in partial fulfillment of the requirements of the degree of Bachelor of Science *with Honors* in Computer Engineering

School of Engineering, Brown University

Prepared under the direction of

Prof. Sherief Reda, Advisor

Prof. R. Iris Bahar, Reader

By signing below, I attest that the undergraduate thesis listed above meets the criteria for Honors, and has been successfully presented to the faculty at the Undergraduate Research Symposium.

_____

Advisor's Signature

_____

Reader's Signature

BROWN

_____

Honors Chair's Signature

ABSTRACT

Linear algebra operations are at the heart of scientific computing solvers, machine learning and artificial intelligence. To achieve high performance, linear algebra operations are typically accelerated with vector processing units in central processing unit (CPU) and graphics processing unit (GPU) cores or custom hardware solutions. GPU accelerators are highly successful due to the massive amounts of parallelism and memory bandwidth that can be achieved. They can reach well into the TFLOP/s for single-precision performance by scheduling thousands of simultaneous operations across many lightweight cores. The shortcomings with the GPU approach is that GPUs suffer from coarse-grained synchronization which can be a bottleneck in vector reduction, have poor single-threaded performance compared to CPUs, and use a separate device memory that requires separate memory transfers before and after kernel execution. The other approach, as mentioned, is the custom hardware solution, or ASIC/FPGA accelerators. These designs typically require either using a High-Level Synthesis tool to convert MATLAB or C code into an equivalent HDL, such as Verilog, or require the user to handwrite the HDL. This custom accelerator approach typically achieves very high performance for very specific applications, but does not generalize well to other computational problems, and has a high development cost.

In this thesis, *LACore*, a novel, programmable accelerator architecture for general-purpose linear algebra applications, is presented. *LACore* is a Large-Format vector architecture that overcomes many of these shortcomings of existing HPC hardware through several architectural features including heterogeneous data-streaming *LAMemUnits*; a mixed-precision systolic datapath that supports scalar, vector and multi-stream output modes; and the decoupling and overlapping of memory-accesses and data-execution. To evaluate the *LACore*, the architecture was implemented as an extension to the RISC-V ISA in the gem5 cycle-accurate simulator. In addition, the *LACore* ISA was implemented in gcc, and a C-programming software framework, the *LACoreAPI*, has been developed for high-level programming of the *LACore*. Using a modified version of the HPCC benchmark suite, the *LACore* architecture is compared against three other platforms: an in-order RISC-V CPU, a superscalar x86 CPU with SSE2 enabled, and a scaled NVIDIA Fermi GPU. The *LACore* outperforms the superscalar x86 processor in the HPCC benchmark suite by an average of 3.43x, outperforms the scaled Fermi GPU by an average of 12.04x, and outperforms the RISC-V CPU by an average of 10.72x.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Modern hardware solutions to linear algebra applications include manycore processors such as GPUs, vector extensions to scalar CPUs such as Intel's Streaming SIMD Extensions (SSE), and custom application-specific integrated circuits (ASIC) or Field-programmable gate arrays (FPGA). There are shortcomings with each of these platforms that the *LACore* addresses through its novel architecture.

GPU Issues

One issue with GPU usage for HPC applications is that GPU architectures inherently lack the ability to reduce data without thread synchronization. This can act as a bottleneck for common kernels such as DGEMM, which is a sequence of many dot-products. An architecture that natively supports arbitrarily large vector reduction without thread synchronization would have a performance advantage.

Additionally, GPUs are built for parallel processing only, and require a high-performance single-threaded CPU to handle the sequential portions of applications. Having a single, powerful thread that can switch between parallel and sequential mode can overcome the inefficiencies in the GPU fork-join model. Although x86 with SSE extensions may appear to support this model, it does not fully support arbitrary vector sizes and is not as parallelizable as a GPU.

Finally, GPUs use separate device memory, which requires transfer of data between the host and device before and after kernel execution. Although the latency can be partially hidden by overlapping kernel execution with data transfer, an architecture that does not require any data movement between sequential and parallel execution modes would be advantageous.

Application-Specific Accelerator Issues

Although custom ASIC/FPGA solutions typically provide the highest performance for a particular application, they are rigid, can have too narrow of scope, and have a high development cost. A better architecture would offer the same custom hardware benefits as ASICs while still being programmable for solutions to a wide range of linear algebra applications.

The LACore Solution

The *LACore* architecture aims to provide solutions of the above shortcomings through several features:

- **Heterogeneous** data-streaming *LAMemUnits* capable of accessing scalar, vector, and sparse matrix objects in both scratchpad and memory.
- **Large-Format Vector** support, providing an interface to work with arbitrarily-large vectors and matrices.
- **Systolic Vector-Reduction** datapath within the *LAExecUnit*, implementing up to 24 different functions on 3 mixed-precision input data-streams.
- **Decoupled** architecture that overlap data-execution with memory-access during complex memory-memory instructions.
- **Multi-Stream**, Vector, and Scalar output modes, with Multi-Stream output mode reducing multiple sub-vectors within larger input vector streams for enhanced data-execution and memory-access overlap.

The rest of this introduction will give a brief high-level overview of the proposed *LACore* architecture, the software framework that has been developed for it, and the implementation and benchmarking methodology and results. Chapter 2 will discuss the various sources of motivation for designing a new computer architecture as well as reasoning for some important design decisions. Chapter 3 will look at related work. Chapter 4 will discuss the *LACore* architecture in depth, and Chapter 5 will discuss the *LACore's* instruction set and usage model. Chapter 6 will discuss the software toolchain for developing applications for the *LACore* platform and the *LACoreAPI*, a C-programming software framework for the *LACore*. Chapter 7 will discuss the benchmark evaluation of the *LACore* vs the RISC-V, x86 and Fermi GPU platforms. Chapter 8 will present and analysis of the Roofline Model for the *LACore*, and Chapter 9 will discuss area footprint of the *LACore* compared to other platforms. Concluding remarks on the merits and deficiencies of the *LACore* will be in chapter 10. All software developed for this project is freely available as a git-repository under the Brown University Scale-Lab account: https://github.com/scale-lab/la-core.

1.1 THE LACORE ARCHITECTURE

1.1.1 Overview

This thesis proposes a new processor architecture called *LACore*, which stands for "Linear Algebra Core", to alleviate the shortcomings of both the manycore chips as well as the shortcomings of the custom hardware solutions. From a high level, the *LACore* is a complex, vector-like data-processing unit embedded inside a normal scalar processor, acting like an on-chip custom accelerator, while being

integrated directly into the CPU's processing pipeline. The *LACore* extends the scalar CPU's instruction set with its own ISA extension, so that a single stream of scalar and *LACore* instructions can be efficiently interleaved and executed with low issuing latency and a unified address space. This thesis chooses the RISC-V processor as the scalar processor to embed the *LACore* into (Waterman A. a., 2016). This is done for numerous reasons: RISC-V is a modern architecture addressing most of the problems faced by previous RISC architectures; RISC-V is rapidly growing and has ubiquitous, open-source, actively developed tool-chains; and finally, one of RISC-V's main goals is to proliferate accelerator-rich architectures by being an easily-extendable ISA, and providing a full-stack toolchain for developing and verifying hardware accelerators (Asanovic K. A., 2016). Using RISC-V as the base ISA will therefore ensure that the CPU the *LACore* is built into will contribute to the *LACore's* overall future usefulness instead of detract from it.

The *LACore* architecture adds a few physical components to the scalar core. The first is a multi-port private 64 kB scratchpad for fast access to temporary results. The scratchpad is a design paradigm found commonly in embedded designs with real-time requirements (Banakar, 2002) and GPU platforms such as NVIDIA's lineup (Lindholm, 2008). The *LACore* also adds a set of configuration registers for determining the type of input or output (scalar, vector, matrix or sparse matrix), the vector layout in memory (address, stride, skip, count), and where the input or output should reside (scratchpad, memory or registers). These registers, called *LACfgs*, are used to configure the *LACore* execution unit. In addition to the scratchpad, the *LACore* uses its own private L1 Cache, called the *LACache* to access the L2Cache instead of using the scalar core's L1 Data Cache. Finally, the most important physical piece of the *LACore* is the novel complex execution unit, which performs all the memory interactions and vector-like data processing.

1.1.2 Novel Contributions

There are several features in the *LACore* architecture that set it apart from existing architectures. The first novel feature is the abstracting of heterogeneous inputs and outputs of the *LACore* execution unit as generic streams of data. These streams are not to be confused with CUDA Streams, which are streams of kernels, not data operated on by kernels (Rennich, 2011). The *LACore* data streams are more closely associated with dataflow processing, where a large vector of similar data is streamed into a fixed, pipelined datapath. Dataflow processing originated with supercomputing in the 1980's (Dennis, 1980) and is still a very common design paradigm in HPC hardware. As mentioned, the

LACore data-streams operate on heterogeneous data types, which means the structure of the input (scalar, vector, matrix or sparse matrix) and their various layout configuration in memory (i.e. the stride, count, skip of a vector config) are handled directly in hardware by something called a *LAMemUnit*. The *LAMemUnit* provides the data stream interface to the *LACore* execution unit while handling the complex interactions with the scratchpad and *LACache* modules. The reason this stream-abstraction is useful, is that it allows a flexible, hardware-supported programming interface for using heterogeneous data types and data structures at the same time without having to do any conversions or memory movements. For example, performing DGEMV (a BLAS-2 routine) on a sparse matrix with a vector and storing the result as a dense matrix can be done in three configuration instruction and one execution instruction, while the complex index mapping is performed directly in hardware.

Another novel architectural feature of the *LACore* is the complex *LAExecUnit* operating on three arbitrary-precision input vector streams and producing an arbitrary-precision output stream. The *LACore's* execution unit can perform a wide range of arithmetic functions on the input data, and can produce either a vector output, a scalar output, or something called a "multi-stream" output, which is the most powerful operation, and used as the inner kernel in routines such as DGEMM. Additionally, this execution unit can take in a mix of single-precision and double-precision IEEE floating point input streams and produce a single-precision or double-precision output stream. The key point is that this is not just a dual-precision execution unit – it can operate on mixed precision vectors simultaneously, where the inputs are cast to the output stream type on the fly. This interface is another useful tool to working with heterogeneous datatypes at the same time.

1.2 THE LACOREAPI SOFTWARE FRAMEWORK

Modern HPC applications require robust and performant software solutions to take advantage of the underlying hardware effectively. The *LACore* is no different: for the hardware to be productively used by HPC application developers, a gcc/binutils toolchain has been created with full *LACore* ISA support. Additionally, a C-programming framework, called the *LACoreAPI*, was created for increasing developer productivity and improving application performance. The toolchain and framework have been used extensively in the development and testing of the *LACore* architecture in multiple simulators.

The *LACoreAPI* exposes three different classes of instructions to the programmer: *ExecInsts*, *XferInsts*, and *CfgInsts* which respectively do complex vector-like operations on streams of data, transfer and transform data from one location to another, and configure an *LACfg*

configuration register. Essentially, operations can be broken into execution, data-movement and configuration, which provides a very complete API for the user to do a wide range of operations on arbitrary data-sources in memory or the scratchpad.

1.3 LACORE EVALUATION AND RESULTS

The *LACore* architecture proposed and elaborated in the paper was successfully implemented 4 times in 2 different simulators: the RISC-V Spike ISA Simulator and gem5. Spike ISA Simulator is dubbed as the "RISC-V golden model", and is used as a functional simulator with limited timing accuracy (Andrew Waterman, 2011). The *LACore* extension was added to the simulator to verify functional correctness of implementing the entire *LACore* ISA. The other simulator, gem5, is an industry-standard cycle-accurate simulator used by numerous semiconductor companies in industry and research groups in academia (Binkert, 2011). The main advantage of the gem5 simulator is its ability to accurately model an entire computing system, including the memory controller, cache hierarchy, and multi-threaded or multi-core CPU designs. Having the timing-accurate memory interaction is necessary for evaluating the *LACore* since one of its main features is a complex memory fetch-and-store unit. Within gem5, the *LACore* was implemented as a functional CPU model, a cycle-accurate non-pipelined CPU, and a cycle-accurate in-order CPU model, and was verified for correctness in all three cases.

In addition to the architectural simulators, a comprehensive set of C unit-tests were developed to exercise the functional correctness of any *LACore* implementation. Specifically, they test all the data-execution and data-movement operations for all variations of input and output configurations. These test-suites act as regression tests when changes are made to the *LACore* ISA or an existing *LACore* CPU model, and allow future *LACore* researchers and developers to have a reference point for a functional design.

Finally, a set of linear algebra benchmarks were developed targeting the *LACore* platform, most of them within the HPCC benchmark suite (Luszczek, 2005). These tests were run on the cycle-accurate gem5 *LACore* CPU model and compared to finely-tuned implementations of other platforms and linear algebra frameworks. All tests on the *LACore* were verified for correctness against the GNU Scientific Library (GSL), which provides an API for highly-performant linear algebra operations (Gough, 2009). Additionally, running both the *LACore* and GSL on the same gem5 cycle-accurate system allows us to adjust the simulator benchmark results of the *LACore* to be more accurate, since we will have GSL benchmark results on both the simulator and a real-world CPU.

2 MOTIVATION

2.1 HARDWARE ACCELERATION FOR COMPUTATIONAL PROBLEMS

The fundamental problem the *LACore* is trying to address is solving linear algebra related applications as efficiently and quickly as possible. The applications the *LACore* directly targets cover the broad range of both dense and sparse linear algebra, which are also the foundations of many other classes of applications as well, such as machine learning and artificial intelligence. Linear algebra applications are two of the original seven dwarves described in the influential *The Landscape of Parallel Computing Research* from Berkeley (Asanovic K. B., 2006), with the other five original dwarves being FFT, N-Body methods, structured grids, unstructured grids and Monte Carlo calculations.

The seven dwarves represent what the authors saw as the major types of computational problems in HPC that people generally care about, and the *LACore* provides a suitable platform to accelerate at least five of them. Although FFT is not directly targeted by the *LACore*'s ISA, a specialized FFT kernel was written for the *LACore* and outperforms many other architectures, as discussed in the Benchmarks and Evaluation section. In addition to the FFT carryover on the *LACore*, structured and unstructured grids have high overlap with dense and sparse linear algebra applications, so these two applications can also be accelerated by the *LACore*. Therefore, in addition to being a general-purpose linear algebra accelerator, the *LACore* is an effective accelerator for most other HPC applications that people generally care about.

2.2 WHY ANOTHER ISA AND MICROARCHITECTURE?

With the primary motivation for studying the *LACore* now addressed, the next question is why develop another radically different architecture, instead of using already existing architectures such as NVIDIA's Pascal GPU or Intel's Xeon Phi architecture? Aside from addressing the various shortcomings with GPUs discussed in the Introduction section, the *LACore* provides a platform that is specialized for a particular domain of applications, unlike GPUs. There is great evidence in literature of the demand for these specialized architectures for large-scale computing, such as the Anton II supercomputer (Shaw, 2014). Anton II is a custom hardware and software solution targeting molecular dynamics, and has shown orders-of-magnitude performance improvements over the best software solution running on

commodity hardware. For simulations with hundreds of thousands of molecules, this can cut runtimes from weeks down to minutes or seconds, which allows for faster and more productive research.

Another example of a processor architecture for a particular domain of applications is Google's Tensor Processing Unit (TPU), which is used to accelerate the inference phase of neural networks (Jouppi, 2017). Their TPU provided an order-of-magnitude improvement in both performance and power consumption compared to GPUs and CPUs. Additionally, the TPU architecture provided Google's users a consistently low and reliable latency that was unachievable with GPUs or CPUs. Since Google spends a significant amount of money to power datacenters, and Google also services billions of users, the TPUs multiple advantages outweighed its high cost of development.

Anton II and the TPU are just two of many case studies showing that there can be a large incentive and large amounts of money invested in even getting an order of magnitude performance increase from custom hardware solutions for important problems classes, which is why developing the *LACore* architecture as a custom solution to linear algebra applications is justified and desirable.

2.3 The Power and Utilization Walls

2.3.1 The Power Wall

Gordon Moore stated what has famously been coined "Moore's Law" in a 1965 article, where he states "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year" (Moore, 1998), which has been interpreted in modern culture as saying transistor density will double roughly every year. A more accurate interpretation of Moore's quote would be that its most cost effective from a chip manufacturer's perspective to double the device count every year, which just correlates company profitability to transistor count per chip over time.

Moore's law was accurate for many decades. In the early years, the main contributors to increasing the device count per integrated circuit were increasing the yield and feature miniaturization (Moore, 1998). Feature miniaturization, also known as Dennard scaling, is what most people associated with Moore's law (and some still do), but has recently slowed as we approach the 7nm and 5nm technology nodes. However, this decline in the rate of scaling has been compensated by numerous other advances, such as novel architectures, higher yield rates, and new device types, such as the FinFET transistor (Hisamoto, 2000). The takeaway is that Moore's Law encompasses more than just device

scaling, which is why processors are still delivering more and more each year in terms of performance and capabilities.

An important side-effect of the extremely small feature sizes used in modern semiconductors, is that the silicon can get very hot if too much of it is powered at once, so only a fraction of the silicon can be usable at any given time. Although the transistor size shrinks and more can fit on a given die area, a smaller percentage of the transistors can be powered at any given time. This unusable silicon area is called “Dark Silicon” (Taylor, 2012), and the amount of it we cannot utilize is determined by the “utilization wall”.

2.3.2 Utilization Wall and Custom Accelerator IP Cores

Because only a fraction of the silicon can be used at any given time due to the Utilization wall, people have found alternative ways to make use of that otherwise-useless silicon. One approach is to put application-specific accelerators in these dark-silicon areas, and only power them on when they need to run. This has the benefit of higher application-specific performance while also being overall-more energy efficient. The paradigm of putting application-specific accelerators in the same silicon as the scalar CPU core is called “Accelerator-rich architectures” (Esmaeilzadeh, 2011).

The *LACore* architecture can be thought of as one of these embedded accelerators within the CPU’s silicon die. The *LAExecUnit*’s datapath can be powered on and off depending on if it is being used, and it provides custom hardware for general purpose linear algebra applications. This makes the *LACore* a smaller, more energy-efficient, and higher-performance solution to linear algebra applications than general purpose GPU accelerators.

2.4 Free and Open Hardware

A final motivation for developing the *LACore* architecture is to provide a fully free and open source parallel-computing platform for linear algebra applications. Software is much easier to open-source due to its lower capital cost of development than hardware (i.e.: no fabrication is required), and therefore the open-source software community is thriving compared to the open-source hardware community. Recent attempts to lower the barrier to open-source hardware include the RISC-V ISA, which is trying to modernize and open up hardware (Asanović, 2014) (Waterman A. , 2016). The RISC-V platform is a scalar CPU though, so parallel-computation is still relatively locked down by proprietary platforms. The *LACore* will fill this void in the open-source parallel-computing hardware space.

3 RELATED WORK

The *LACore* architecture possesses features similar to other existing linear algebra accelerating architectures. Some of these features show up in several vector-processor architectures such as Cray-1, Hwacha, and SX-ACE or multimedia-application processors such as RSVP and MOM. The *LACore* even possesses some features similar to more exotic architectures such as the Cell processor. In addition to related processor designs, this thesis presents a methodology for designing and evaluating accelerator architectures that is related to the methodology presented in gem5-aladdin.

CRAY-1 Supercomputer

The CRAY-1 Supercomputer was one of the original vector-processors (Russell, 1978) (Cray, 2003). It aimed to provide both a high-performance sequential processing mode as well as a high-performance parallel-processing mode. The vector-processor was a memory-memory architecture, and it targeted general HPC and supercomputing applications.

There are a handful of similarities between the CRAY-1 vector processor architecture and the *LACore*. First, they both have a type of decoupled data-streaming architecture, since the CRAY-1 streams data from vector registers into functional units and can "chain" functional units together, meaning the outputs of one functional unit flow directly into the inputs of another functional unit. Second, both architectures provide a mechanism for intermediate result storage, where the *LACore* uses a scratchpad and the CRAY-1 uses temporary vector registers. Third, both architectures allow working with arbitrarily-large sized vectors using a gather-scatter interface.

There are major differences, however. First, the CRAY-1 architecture uses traditional vector registers for functional unit inputs and outputs, while the *LACore* uses FIFOs in a more stream-processing-like paradigm. Second, CRAY-1 does not appear to support the novel multi-stream output mode that the *LACore* provides. Finally, CRAY-1 does not support heterogeneous, mixed-precision data-sources like the *LACore*.

The Hwacha Vector Processor

The Hwacha Vector processor is RISC-V-based vector accelerator unit (Lee Y. O., 2015). The Hwacha architecture is an archetypal vector processor, similar to the Cray-1 Supercomputer (Cray, 2003), and it appears to primarily target linear algebra-related applications, similar to the *LACore*.

There are a handful of similarities between the *LACore* and Hwacha. First, they both use RISC-V as the primary vehicle for their implementation. Second, both accelerators are integrated into the scalar CPU's die, so no memory transfers are needed before and after kernels, like with GPUs. Third, they both allow working on arbitrarily-large vectors in memory, and can be described as memory-memory architectures. This leads to the fourth similarity of the *LACore* and Hwacha both having a decoupled access/execute architecture to allow simultaneous memory accesses and data execution.

There are major architectural differences though. First, Hwacha is a traditional vector architecture, and is composed of vector lanes and vector registers to hold operands, while the *LACore* has a novel complex datapath and data-streaming FIFOs in place of these two. Second, Hwacha does not synchronously execute its instruction stream inside the host scalar CPU's instruction stream, and instead dispatches commands from the scalar CPU to the Hwacha accelerator over the ROCC interface. This means that the Hwacha accelerator needs to implement its own redundant scalar CPU in order to execute its instruction stream. The *LACore* accelerator, on the other hand, uses the same instruction stream for both the scalar instructions and the *LACore* instructions, which simplifies the programming model and synchronization. Third, Hwacha does not have a scratchpad for intermediate results like the *LACore*. Fourth, Hwacha does not provide a mechanism similar to the *LACore*'s multi-stream output mode. Finally, Hwacha does not allow working with heterogeneous data sources or sparse matrix configurations, like the *LACore*.

NEC SX-ACE Supercomputer

The SX-ACE is a Vector-processing supercomputer (Momose, 2014). Its primary goal is to improve performance in the broadening range of HPC applications, and its architecture specially addresses the increasing Bytes/Flop ratio in these newer applications, where memory bandwidth is increasingly more important than peak computation throughput.

There are a handful of similarities between SX-ACE and the *LACore*. First, both architectures support arbitrarily-large vectors using a gather-scatter interface. Second, both architectures provide hardware support for fast temporary storage for intermediate results. SX-ACE uses what it calls "Assignable Data Buffers" (ADBs), which are high-bandwidth cache-like memories, while the *LACore* uses a scratchpad. Third, both architectures support single-precision and double-precision arithmetic. Finally, both have a type of data-flow architecture, since the SX-ACE allows chaining functional units

within the vector-processing unit, where the outputs of one functional unit flow into the inputs of another functional unit.

There are major differences, however. First, SX-ACE assumes a traditional vector architecture with vector-registers, and 16 vector execution pipelines, similar to CRAY-1 and Hwacha, but the *LACore* uses FIFOs for input and output data-streams due to its stream-processing architecture. Second, SX-ACE connects the memory controllers directly to multiple DIMMs, while the *LACore* uses a cache-based memory hierarchy. This means the SX-ACE is designed primarily for high-throughput, high-latency memory operations, but the *LACore* is designed for high-throughput and potentially *low-latency* operations when the workloads are smaller. Third, SX-ACE has separate instructions for loading data into and out of vector registers, which means it does not efficiently overlap memory access with data execution for the same operation, like the decoupled access/execute architecture of the *LACore*.

RSVP (The Reconfigurable Streaming Vector Processor)

RSVP is a vector-like processor targeting multimedia applications (Ciricescu, 2003). Its main architectural feature is that it is a decoupled stream-processor. The RSVP authors define the stream-processing paradigm as a high spatial-locality and low temporal-locality paradigm where streams of elements are identically processed and never used again. An example of this type of computation would be the STREAM-Triad application in the HPCC benchmark suite (Luszczek, 2005). There are two main problems with other architectures that are used for these streaming applications. First, traditional SIMD fixed-size vector extensions, such as Intel's SSE, are difficult to program since they have a low programming abstraction, and they offer lower speedups. Second, traditional Vector architectures use memory-memory operations, but run into memory bandwidth limitations easily. The RSVP addresses both of these issues with its streaming architecture and high level programming model.

There are a handful of similarities between the *LACore* and RSVP. The first is that they both exemplify a decoupled access/execute architecture, where memory access and data execution occur simultaneously in different sub-processing units, and both sub-processing units are connected by FIFOs. Second, RSVP provides a gather-scatter vector configuration that is similar to the *address, stride, count, skip* vector configuration used by the *LACore*. Both architectures offer a higher-level programming model that efficiently utilizes the hardware, and both have a programmable or reconfigurable datapath. Finally, RSVP instructions run synchronously with host processor, giving the illusion of a single instruction stream, which is the same as the *LACore*'s architecture.

There are major architectural differences between the *LACore* and RSVP, however. First, RSVP primarily targets multimedia applications and is therefore tuned for smaller data sources and simpler kernels. Second, RSVP's execution unit is programmed using dataflow graphs, instead of traditional programming languages. This limits the size of the kernel that can be performed on the data-streams, unlike the *LACore*. Third, with RSVP, data is meant to be used once (due to the streaming paradigm), and there is no optimization for intermediate results, like the scratchpad in the *LACore*, so more complex operations are less performant.

MOM (Matrix-Oriented Multimedia-extension)

The MOM ISA is a matrix-oriented ISA used in multimedia applications (Corbal, 1999). Like the *LACore*, it is a parallel computing extension to a simple RISC ISA, in this case the Alpha processor. The MOM architecture targets small multi-media kernels, which have a large overlap with general linear algebra computing, and many of the examples in the MOM literature use DGEMM as an example subroutine. This means MOM and *LACore* both target the same general application domain, but *LACore* is generalized and provides a larger feature set than the MOM extension.

There are two main similarities between the *LACore* and MOM. The first is that MOM supports instructions that work on small matrices operands. These matrices can be up to 16x8 in dimensions, and are stored in vector-like registers. The ability to work on matrices instead of just vectors provides the same functionality as the *LACore*'s multi-stream output mode, where large input data-streams can be partitioned into sub-streams and each sub-stream can be used in a separate dot-product simultaneously. The other similarity between MOM and *LACore* is that they both support general strided vector memory accesses, instead of just sequential vector memory accesses. So both support some version of a gather-scatter interface.

There are a few key differences between the architectures though. First, the MOM authors clarify that traditional vector processors are more performant that multimedia extensions for arbitrarily large vector sizes. This means that the *LACore* inherently scales better with problem size. Second, MOM is limited to 16x8 matrices as operands, and they must be stored in vector-like registers, while the *LACore* streams arbitrarily large matrices in and out of FIFOs. Third, MOM does not support Large-Format vectors, heterogeneous data-types or sparse-matrix data-sources, like the *LACore*. Finally, the MOM architecture provides separate instructions for memory accesses and data execution, since it is a load-store architecture, and does not provide a decoupled access/execute interface like the *LACore*.

IBM Cell Processor

The Cell processor, from Sony, Toshiba and IBM, is an architecture radically different from traditional vector processors or scalar processors with multimedia extensions (Williams S. S., 2006). It has a powerful single-threaded PowerPC core, and 8 smaller parallel-processing cores, and is used in a range of applications including linear algebra kernels.

Due to the Cell's unique architecture, it does not share very many architectural features with the *LACore*. One major feature it does share, though, is programmer-controlled memory management. The Cell processor takes this idea to much more extreme levels than the *LACore*, though, since the user explicitly manages every single memory operation. The *LACore* only provides an API for manually managing the scratchpad-memory and scratchpad-scratchpad memory transfers. This direct memory management is a feature that allows the application to perform more deterministically, with the tradeoff of being slightly more cumbersome to program.

There are additional differences between the Cell processor and *LACore*, however. First, the Cell is composed of multiple independent processing units, while the *LACore* is a single processor with sequential and parallel functional units all included. Second, the Cell does not provide a decoupled access/execute architecture like the *LACore*. Third, the Cell does not allow working with heterogeneous, mixed-precision data-sources, like the *LACore*. Finally, the Cell does not appear to support any functionality analogous to the *LACore*'s multi-stream output mode.

Gem5 Aladdin

gem5-Aladdin is not a computer architecture, but a simulation framework for accelerator-rich processors (Shao, 2016). This work developed an extension in gem5 for design-space exploration of fixed-function accelerators that use scratchpad memories. The simulation methodology was similar to that of the *LACore*, since both works evaluated accelerator architectures by extending gem5. The major differences were that the gem5-Aladdin research does not do any functional verification of their algorithms in gem5 since their primary concern was area and power estimation, and that the fixed-function accelerators were not tightly coupled to the scalar CPU, which is a major architectural feature of the *LACore*. Additionally, datapaths for each specific kernel were instantiated using the Aladdin framework, which contrasts with the *LACore* that uses a single general-purpose architecture to address a variety of linear algebra kernels.

4 LACORE ARCHITECTURE

4.1 LACORE AS A SCALAR CPU EXTENSION

In any computer program, there will be a scalar, sequential part of the code and a potentially parallelizable part of the code. In a paper by Gene Amdahl, it was declared that about 40% of instructions in typical applications fall into the *data management housekeeping* category, or the sequential code (Amdahl, 1967). The result is that 60% of the code can be run in a highly parallel fashion, costing up to zero runtime, but the overall runtime will end up being dominated by the scalar 40% of instructions. In the wide array of applications, the scalar portion can vary from 20%-40%, but this fact remains the same.

Amdahl's law makes the point that the scalar CPU performance is very important, and since the `LACore` is mainly designed for parallel execution, a decision had to be made how to integrate the `LACore` into a scalar CPU. One option would be to co-design a scalar architecture tightly coupled with the `LACore`, effectively creating a full HPC vector processor like the CRAY X1 (CRAY, 1977). An opposite approach would be to fully decouple the vector part from the scalar CPU and have them communicate over the PCIe bus, similar to discrete GPU accelerators. An intermediate approach would be to decouple the scalar part from the vector part on the same piece of silicon, which is exemplified by the Hwacha vector processor using the RISC-V ROCC accelerator interface (Lee Y. O., 2015), (Asanovic K. A., 2016). The first approach, following CRAY, would require redesigning a full Out-of-Order (OoO), SMT scalar processor or something with equivalent or better performance. Since this is already being done in industry, and gem5 already supports a highly detailed OoO scalar CPU (Binkert, 2011), time spent reinventing the wheel here would be wasted. The second approach mentioned will lead to a design that suffers from the same shortcomings as GPUs. The third approach gives us the higher performance of putting the scalar and vector parts on the same chip, while also providing a clean line of separation between them. This allows development of the vector implementation without having to worry about the scalar implementation, which is why this third approach is used by the `LACore`.

The major difference between the `LACore` and Hwacha implementations, however, is that from the CPU's perspective, the `LACore` is just a functional unit with variable latency within the CPU's execution stage, while the Hwacha vector processor is a different module sitting next to the CPU and receives instructions over an interface from the CPU. Additionally, it is not clear from the Hwacha

architecture manual, but it appears the Hwacha vector processor can execute in parallel with the scalar CPU, allowing vector and sequential work to be done simultaneously (Lee Y. S., 2015). Assuming some mechanism for this does exist, this would be a slight advantage of the Hwacha processor over the *LACore* processor, since the *LACore* instructions will block the currently executing hardware thread until it is complete. This penalty is easily masked using modern OoO SMT processors, though, so advantage held by Hwacha is most likely negligible.

The *LACore* architecture was therefore chosen to be an extension to the RISC-V scalar processor (Waterman A. a., 2016), similar to Hwacha. The choice of RISC-V could be swapped with MIPS, x86 or ARM with only changes to the ISA and scalar processor's decoder, and no changes to the actual *LACore* implementation. The reason RISC-V was then chosen for this research is that it is a modern, free, and open-source platform, with robust toolchains for developing both hardware and software. Additionally, fully tested RTL models exist for a plethora of different CPU models in RISC-V, including one, three and five-stage single-issue pipelined processors, called the Sodor collection (Celio C. , 2014), as well as a more complex OoO processor, called BOOM (Celio C. P., 2015). So, the *LACore* needs to only be implemented once in gem5 and once in RTL in order to plug it into the many different freely-available scalar CPU models. This cannot be as easily done with other scalar architectures like x86 or ARM.

4.2 LACore Microarchitecture Overview

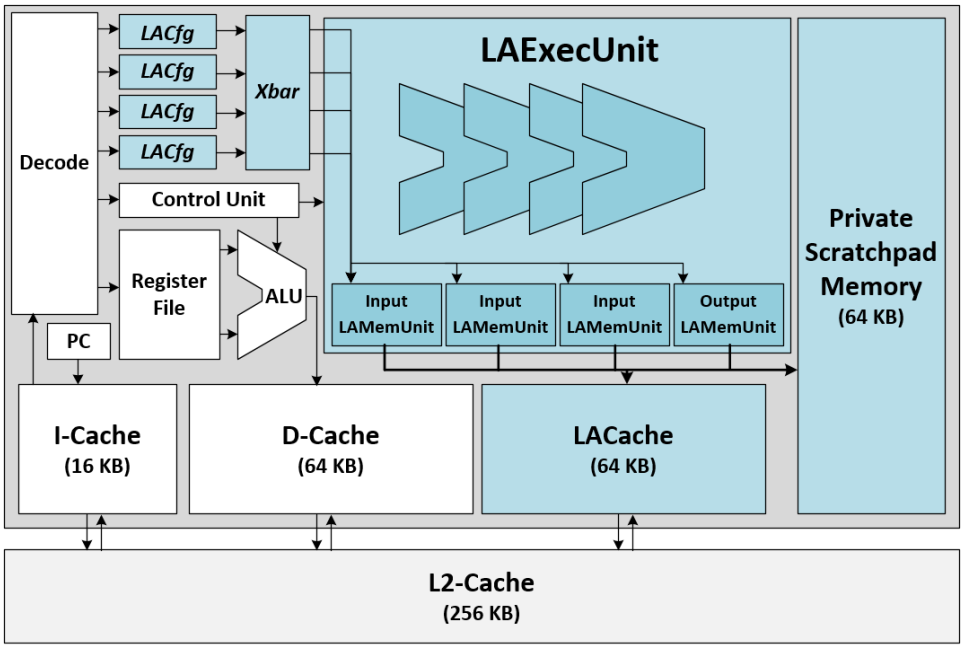

Figure 4-1: LACore Microarchitecture Block Level Design.

Figure 4-1 illustrates the high-level microarchitecture which consists of a scalar CPU and an *LACore* acceleration unit. The gray box represents a single CPU. The white blocks within the CPU are the components from a typical scalar processor: The Instruction Cache, Data Cache, Decoder, Scalar ALU, and Register File. For this paper, a RISC-V processor was used as our scalar CPU implementation. The blue blocks in Figure 4-1 are the part of the *LACore*'s extension to the base scalar CPU: The *LACore* configuration registers (called *LACfgs*), an *LACore* execution unit (called the *LAExecUnit*), a 64 kB per-core private scratchpad memory, and a high-throughput multi-banked cache (called the *LACache*).

The scalar CPU's Instruction Set is extended with *LACore*-specific instructions. The *LACore*'s instructions are decoded by the scalar Decode Unit, providing configuration for the *LACfgs*, *LAExecUnit* and *LAMemUnits*. The *LACore*'s Execution instructions are 4-operand (3 inputs and 1 output) complex memory-memory operations (where memory could be main memory or scratchpad) and will execute for a variable amount of processor-cycles, concurrently accessing data in memory and scratchpad through the *LAMemUnits*, while executing arithmetic operations in the *LAExecUnit*'s datapath. The *LAMemUnits* and the datapath are connected by clock-crossing FIFOs, with the datapath running in a separate, slower domain from the rest of the *LACore*.

The *LACfgs* are large configuration registers that configure how the four *LAMemUnits* access data in memory or scratchpad. The *LAMemUnits* use these configurations to stream data between the *LAExecUnit* and the memory or scratchpad. The scratchpad is a low-latency, high-throughput memory typically used to store temporary intermediate results of multi-instruction operations in the *LAExecUnit*. The *LACache* is a third L1 cache alongside the Data-Cache and the Instruction-Cache, with multiple ports and banks for the *LAMemUnits* to more effectively access the memory system.

4.3 THE LAEXECUNIT

The *LAExecUnit* is a highly-configurable parallel processing unit with three major types of sub-units: the *LAMemUnits*, the datapath, and FIFOs connecting them all, as illustrated in Figure 4-2. There are four *LAMemUnits*, three which read large-format streams of data from the scratchpad, the memory or an *LACfg*, and write them into the datapath's input FIFOs A, B and C. The fourth *LAMemUnit* reads the datapath's output FIFO, D, and writes the resulting large-format data-stream to the scratchpad or memory. This categorizes the *LACore* as a Decoupled Access/Execute Architecture (Smith, 1982), where the memory access and data execution are separate processing units connected by

FIFOs, reminiscent of the design presented in (Lee Y. O., 2015). The datapath is composed of 8 `VecNodes` which support 8 different arithmetic operations, 7 `ReduceNodes` which support 3 different reduction operations, and an `AccumulateNode`. For brevity in Figure 4-2, only 3 out of the 8 `VecNodes` and 3 out of the 7 `ReduceNodes` are shown. So in reality, the `LAExecUnit` is about three times as tall, but still has the same number of `LAMemUnits`: 3 input `LAMemUnits` and 1 output `LAMemUnit`. Also, take note that the `LAExecUnit's` datapath operates on 512-byte SIMD buffers, which hold either 16 single-precision or 8 double-precision floating-point elements.

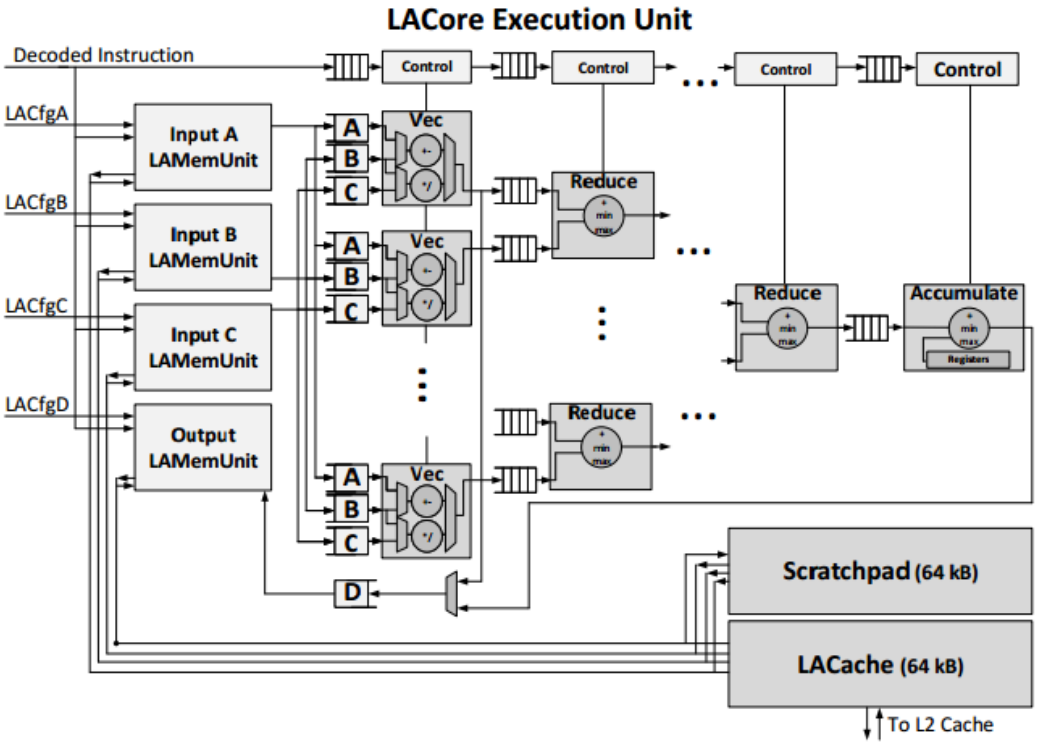

Figure 4-2: LAExecUnit with major sub-blocks shown.

4.3.1 The LAExecUnit Datapath

Novel Stream Processing

The configurable datapath performs operations on three input streams and produces a single output stream. In this sense, the `LAExecUnit's` datapath can be thought of as a traditional stream processing unit or dataflow processing unit. In stream processing, the data flow through fixed execution units connected by queues, in this way, flow control is automatically performed and execution happens at maximum possible speed (Beard, 2013). Dataflow processing as a paradigm contrasts control flow processing (an example is the Von Neumann architecture), where a program counter continually fetching the next instruction from memory and performing branching and jump operations based on

current instruction results. The advantage of using dataflow in the *LAExecUnit's* datapath is that it does not need to operate conditionally based on the input elements or re-implement a whole scalar CPU for instruction fetching, like the Hwacha processor (Lee Y. O., 2015), but instead performs the same operation on all inputs data elements. The minor caveat is that the *LAExecUnit's* datapath needs to be configured at the beginning of the current instruction, but then the configuration remains constant for the remained of the operation.

The three streams of input vectors are labelled A, B and C in Figure 4-2. They arrive at the datapath through FIFO queues. This means the datapath is completely decoupled from the vector stream generation done by the *LAMemUnits*. The output vector stream gets put into the D FIFO queue, which is consumed by another *LAMemUnit*, which handles writing the data back to memory, scratchpad or a register. The datapath is completely unaware of the shape and location of its input and output vectors, it is just concerned with execution. Most vector-processor architectures require the inputs and outputs to go to "vector registers", such as CRAY and Hwacha (CRAY, 1977) (Lee Y. O., 2015), but the *LAExecUnit* instead uses the FIFOs to store the inputs and outputs of the datapath in order to support a decoupled dataflow model. Another noteworthy architecture that uses a very similar streaming computation model is the RSVP architecture (Ciricescu, 2003), which connects the memory system to the execution system through input and output FIFOs. However, the RSVP targets multi-media applications that operate on smaller batches of data, and therefore has a few major architectural differences regarding the streaming, which is discussed in the *LAMemUnit* section.

Since the execution and memory access are fully decoupled, the *LAExecUnit* represents a *Decoupled Access/Execute Architecture* (Smith, 1982). Decoupling the memory access from execution has been shown to provide performance improvements by allowing individual components to operate at their own rate all the time. This is especially useful in the *LAExecUnit* since the latency of all the sub-blocks within it are variable, and allowing some sub-blocks to proceed many cycles ahead of other sub-blocks instead of stalling the whole system is a fluid and fine-grained approach. For example, let's say *LAMemUnit* A produced 9 elements very quickly, and put them in the queues to the datapath, but the next 3 items produced a series of cache misses and stalled input A for 10 cycles. The latency of the cache misses in this case is partially hidden by the fact that beforehand, *LAMemUnit* A could run ahead and place extra elements in the datapath's input queues. If input A and the datapath had to operate synchronously as 1 unit, then the datapath would have to stall for the same amount of time as A, and overall the stalling latency would be larger.

Novel Reduction Unit

Another distinguishing feature of the *LAExecUnit* compared to the datapath of other parallel processors is the vector reduction unit, composed of the tree of *ReduceNodes* seen in Figure 4-2. The CRAY X1 and Hwacha Vector processor do not natively support reduction within a single cycle, so performing a dot product on vectors of length N (with N up to 2^{64}) will take $O(log\ N)$ instructions to complete, while with the *LACore* it would take $O(1)$ instruction to complete. We can see other mainstream parallel processing platforms suffer the same fate as other processors. For example, on NVIDIA's CUDA platform, many lightweight threads can perform the vector-vector multiplication on independent data instantly, but then the programmer must manually reduce the result using a repeating pattern of synchronizations and vector-adds (Sanders, 2010). That means that on GPUs, in order to perform a dot-product on vectors of length N, $O(log\ N)$ thread synchronization calls must happen, which becomes a performance bottleneck compared to the *LAExecUnit's* reduction unit, which does not need explicit synchronization since only one heavyweight thread is performing the parallel processing.

The reduction unit inside *LAExecUnit* has been implemented in fixed-function linear algebra accelerators, such as (Morris, 2005), but those implementations were rigid, single purpose circuits for solving specific problems. The *LACore* takes the reduction unit concept and applies it generically to a broad range of linear algebra applications.

Novel Vector, Scalar, or Multi-Stream Vector output

In Figure 4-2, the contents of the *LAExecUnit's* datapath show there is a set of parallel nodes, called *VecNodes*, and a binary tree of nodes called *ReduceNodes* followed by an *AccumulateNode*. A more detailed image is seen in Figure 4-3. All input data flows in parallel through the *VecNodes* to produce an output vector of data. This output vector can either be fed to the *ReduceNodes* or be fed directly to the output *LAMemUnit* D. This allows the execution unit to take 3 input vectors and either produce 1 output vector, a scalar value, or a novel output called a "multi-stream" vector, depending on the instruction. The latency of a vector output is only the latency of the *VecNode* array, while the latency of reducing to a scalar value or a multi-stream vector is the sum of the *VecNode* latency in addition to the latencies of each column of *ReduceNodes* and the

`AccumulateNode`. The machine instruction determines whether to take the `VecNode` vector output or the `AccumulateNode` scalar output by using a MUX before the D queue.

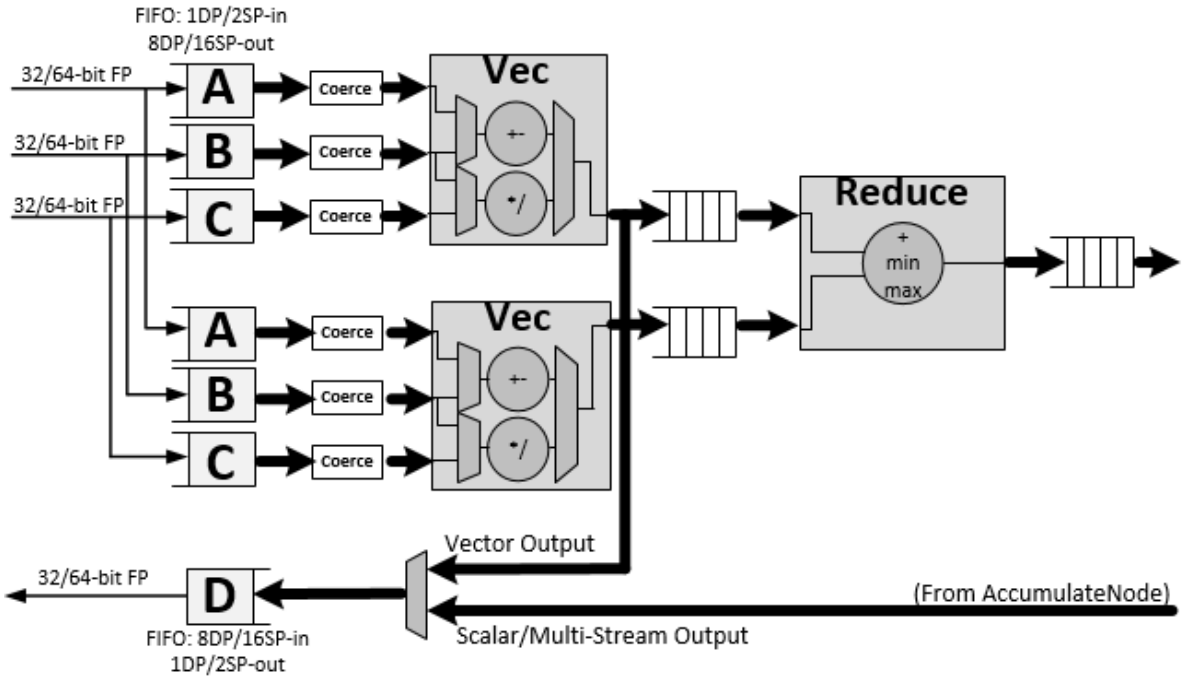

Figure 4-3: LAExecUnit datapath details.

When a vector output is configured by the current instruction, the output stream D matches up 1-1 with the input streams A, B, and C. This means the first elements produced by A, B and C will be operated on and produce the first element in stream D, the second element in A, B, and C will produce the second element in stream D, and so forth. When a scalar output is requested, only one element is produced by the `AccumulateNode` and consumed by the output `LAMemUnit` D. The scalar output is used in operations like a large vector-vector dot-product, or finding the maximum value in a vector. The multi-stream output mode is a powerful way to take input streams with N elements and reduce them to an output stream with M elements, where M divides N. Multi-stream mode can be thought of as breaking the input stream into smaller consecutive sub-streams, and reducing each of those sub-streams to their own scalar. The resulting output stream is then composed of these scalar elements.

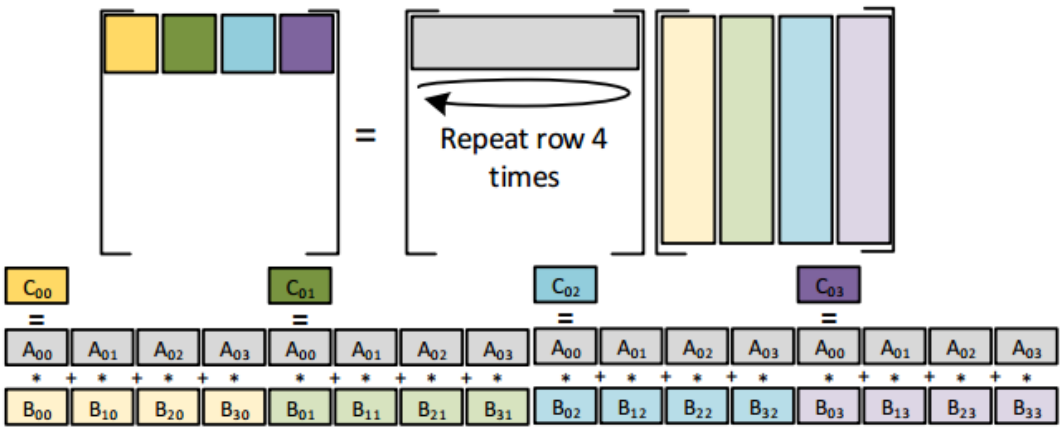

Figure 4-4: LAExecUnit Multi-stream output, DGEMM example.

An example use case for multi-stream output mode is the double-precision Matrix-Matrix multiply kernel, also known as DGEMM in the BLAS Library (Lawson, 1979). DGEMM is a core routine used by many higher-level linear algebra routines. Figure 4-4 shows how DGEMM is executed in the `LAExecUnit`'s datapath using the multi-stream output mode, where a single row in matrix C is the result of computing the dot-product of a row in matrix A with each column in B. The advantage of the multi-stream mode is that it can perform entire complex operations through a single instruction, which allows even more memory-access and data-execution overlap, without having to reissue instructions between each dot-product operation. For example, in Figure 4-4, while dot-product for C_{00} is being computed in the datapath, the elements required for the dot-product for C_{01} are being fetched from memory. Additionally, since the datapath is so deep (high-latency) due to the multiple levels of `ReduceNodes`, it is necessary to fill the datapath with dot-products of multiple sub-vectors at the same time for full utilization.

There have been other parallel computers, mostly for multimedia applications, that have implemented something similar to the `LACore's` multi-stream output mode. One example is the "Matrix Oriented Multimedia" extension (MOM), which provides an instruction set for doing efficient matrix operations on special registers (Corbal, 1999). However, MOM is a traditional load-store architecture that requires moving data in and out of registers before operating on them. Furthermore, MOM is not a vector processor and therefore only operates on matrices up to 16x8 in size, which is a large limitation for general purpose linear algebra calculations on large matrices – but this was not the exact target application for MOM anyways.

The last major advantage of the multi-streaming mode is that it provides a greater opportunity to fully utilize the `LAExecUnit`, especially when using shorter vectors. The `LAExecUnit` uses *stream-ids* as book-keeping for the different sub-streams in multi-streaming mode in order to only reduce elements within the same stream. This logic has no effect on the `VecNodes` but it does slightly complicate the `ReduceNodes` control circuitry, and can cause stalling if the sub-stream count is not a multiple of $(SimdWidth * VecNodes)$.

Eight Unique Vector Operations and Three Unique Reduction Operations

Another unique feature of the `LAExecUnit` is the ability to perform eight different arithmetic operations in the `VecNodes`, and three different operations in the `ReduceNodes`, which are itemized in Table 4-1. A total of 24 possible datapath configurations are possible when operating in single-output or multi-stream-output modes, and eight datapath configurations are possible when operating in vector-output mode.

`VecNode` Ops		`ReduceNode` Ops
(A+B)*C	(A+B)/C	sum(X, Y)
(A-B)*C	(A-B)/C	min(X, Y)
(A*B)+C	(A*B)-C	max(X, Y)
(A/B)+C	(A/B)-C	

Table 4-1: LAExecUnit datapath configurations.

A common configuration of the datapath would be for vector dot-product, in which case the `VecNodes` would be configured for $(A * B) + 0$ and the `ReduceNodes` would be configured for $sum(X, Y)$.

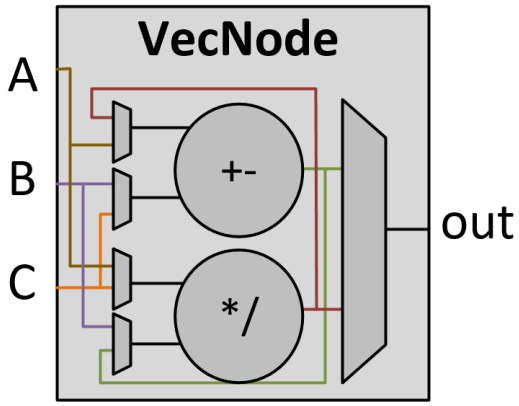

Figure 4-5: VecNode Operation Bypassing.

Another important feature is called *operation bypassing*, which can be seen in Figure 4-5, which is a latency reduction technique that uses inputs to determine if a part of the `VecNode` operation can be bypassed altogether. For example, if $(A * B) + 0$ was being computed, it would be a waste of cycles to compute the floating-point addition of $(A * B)$ and 0, so in that case, the datapath control units (which can be seen at above the `VecNodes` in Figure 4-2) directly select the output of $(A * B)$ using the MUX after the multiplier/divider. A similar bypassing technique can be used if the user just wants to perform $(A + B)$, in which case C would be a constant, 1. The datapath control unit would select $(A + B)$ directly in this situation using the MUX after the adder/subtractor in the `VecNode`, and ignoring the output of the multiplier/divider. All this latency hiding through bypassing is done transparently by the datapath control logic, and is not controlled directly by the user. All the user needs to do is specify the right input values for A, B, and C for the bypassing to transparently take effect.

Novel Simultaneous Dual-Precision Hardware

To support a diverse range of applications, the `LACore` supports both single-precision and double-precision (32-bit and 64-bit) floating point arithmetic, since support for both precisions is standard in all mainstream hardware vendors nowadays. To maximize the performance while minimizing hardware redundancy, the `LAExecUnit's` datapath is dual-precision. In other words, the `VecNodes`, `ReduceNodes`, and `AccumulateNode` all support single-precision and double-precision arithmetic by reusing the same hardware. This results in area reduction and power-reduction while marginally increasing the latency. For the entire `LAExecUnit` to be capable of dual-precision arithmetic, the basic operations that need to be supported in dual precision are *add, subtract, multiply, divide* and *compare*. Implementations for each of these operations have already been presented in literature, and shown to barely affect latency on the critical path, which means it is optimal to implement the `LAExecUnit` using dual-precision components. A summary of these dual-precision components is found in Table 4-2.

Dual-Precision Operation	Latency	Frequency	Source
Add	**5 cycles**	**1.5 GHz**	(Akkaş, 2008)
Subtract	*see Add*	*see Add*	*see Add*
Multiply	**4 cycles**	**1 GHz**	(Jaiswal, 2015)
Divide	**15/11 cycles (DP/SP)**	**1 GHz**	(Jaiswal, 2016)
Compare	*see Add*	*see Add*	*see Add*

Table 4-2: Dual-Precision Hardware Latencies and Frequencies.

Dual-precision hardware can be used for mixed-precision HPC applications such as Iterative Refinement techniques, which are used to improve the accuracy of the solution to a system of linear equations (Kurzak, 2016). Since floating point arithmetic has inherent error due to round-off, the result of solving $Ax = b$ for x may have an unacceptably large error. To fix this, iterative refinement can be applied until x converges to an acceptably low error. Kurzak has shown that when using mixed precision, the initial factorization of $PA = LU$ during decomposition can be performed in single-precision, while the iterative refinement can be performed in double-precision, with very little effect on overall accuracy or performance. The `LACore` is well suited for this type of application, since it targets linear algebra applications and supports both single-precision and double-precision arithmetic using the same hardware. Another useful application of dual-precision hardware is in approximate computing, where the algorithms are inherently error tolerant, since the input data is noisy (Nepal, 2016). In these situations, it may be advantageous to also reduce the accuracy of the calculations by using a lower precision floating point format, something the `LACore` is again suited for.

The unique contribution to dual-precision hardware that the `LACore` adds, however, is the ability to operate on both 32-bit floating point and 64-bit floating point and produce either 32-bit or 64-bit outputs regardless of input precision. For example, if the current `LACore` instruction was $(A + B) * C = D$, and the inputs A and B were 32-bit floating point, while input C and output D were 64-bit floating point, the `LAExecUnit` will be able to handle this by *coercing the inputs to the output's precision* using special "coerce units", which can be seen in Figure 4-3. So, in our current example, inputs A and B will be coerced to 64-bit streams in hardware right before entering the `VecNodes`, and the entire pipeline will operate in 64-bit precision, since this is precision of output stream D. A more detailed view of how the coerce unit operates is seen in Figure 4-6, which makes clear that the coerce unit is a glorified MUX that selects based on the input stream's type and the output stream's type.

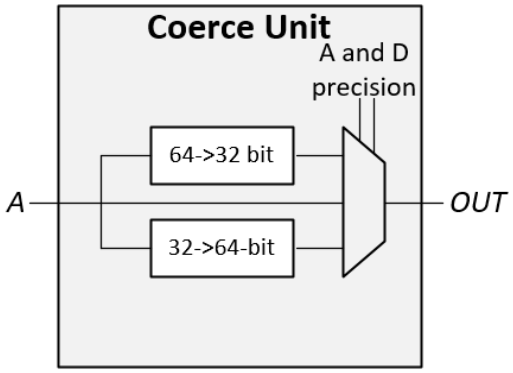

Figure 4-6: Coerce Unit used for Simultaneous Dual-Precision

To knowledge, no other parallel computing architecture allows the seamless usage of both 32-bit and 64-bit floating point at the same time, and typically require an explicit conversion operation on a vector to convert it between 32-bit and 64-bit floating point. So, in applications where the need to quickly switch precisions or use different precisions simultaneously is important, the *LACore* provides a unique advantage over other computing platforms.

Packed SIMD stream processing

As seen in Figure 4-3, not only does the *LAExecUnit* have parallel processing *VecNodes* and *ReduceNodes*, but it also works on packed SIMD data. Packed SIMD is orthogonal to vector processing and it is a technique typically used when traditional vector processing is not used. For example, Intel's Multimedia Extensions (MMX), Streaming SIMD extensions (SSE), and Advanced Vector Extensions (AVX) are all established packed SIMD instruction sets used for parallel data processing for the x86 platform (Patterson D. A., 2013). One of the major drawbacks of SIMD-only parallel architectures is that they define an instruction set for a specific SIMD width, so the result is that new ISAs need to be developed for each successively larger SIMD width. The *LACore* does not suffer from this drawback since it is a Large-Format vector accelerator, and can operate on arbitrarily sized vectors using a few instructions.

The *LACore* uses combined vector processing with packed-SIMD to maximize the ratio of control hardware to data-processing hardware and to also improve throughput. For example, instead of using a 64-byte wide SIMD buffer, if the *LAExecUnit* had used 64 *VecNodes* that operated on one 8-byte input each, then the control circuitry and infrastructure would have to be duplicated for each of the 64 *VecNodes*. But with the *LACore*'s combined vector processing and packed SIMD approach, the control circuitry only needs to be duplicated for each of the 8 *VecNodes*. Although both approaches would produce the same results in the end, the packed SIMD approach requires much less control circuitry, and is therefore more area efficient. Additionally, the *LACore*'s combined vector processing and packed-SIMD approach keeps the throughput of the datapath the same, while reducing the latency, since adding more *VecNodes* in parallel results in a deeper *ReduceNode* tree with more layers.

Separate Clock Domain

Industry-leading scalar CPUs like the Intel Xeon E7 are now capable of running over 3-4 GHz (Intel, 2017), however, the dual-precision functional units listed in Table 4-2 only have been confirmed to operate in the low 1 GHz range. Additionally, NVIDIA GPUs only operate in the low to mid 1.5 GHz as well (NVIDIA, 2016). In order to not throttle the scalar CPU and memory system's frequency due to the

slower *LAExecUnit* datapath, two separate clock domains are used in the processor: 3-4 GHz for the scalar CPU and 1-1.5 GHz for the *LAExecUnit*. Another reason for using a slower clock speed in the *LAExecUnit* datapath is that it allows the *LAMemUnits* to stream data at a proportionally higher rate compared to how fast the *LAExecUnit's* datapath is consuming the streams, which is helpful in allowing the datapath to reach its theoretical peak throughput.

Latency and Throughput

Performance for the *LACore's* datapath is summarized in Table 4-3 below. The numbers given assume an 8 *VecNode* configuration (4, 16 or 32 *VecNodes* is also possible), 8-wide SIMD as the fundamental operand, and Dual-Precision hardware latencies taken from Table 4-2. The scalar CPU is assumed to be running at 2-3x the clock speed listed in Table 4-3, which is the clock speed of the *LAExecUnit* datapath. Although the reduce operations and multi-stream operations have a higher latency, they result in a higher overall throughput than the vector processor, since they utilize more of the *LAExecUnit's* hardware at any given time.

Performance of *LAExecUnit* datapath for all configurations using Dual-Precision hardware latency estimates					
Output Type	***VecNode* config**	***ReduceNode* config**	**Datapath Latency (cycles)**	**FLOP/cycle sustained**	**Throughput @ 1 GHz sustained**
Scalar / Multi-Stream (32-bit)	(A+0)*C	ALL	19 *(4+15)*	240 *(128+112)*	240 GFLOP/s
	(A+0)/C	ALL	29 *(14+15)*	240 *(128+112)*	240 GFLOP/s
	(A+B)*C	ALL	24 *(9+15)*	368 *(256+112)*	368 GFLOP/s
	(A+B)/C	ALL	34 *(19+15)*	368 *(256+112)*	368 GFLOP/s
Scalar/Multi-Stream (64-bit)	(A+0)*C	ALL	19 *(4+15)*	120 (64+56)	120 GFLOP/s
	(A+0)/C	ALL	33 *(18+15)*	120 (64+56)	120 GFLOP/s
	(A+B)*C	ALL	24 *(9+15)*	184 (128+56)	184 GFLOP/s
	(A+B)/C	ALL	38 *(23+15)*	184 (128+56)	184 GFLOP/s
Vector (32-bit)	(A+0)*C	-	4	128	128 GFLOP/s
	(A+0)/C	-	14	128	128 GFLOP/s
	(A+B)*C	-	9	256	256 GFLOP/s
	(A+B)/C	-	19	256	256 GFLOP/s
Vector (64-bit)	(A+0)*C	-	4	64	64 GFLOP/s
	(A+0)/C	-	18	64	64 GFLOP/s
	(A+B)*C	-	9	128	128 GFLOP/s
	(A+B)/C	-	23	128	128 GFLOP/s

Table 4-3: Theoretical Latency and Throughput of LAExecUnit datapath.

Again, these performance figures could easily double or halve, depending on how wide the packed-SIMD is or how many parallel *VecNodes* there are; the numbers chosen were picked because

they provide the best performance for a range of applications while keeping the area footprint as small as possible. The performance tuning was done with gem5 parameters sweep, which will be discussed shortly.

For the `VecNode` config column in Table 4-3, only two out of the eight configurations are shown, since subtraction has the same latency as multiplication, and the order of the add/subtract and the multiply/divide does not affect the overall latency. The latencies for scalar-output and multi-stream-output are the sum of the `VecNode` latency and the `ReduceNode` tree latency, corresponding to the two values in parenthesis in the "Datapath Latency" column. Similarly, FLOP/cycle is the sum of the `VecNode` array throughput, and the `ReduceNode` tree throughput, at either 8-SIMD for double-precision and 16-SIMD for single-precision. It also must also be stressed that these are the theoretical peak numbers *without respect to the* `LAMemUnit` performance. In other words, the numbers in Table 4-3 are the theoretical upper-bound for performance for a single `LACore`. In applications with a low computational complexity, the actual peak performance will be much lower.

NVIDIA GP100 Performance per Streaming Multiprocessor					
SMs	**FP32 CUDA cores/SM**	**FP64 CUDA cores/SM**	**Clock**	**Peak FP32 GFLOPs /SM**	**Peak FP64 GFLOPs /SM**
56	64	32	1.3 GHz	189	95

Table 4-4: Performance of two NVIDIA GP100 Streaming Multiprocessors (NVIDIA, 2016).

It is useful to compare the performance figures in Table 4-3 with leading industry hardware platforms. One leading platform is NVIDIA's recent GP100 which uses the Pascal architecture. A summary of its performance is found in Table 4-4 above. Given that the `LACore` is roughly equivalent to two Streaming Multiprocessors in an NVIDIA GPU in terms of area and memory resources used (see the Design Area Estimates section), it makes sense to compare the `LACore's` performance to two Streaming Multiprocessors. Scaling the `LACore` can be done by using a multi-core design with tens or hundreds of `LACores` on a single chip, which then would be a good time to compare to the GP100's overall performance. The performance results from Table 4-3 and Table 4-4 show that the `LAExecUnit's` datapath has roughly the same theoretical throughput as two Streaming Multiprocessors; the `LACore` can achieve 368 GFLOP/s single-precision and 184 GFLOP/s double-precision at theoretical peak-performance, while the two SMs can achieve 378 GFLOP/s single-precision and 190 GFLOP/s double-precision. So the `LACore` and the GP100 SMs, two parallel-computing architectures, have roughly equal area and theoretical performance numbers, but their architectures are radically different. As we will discuss in the Benchmarks and Evaluation section, the `LACore`

architecture achieves better real-world results than the GPU SMs for a range of HPC applications despite these similar statistics.

Design Space Optimization Using Parameter Sweeps

The `LAExecUnit`'s datapath parameters were not arbitrarily chosen, but were selected after studying the linear algebra applications that were being targeted and finding the hardware configuration that performed best for all of them. The methodology consisted of selecting a handful of linear algebra applications and implemented them for the `LACore` architecture, then implementing the `LACore` architecture in the gem5 cycle-accurate simulator (Binkert, 2011), and then determining the parameters to tune and benchmark the selected applications for each of the parameter configurations.

For the `LAExecUnit`'s datapath, the relevant parameters were the SIMD width and the number of `VecNodes` (which also determines the number of `ReduceNodes`). The applications chosen were *Double-Precision Matrix-Matrix Multiply* (DGEMM), a BLAS-3 kernel (Lawson, 1979), and *Double-Precision Triangular-Solve* (DTRSM), which is another BLAS-3 kernel. The reason these two kernels were selected is because they have been shown to be the minimal set of kernels needed to be implemented in order to implement the rest of the BLAS-3 routines, as discussed in (Van Zee, 2015). In other words, we can implement most of complex routines in the BLAS API by only implementing the DGEMM and DTRSM kernels, and since BLAS is a fundamental API in solving computational linear algebra problems, this seemed like a simple and effective way to optimize the `LAExecUnit`'s datapath for most linear algebra applications.

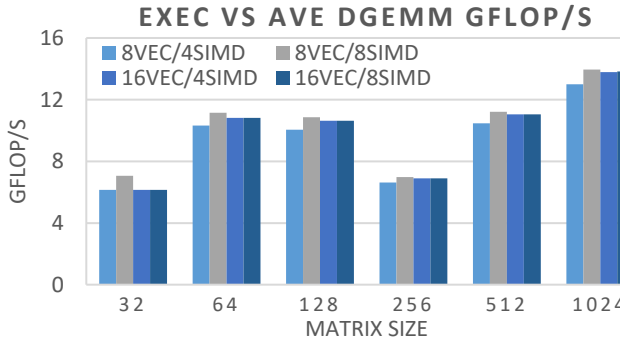

Figure 4-7: LAExecUnit parameter sweep using DGEMM average performance.

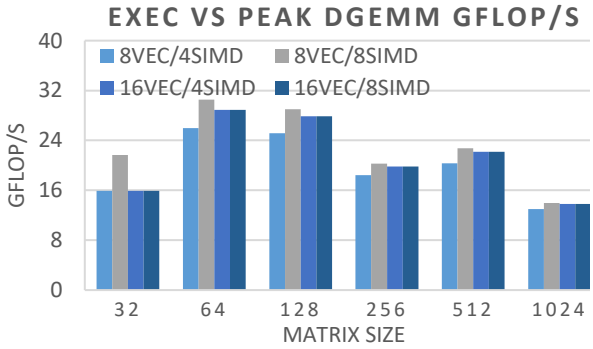

Figure 4-8: LAExecUnit parameter sweep using DGEMM peak performance.

The DGEMM benchmark results for the parameter sweep are shown in Figure 4-7 and Figure 4-8. Figure 4-7 shows how the SIMD-width and `VecNode` count affect the average DGEMM performance across the four different variations of the DGEMM kernel (the variations just involve

transposing input matrices), while Figure 4-8 shows how these parameters affect the peak DGEMM performance from all the variations. It is evident that the 8-SIMD double-precision (512-byte buffers), and 8 *VecNode* configuration is optimal at every matrix size for both peak and average performance.

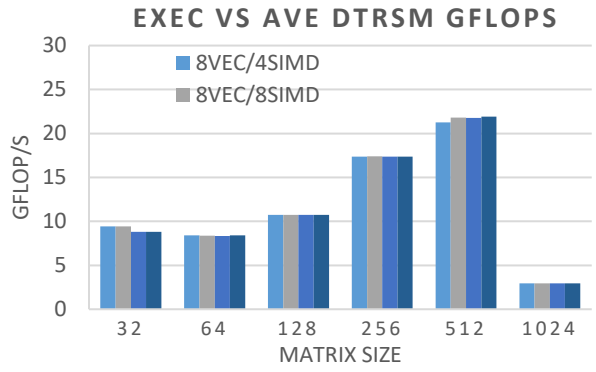

Figure 4-9: LAExecUnit parameter sweep using DTRSM average performance.

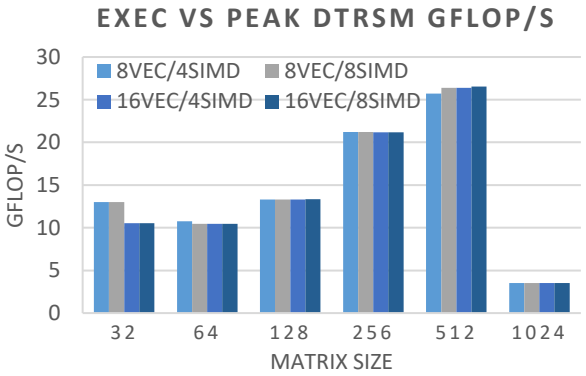

Figure 4-10: LAExecUnit parameter sweep using DTRSM peak performance.

The parameter sweep results of the DTRSM average performance are shown in Figure 4-9, while the peak DTRSM performance is shown in Figure 4-10. There are 16 variations of the DTRSM kernel, and all of them were tested and averaged to produce the results in Figure 4-9, while the highest-performing variation is shown in Figure 4-10. Unlike the DGEMM parameter sweep results, the DTRSM results show that the SIMD-width and *VecNode* count have little effect on the kernel's performance on the *LACore*. This is dues to DTRSM being a more memory-bound application than DGEMM, so the memory accesses are the bottleneck and sweeping over the datapath's parameters will have a negligible effect. Therefore, there is no reason to use a 16 *VecNode* configuration, since this would just take up more area without providing any performance benefits.

To summarize the parameter-sweep findings: using the DGEMM and DTRSM applications for performance tuning, the configuration of 8-wide double-precision SIMD and 8 *VecNodes* was deemed optimal with respect to both performance and area utilization of the *LACore*.

4.3.2 LAMemUnit Architecture

The *LAMemUnits* are responsible for taking the wide range of data configurations and converting them into a universal stream format for the datapath. From Figure 4-11, we can see a more detailed architecture of an input *LAMemUnit*, which uses an instruction and an *LACfg* configuration register as the only inputs needed to start interacting with the *LACache* and scratchpad with 128-byte

line-sizes. Internally, the *LAMemUnit* has three major components: A Next-Address Generator FSM, a FIFO controller, and a Memory Controller. As mentioned previously, the *LAExecUnit* datapath runs at 2-3x slower clock speed than the rest of the *LACore* and scalar CPU since the *LAMemUnit* must be able to fetch and write data fast enough to allow the datapath to attain peak performance and the dual-precision functional units are inherently slower. Additionally, data is written to the FIFOs on both edges of the CPU's main clock, and written 2-items at a time. The combination of the above three performance enhancements allow the *LAMemUnit* to fill the FIFOs at the maximum theoretical rate the *LAExecUnit's* datapath will consume 8-wide double-precision SIMD blocks or 16-wide single-precision SIMD blocks. But this maximal rate is only achievable if the *LAMemUnit* can fetch data that fast from the *LACache* or scratchpad, which is only possible in high-spatial locality applications.

Finally, the *LAMemUnits* support Large-Format data sources, meaning that input and output vectors/matrices can be arbitrarily large, similar to archetypal vector processors (Cray, 2003). The major distinction from vector architectures is that in the *LACore*, data sources are not limited to strided vectors.

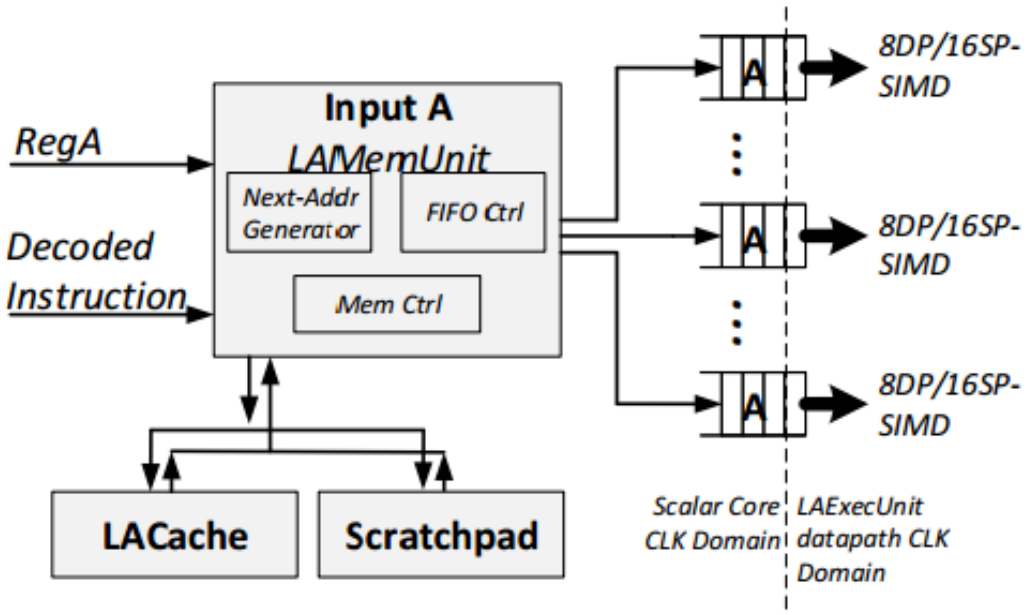

Figure 4-11: LAMemUnit High-Level Design.

Novel Scalar, Vector, Matrix and Sparse Configuration

The *LAMemUnit's* Next-Address Generator is a Finite-State-Machine that is capable of generating the address of arbitrary scalars, vectors, and dense or sparse matrices. For scalars, it simply

generates the location of the element, and the Mem-Ctrl within the `LAMemUnit` will then produce that scalar repeatedly. For vectors, the FSM generates the next address from a configured *base-address, stride, skip,* and *count* according to the following equation:

$$NextAddr(i) = BaseAddr + SIZE * (i * stride + skip * (i/count))$$

Here, $SIZE$ is four for single-precision and eight for double precision, and i is the current offset within the vector. This general formula for vectors allows the user to specify arbitrary submatrices within a larger matrix to operate on. Combining this feature with the multi-stream datapath configuration provides a simple and powerful interface for operating on arbitrarily sized and positioned dense matrices in memory. An example of the different vector configurations is seen in Figure 4-12. An additional benefit of being able to express the vector layout as a simple equation using *address, stride, skip* and *count* is that the `LAMemUnit` can perform pre-fetching ahead of the datapath.

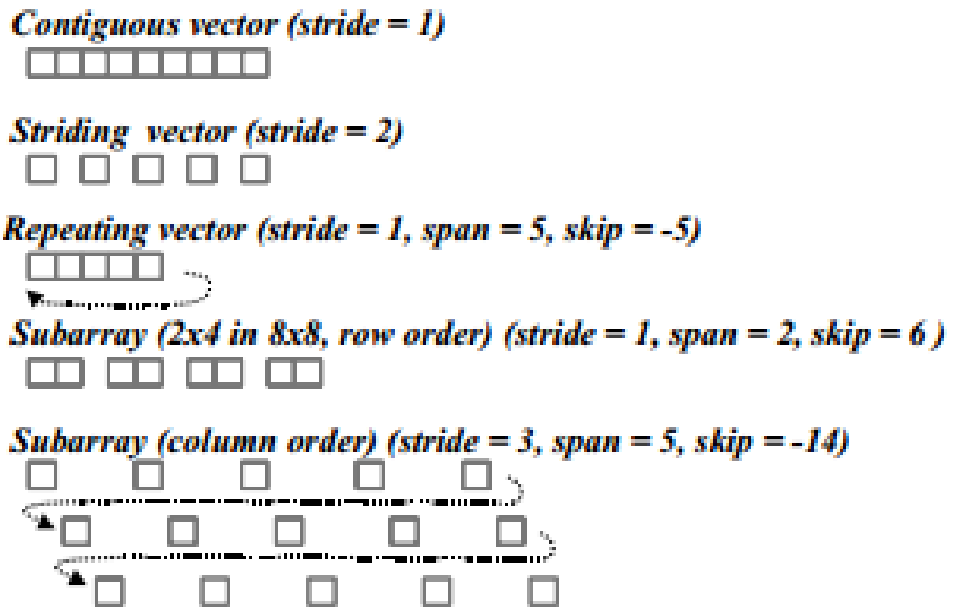

Figure 4-12: LAMemUnit Vector stride, skip, count config. Image from (Ciricescu, 2003).

It should be noted that the *address, stride, count* vector configuration is implemented in many existing vector architectures, like Cray X1, Hwacha, and RSVP (CRAY, 1977) (Lee Y. O., 2015) (Ciricescu, 2003), and is more generally known as a *gather-scatter* interface. However, from the above three, only

RSVP appears to provide the additional *skip* configuration that allows the user to select sub-matrices from larger matrices in memory.

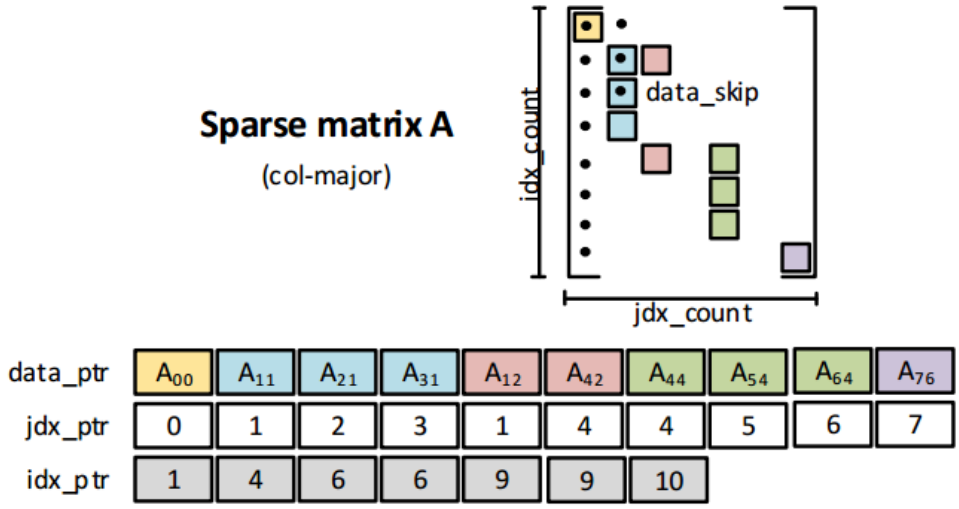

Figure 4-13: LAMemUnit Sparse Matrix Configuration, with all 6 config fields shown.

The last, and most novel feature of the Next-Address generator FSM in the `LAMemUnit` is that it can be configured for sparse matrix input or output, as seen in Figure 4-13. The `LAMemUnit` only processes sparse matrices that are similar to the *Harwell-Boeing* sparse matrix format (Duff, 1989). The *Harwell-Boeing* format, also known as *Compressed Column Storage* format (CCS), requires a column-pointer, a row-pointer and a data-pointer to arrays in memory. The `LAMemUnit` generalizes this to allowing both column-major and row-major matrices in memory, by using pointers called `idx_ptr`, `jdx_ptr`, and `data_ptr`, which point to the major-index, the minor-index and the data-array respectively. This generalizes the `LAMemUnit` to be able to read or write Compressed Column Storage or the Compressed Row Storage unambiguously.

Additionally, three more parameters need to be specified for the `LAMemUnit` to be able to access sparse matrices: `idx_count`, `jdx_count`, and `data_skip`, which provides the rest of the information needed for the `LAMemUnit` to interact the sparse matrix with other input streams such as vectors or scalars, which is described in the next section. The `idx_count` and `jdx_count` simply state the dimensions of the uncompressed matrix, while the `data_skip` tells the `LAMemUnit` how many elements in the *uncompressed matrix* to skip. Together, these last three configuration parameters allow multi-core parallelization of processing sparse matrices efficiently, since each `LACore` can easily determine the offsets in `idx_ptr`, `jdx_ptr` and `data_ptr` that it needs to skip to when

inputting or outputting elements. Also, it is possible for multiple cores to *write* to different sections of the sparse matrix in memory with all 6 of these parameters (with proper synchronization of course).

An additional novel feature of the `LAMemUnit's` sparse matrix configuration is that it also allows reading sparse matrices *transposed*. All the necessary information in the six sparse matrix configuration parameters give the Next-Address generator FSM enough information to calculate where the next address is. Being able to read and transpose arbitrary sections of sparse matrices and generate and abstract stream from the data is a unique feature to the `LAMemUnit` that, to knowledge, does not exist in current literature.

Novel Generic Item-Stream Interface

As mentioned above, the `LAMemUnit` provides a powerful abstraction for turning heterogeneous vectors and matrices in memory into generic streams of data for the `LAExecUnit's` datapath to operate on. For example, referring to Figure 4-2, the user may want to specify input A to be a single-precision scalar, B to be a compressed matrix in column-storage format in memory, and input C to be a long vector in the scratchpad, while outputting the result of the `LAExecUnit` to a double-precision sub-matrix inside a dense matrix in memory, as in Figure 4-14. This could be the equivalent of computing a sub-block of an output dense matrix in DGEMM by multiplying an input sparse matrix and an input dense matrix. With the `LACore`'s ISA, this is all entirely possible by simply configuring the input and output `LACfgs` appropriately and then executing an `LACore` instruction. No other parallel computing architecture to knowledge provides a similar interface for turning heterogeneous sets of scalars, vectors and sparse or dense matrices into generic streams that can easily interoperate.

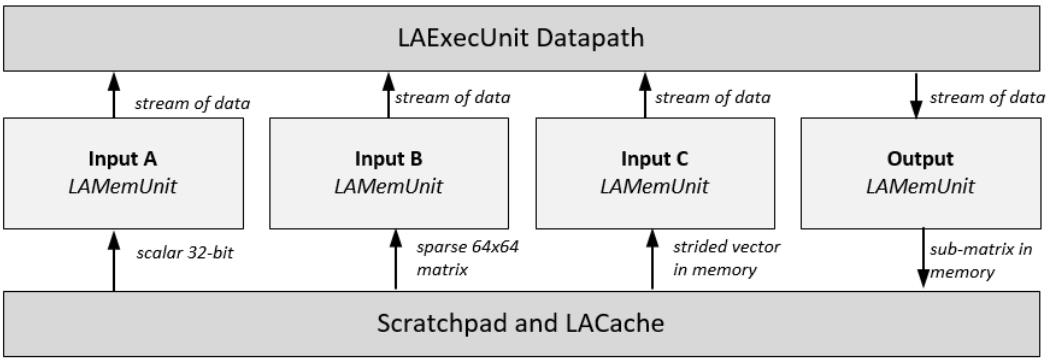

Figure 4-14: LAMemUnit Generic Stream Interface.

There are a handful of incompatible input and output configurations, which can be seen in Table 4-5 below. It is always safe to use a scalar input stream, but you can only use a scalar output stream if

the datapath is configured for a scalar output. For vector and multi-stream output streams, the datapath must be configured for multi-stream or vector output modes. Finally, when using multi-stream output, there are further restrictions on the input vectors: there must be a compatible *count* or `idx_count` field for all vectors and sparse matrices, so the datapath knows how many segments to partition the input streams into. For example, if the datapath was configured for multi-stream, with 100 input elements and 20 output elements, any input vector must have a *count* field of 5, and any sparse-matrix config must have an `idx_count` field of 5 or a `jdx_count` of 5 if the input sparse-matrix is transposed.

Data Format	Location	Possible	Can be Input?	Can be Output?
Scalar	`LACfg`	Yes	Yes	Only if scalar-out
Scalar	Scratchpad/memory	Yes	Yes	Only if scalar-out
Vector	`LACfg`	Yes	-	-
Vector	Scratchpad/memory	Always	Yes	Only if vector-out or multi-stream
Sparse Matrix	`LACfg`	Yes	-	-
Sparse Matrix	Scratchpad/memory	Always	Yes	Only if vector-out or multi-stream, AND not transposed

Table 4-5: LAMemUnit Possible Configurations.

Scratchpad, Register, or Main Memory Location

The `LAMemUnit` is also capable of reading or writing to 3 different locations: an `LACfg` holding a scalar, the private scratchpad, or the main memory via the `LACache`. A scalar input or output can be in any of the three locations, where the scratchpad and the `LACfg` locations have a single-cycle access latency and the `LACache` has a potentially higher latency on cache misses. A vector and a sparse matrix can only be in the scratchpad or main memory. The use of a scratchpad for intermediate results is not a new concept, and is commonly found in embedded applications and in GPUs (Banakar, 2002) (Lindholm, 2008).

A summary of the possible input and output configurations of the `LAMemUnits` is given in Table 4-5. There are a few configurations that will produce an error in the `LACsrReg` and the data operation will abort. For example, if the datapath is configured for vector-output but the output `LAMemUnit` is configured for a scalar output, this will produce an error. Additionally, trying to read or write vectors from an `LACfg` is impossible and will generate an error condition in the `LACsrReg`.

Cache-line Optimizations

As will be discussed in the *LACache* section later, the cache-line size of the *LACore* system is 128 bytes (16 double-precision or 32 single-precision elements). In order to take advantage of spatial-locality of vector-elements within a cache-line, the *LAMemUnit*'s memory- controller can determine how many of the *next* vector stream's elements are located in the current element's cache block, and it can deliver all of them simultaneously to the *LAMemUnit's* FIFO-controller.

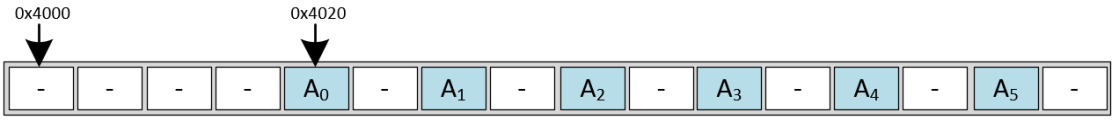

Figure 4-15: LAMemUnit Can read multiple elements per cache line.

As an example, recall the $NextAddr$ equation used by the Next-Address generator FSM:

$$NextAddr0 = BaseAddr + SIZE * ((i + 0) * stride + skip * ((i + 0)/count))$$
$$NextAddr1 = BaseAddr + SIZE * ((i + 1) * stride + skip * ((i + 1)/count))$$
$$NextAddr2 = BaseAddr + SIZE * ((i + 2) * stride + skip * ((i + 2)/count))$$

The *LAMemUnit* can compute the next series of addresses simultaneously, and then determine how many consecutive elements are in the current cache-line by simply truncating the low bits and performing a bitwise-and on the high-bits. Since the cache-line is 128 bytes, the low 7 bits of each address can be truncated, and the high 57 bits (on a 64-bit address space) need to be equal. Figure 4-15 shows how a vector with a stride of 2 and the current vector element having an address of $0x4020$, could have multiple elements per cache line, and therefore increase the memory throughput by accessing them all in one cycle.

Scratchpad Optimizations

As will be discussed in the scratchpad section later, the scratchpad has a 128-byte line size, three 8-byte wide read ports and one 8-byte wide write port. Each bank can be accessed simultaneously and concurrently by all four of the *LAMemUnits*, by using independent buses for each *LAMemUnit* and bank combination. Each *LAMemUnit* memory-controller uses a similar technique to the cache-line optimization technique. The *LAMemUnit* computes the next 32 addresses in parallel, and determines how many addresses are within the current scratch-pad line, and then it reads or writes all of these elements to or from the FIFOs connected to the datapath.

4.4 LACfg and LACsrReg Registers

4.4.1 The LACfg Configuration Registers

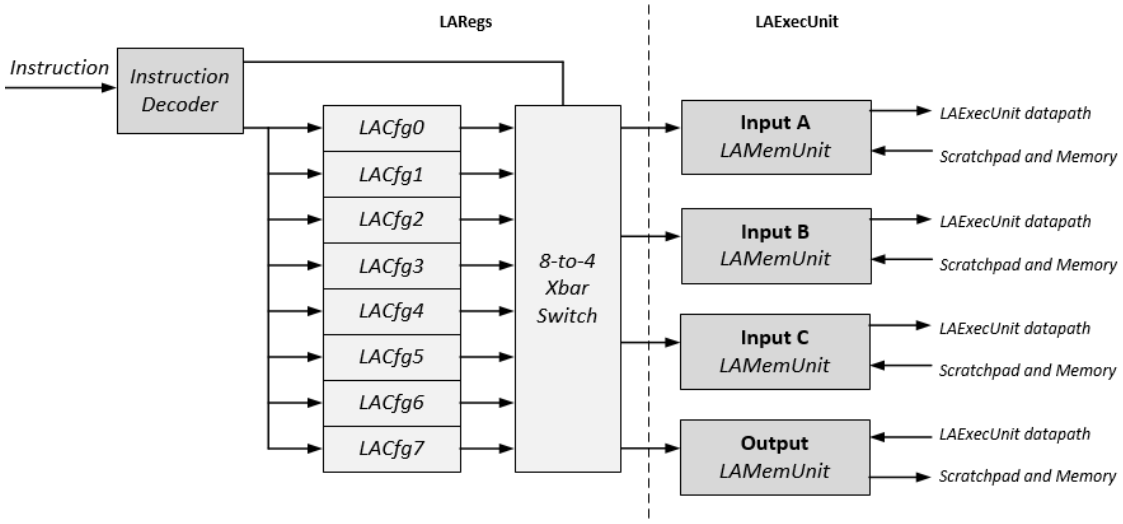

Figure 4-16: LACfg Detailed View.

The `LACore` adds eight configuration registers to the CPU in order to configure the `LAMemUnits` to read and write a wide range of data structures, layouts and precisions, as seen in Figure 4-16. These configuration registers are called `LACfgs` and should not be confused with traditional vector processor "vector registers", which are used to hold data before and after the vector lanes operate on it. The eight `LACfgs` are connected to the four `LAMemUnits` by a crossbar which allows any `LACfg` to provide configuration for any subset of `LAMemUnits` at any given time.

	Bits	Scalar Config	Vector Config	Sparse Config
data0	0-63 *(64-bit)*	32/64-bit scalar or address	address	`data_ptr`
data1	64-127 *(64-bit)*	-	-	`idx_ptr`
data2	128-191 *(64-bit)*	-	-	`jdx_ptr`
layout0	192-223 *(32-bit)*	-	stride (signed)	`idx_count`
layout1	224-255 *(32-bit)*	-	count (unsigned)	`jdx_count`
layout2	256-287 *(32-bit)*	-	skip (signed)	`data_skip`
location	288-289 (2-bit)	0=`LACfg`, 1=mem, 2=scratch	1=mem, 2=scratch	1=mem, 2=scratch
vector	290 *(1-bit)*	FALSE	TRUE	TRUE
sparse	291 *(1-bit)*	-	FALSE	TRUE
transpose	292 *(1-bit)*	-	-	*1=yes, 0=no*
double	293 *(1-bit)*	*1=yes, 0=no*	*1=yes, 0=no*	*1=yes, 0=no*

Table 4-6: LACfg Configuration Space

As seen in Table 4-6, the *LACfg* has 294 bits of configuration, and it has four classes of fields*: data, layout, location* and *flags*. The *data* fields (*data0, data1, and data2)* are used to inform the *LAMemUnit* where the start of the data is located or what the value of the data is. The *layout* fields *(layout0, layout1, and layout2)* are used to inform the *LAMemUnit* how the multi-element data are laid out in memory. These fields do not apply to scalar data sources, since it only has one element. The *location* field informs the *LAMemUnit* which address space the data are located in. There are currently three different locations: an *LACfg* (which only applies to scalars), the scratchpad or the main memory. A value of 0x3 in the *location* field will generate an error condition and abort any data operations. The *flags* fields are just a collection of Boolean flags that inform the *LAMemUnit* about which type of data is present in the register – so it needs to read these flags before processing any of the other fields.

Typically, the four registers for three input streams and one output stream of the *LAExecUnit* are configured in four separate instructions before the execution instruction, while the datapath is configured by the execution instruction itself. So, at most, five instructions need to be issued to configure and execute on an *LACore*. One feature of the *LACore* that helps minimize the number of instructions needed for configuration is having eight *LACfgs* to configure only four *LAMemUnits*. In any non-trivial application, even DGEMM, it is necessary to issue a handful of different instructions to the *LAExecUnit*, each of which require different *LAMemUnit* configurations. Instead of running four *LACfgs* directly to four *LAMemUnits*, which is the simple solution, an 8-to-4 crossbar is placed in between the *LACfgs* and the *LAMemUnits*, which provides a simple and flexible way for the user to choose which registers should be used as the three sources and one destination register for any given operation, which could save multiple cycles of configuration each iteration of tight inner kernels.

4.4.2 The LACsrReg Control Status Register

As touched on in the *LAMemUnit* and *LACfg* section, due to the large amount of configuration freedom given to the user, there are many potential sources of exceptions. In the case of an exception, a flag in the *LACsrReg* (the control-status register), is asserted, and the current operation is aborted. No future operation will be allowed until the user has cleared the *LACsrReg* with a special instruction. Additionally, the user should always check the *LACsrReg* flags after running an application to make sure no exceptions were generated. A list of the current flags and their bit-positions is given in Table 4-7 below.

The *LACsrReg* is a 64-bit register that can be written to any of the general-purpose registers using the *lxfercsrget* command, which is discussed in the ISA section. More flags can be added as additional features and complexity is added to the *LACore*, which is why a full 64-bit register was used instead of 32-bits.

Bit	Code	Description	Bit	Code	Description
0	*badinsn*	Failed to parse *LACore* instruction	10	*mstdstreg*	sparse dest is *LACfg*
1	*memovrflw*	address outside scratch/mem range	11	*mstsrcreg*	sparse src is *LACfg*
2	*badregloc*	reg loc is 0x3	12	*srccntzero*	src count is zero
3	*floaterr*	Floating point arithmetic error	13	*dstcntzero*	dest count is zero
4	*scaldstvec*	scalar dest but vector output	14	*mstdiffcnt*	count mismatches in multi-stream
5	*scaldstmst*	scalar dest but multi-stream output	15	*mstbadcnts*	count % src count != 0 in multi-stream
6	*vecdstscal*	vector dest but scalar output	16	*spvtrnsout*	sparse dest is transposed
7	*spvdstscal*	sparse dest but scalar output	17	*badalign*	32/64-bit address not 4/8-byte aligned
8	*vecdstreg*	vector dest is *LACfg*	18		
9	*vecsrcreg*	vector src is *LACfg*	19		

Table 4-7: LACsrReg Error Flags.

There are a few errors in Table 4-7 that have not been elaborated on yet. The first is *badinsn*, which arises when the *LACore* receives a malformed instruction. This happens when one of the fields that should be all zeros has a non-zero value, or one of the *lfunc* fields (discussed in ISA section) has an invalid value. The second is *memovrflow*, which is self-explanatory: if the user specifies an address outside of the scratchpad or main memory's size, this error is produced. The *floaterr* flags is asserted if any of the normal floating point exceptions occur, such as divide by zero. There are a few errors that have to do with invalid "count" fields on vectors and sparse matrices. The first two, *mstdiffcount* and *mstbadcnts* have been elaborated in the ***Novel Generic Stream-interface*** section. The last two are *srccntzero* and *dstcntzero*, which arise when the source or destination vector or sparse matrix has a *count,* *idx_count*, or *jdx_count* value of zero, which will cause a divide-by-zero error in the *LAMemUnit's* Next-Address Generator. The *badalign* exception is caused by configuring a scalar, vector or sparse matrix on an unaligned address.

4.5 THE LACORE SCRATCHPAD

Scratchpad Architecture

The *LACore's* scratchpad is a single-bank, multi-port, high-throughput, low-latency private memory with physical addressing disjoint from the main memory address range. The purpose of the *LACore*'s scratchpad is to provide temporary storage for intermediate results in multi-instruction applications. For example, in order to add three vectors, A, B, and C together, $T = A + B$ must be computed and then $T + C$ is computed, where T is an intermediate-result vector. It is faster to write T to the scratchpad than to dynamically allocate space in memory for T, store T there, and then immediately reload T back into the *LAExecUnit*

The approach to make two separate address spaces for the scratchpad and main memory is different than the embedded application approach proposed in (Banakar, 2002), which uses a unified address space and lets the compiler manage the complex task of mapping data to the scratchpad when it sees fit. The approach the *LACore* takes is closer to how IBM's Cell Processor approaches memory management: the programmer is given explicit control over the movement of data between the different memory components of the system (Williams S. S., 2006). The reason this approach was taken over the intelligent-compiler approach is because the configuration space for the *LACore* is very large, and the kernels that execute on the *LACore* are not lightweight kernels that execute on GPUs or multi-media application accelerators; therefore, it would be a difficult task to intelligently map resources to and from the scratchpad over a long-running kernel the way a programmer with a high-level understanding of the algorithm is able to do. Additionally, the main reasons the scratchpad exists is to hide latency of memory interactions, overlap data transfer with data execution effectively, and provide a fast cache for intermediate result. In order to achieve all of these, the programmer must carefully manage where data is placed, even down to the byte address.

Scratchpad Implementation

The design of the scratchpad is seen in Figure 4-17. The scratchpad is 64 kB, and has three independent read ports and one independent-write port. The SRAM cell therefore would be something like a 12-T, with three independent single-ended reads and one double-ended write, a realistic cell in modern VLSI techniques. For example, a 9-read-port/7-write-port is presented in (Sumita, 2005), and the Intel Itanium processor utilized 12-read/8-write ports in its integer register file design (Fetzer, 2006).

Although these SRAM cells were designed for register-file usage, the idea can be applied to the scratchpad, which uses just a fraction of the ports, and therefore transistors, per SRAM cell.

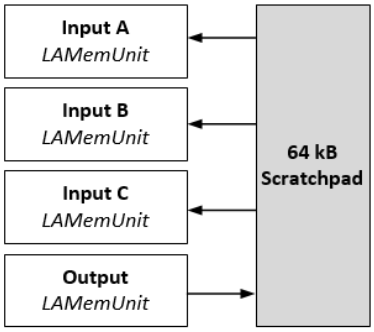

Figure 4-17: LACore Scratchpad Detailed View.

The `LACore`'s scratchpad design contrasts with the NVIDIA GPU shared memory design (Lindholm, 2008), which uses 32 banks accessed simultaneously by many lightweight threads. The reason a GPU needs to have highly-banked scratchpads (or shared memory), is to allow independent threads within a warp to access independent data in the scratchpad at the same time. The `LACore` does not have many lightweight threads, instead it has a heavy-weight complex thread that uses four `LAMemUnits` to simultaneously access the scratchpad, and since only four simultaneous accesses are being done at a time, all that is needed a multi-port SRAM cell to provide this functionality. The three input `LAMemUnits` will read on the first half of the clock cycle, and the one output `LAMemUnit` will write to the scratchpad on the second half of the clock cycle, so all data accesses will remain consistent.

Initially, the `LACore` design went with a highly-banked scratchpad, similar to the GPU design discussed here. However, after preliminary results on the benchmarks, it was clear that the scratchpad was not providing the benefit it should be. After investigating the results further, it became clear that this poor performance was due to using a scratchpad banking model designed for many lightweight threads with the `LACore`, which needed a scratchpad design for a single heavyweight thread.

Optimal Scratchpad Usage

The programmer should try to use consecutively placed elements within the scratchpad as much as possible. Sometimes, accessing a vector with a strided pattern cannot be avoided, such as during a matrix transpose, but the programmer should always try to place data in the scratchpad in a way that will allow sequential reading later in the application. The reason for this, as discussed in the `LAMemUnit` section, is because the `LAMemUnits` have special mechanisms to access multiple items

within the same 128 Byte access-line within the same clock cycle. So packing as many consecutive elements within that 128-Byte address range can potentially improve the memory throughput by up to 16x for double-precision vectors and 32x for single-precision vectors. This can greatly improve the performance of memory-bound applications.

Design Space Optimization: Scratchpad Usage

Similar to the `LAExecUnit`'s datapath parameters, the usage of a scratchpad and the scratchpad's size and access line-size were not arbitrarily chosen, but were decided upon after extensive parameter sweeps using common linear algebra applications on gem5 implementations (Binkert, 2011). Similar to the `LAExecUnit` parameter sweeps, *Double-Precision Matrix-Matrix Multiply* (DGEMM), and *Double-Precision Triangular-Solve* (DTRSM), which are BLAS-3 kernels, were chosen for evaluating the optimal scratchpad configuration (Lawson, 1979). These kernels allow us to implement most of complex routines in the BLAS API with as few lines of code written as possible.

The first parameter sweep involving the scratchpad was testing whether using a scratchpad even provided any performance benefits. All DGEMM and DTRSM variations were performed both with and without a scratchpad present, across a range of matrix sizes. The results for the DGEMM sweep are shown in Figure 4-18, with four different scratchpad configurations shown. "Yes-S" means a scratchpad was used, while "No-S" means no scratchpad was used. The micro-kernel within DGEMM was either "STRIDE" or "PANEL", which were just two variations of the DGEMM kernel that tried to utilize the `LACore` hardware in different ways. The "STRIDE" micro-kernel attempts to maximize the length of the vectors the `LACore` worked with, while the "PANEL" algorithm attempts to minimize the cache-miss rate. The results show that, for both "STRIDE" and "PANEL" algorithms, the performance was much better when a scratchpad is used. As the matrix size grows, the "PANEL" algorithm while using the scratchpad is the optimal configuration.

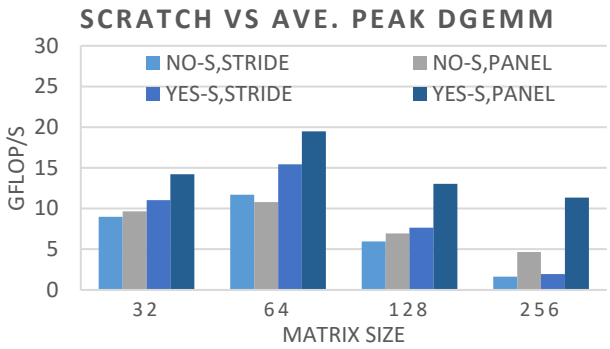

Figure 4-18: Scratchpad Usage vs Peak DGEMM.

Similar results are seen when evaluating all DTRSM variations with and without the scratchpad. Figure 4-19 shows that across all matrix sizes and DTRSM variations, using the scratchpad provides a noticeable performance improvement. Figure 4-20 shows that the peak performance is even more pronounced when using a scratchpad vs not using a scratchpad. Because both the DGEMM and DTRSM results indicate that using a scratchpad is optimal for performance, the `LACore` design chose to use a scratchpad

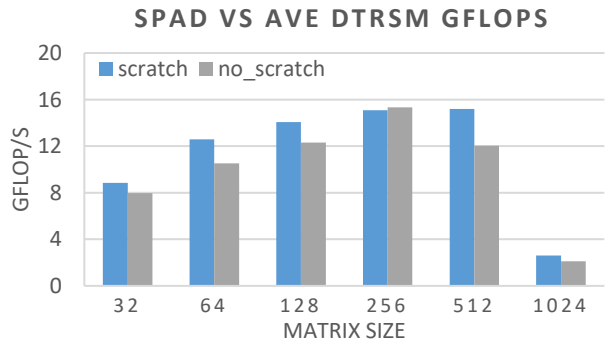

Figure 4-19: Scratchpad Usage vs Average DTRSM performance.

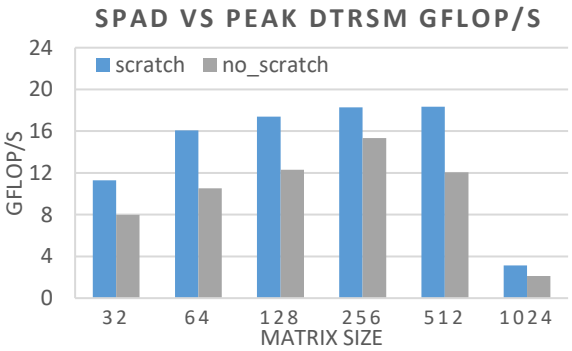

Figure 4-20: Scratchpad Usage vs Peak DTRSM performance.

Design Space Optimization: Scratchpad Size and Line-Size

After determining that a scratchpad should be used, the next step was to determine the size and line-size of the scratchpad. These, too, were found using gem5 parameter sweeps using DGEMM. The scratchpad size was swept across 64 kB – 256 kB, and it had nearly zero effect on the performance of the application at all matrix sizes. Therefore, the smallest evaluated scratchpad size of 64 kB was chosen. The scratchpad access line-size was also determined using parameter sweeps. The access line-size is

similar to the cache line-size of the main memory system, except it is between the `LAMemUnits` and the scratchpad only, and does not necessarily need to be the same size as the cache-line size. The parameter sweep results are shown in Figure 4-21, with the 128-byte line size clearly outperforming the 32-byte and 64-byte line sizes for all matrix sizes.

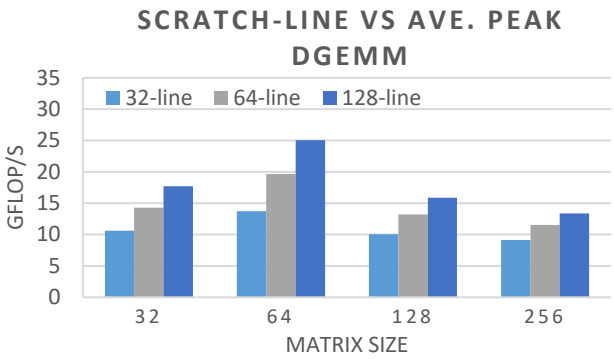

Figure 4-21: Scratchpad Line-Size vs DGEMM performance.

To summarize the parameter-sweep findings for the scratchpad: using the DGEMM and DTRSM applications for performance tuning, it was determined that using the scratchpad provided enough performance improvement to warrant the increase in design area. The size of the scratchpad and the access line-size were determined to be 64 kB and 128 Bytes through additional parameter sweeps using the DGEMM kernel.

4.6 THE LACACHE

Multi-Port, Multi-Bank Architecture

The fourth major sub-block of the `LACore` processor is the `LACache`, which is a special 64 kB cache connecting the `LAMemUnits` to the memory controller, as seen in Figure 4-2. Similar to the scratchpad, it has three read ports and one write port, which connect independently to each of the four `LAMemUnits`, as seen in Figure 4-22. Similar to the CPU's data-cache and instruction-cache, the `LACache` connects to the higher-level L2 cache.

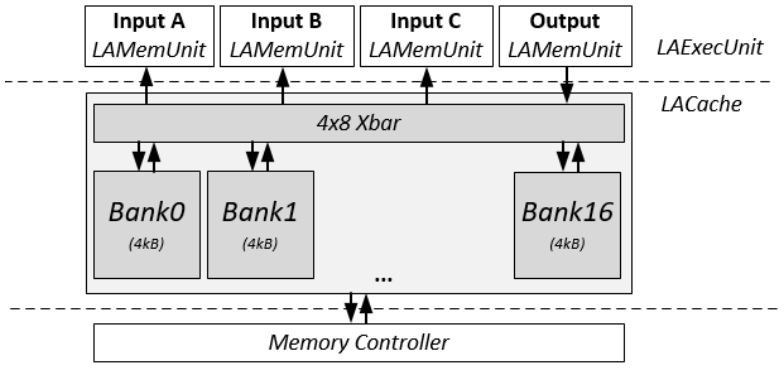

Figure 4-22: LACache Detailed View.

The `LACache` has 16 internal 4 KB, non-replicated banks, with cache-block granularity interleaving. This configuration was chosen after a large amount of manual performance tuning and system parameter sweeps using gem5 and converging on the highest-throughput configuration. The performance of multi-ported multi-banked SRAM caches is discussed by (Rivers, 1997), which compared the throughput of true multi-ported, multi-ported with bank-replication, and true multi-banked caches with up to 16 banks and ports each. It was reported that all three perform comparably on SPECint benchmarks and plateau between 8 and 16 ports or banks. The use of multi-ported and multi-banked caches is common in CPU cache-hierarchies, especially higher-level caches, since multiple-issue multi-processors can issue up to 16 instructions per clock cycle, requiring high cache-bandwidth (Rivers, 1997) (Thimmannagari, 2004).

Optimal LACache Usage

Similar to the scratchpad's optimal usage, the programmer should try to use consecutively placed elements within memory as much as possible when using the `LACache`. This allows the `LAMemUnits` to access more elements per cache-line access, and reduce the total number of lines that need to be pulled in the `LACache`. Reducing the number of lines pulled into the `LACache` is important at larger workload sizes, because the four `LAMemUnits` will begin to start evicting each other's data which leads to poor performance due to the increased number of cache misses. This thrashing phenomenon is clearly visible in almost all of the benchmark results for all cache-based systems (RISC-V, x86 and `LACore`) in the Benchmarks and Evaluation section below. This degeneration of performance due to increase cache-evictions is not something that can be avoided, but can be mitigated with the `LACore` if the programmer accesses data in a sequential fashion.

Design Space Optimization: LACache Usage

The first question for evaluating the *LACache* is whether it is necessary, and this was answered by performing more parameter sweeps using gem5. Similar to the previous parameter sweeps, *Double-Precision Matrix-Matrix Multiply* (DGEMM), and *Double-Precision Triangular-Solve* (DTRSM), which are BLAS-3 kernels (Lawson, 1979), were chosen for evaluating the optimal scratchpad configuration, using the same reasoning for their selection. The parameter sweeps tested multiple high-level cache configurations:

a) Routing all *LAMemUnit* requests through the L1 Data-cache
b) Using a 1-bank, 4-bank or 16-bank *LACache* connected directly to the memory controller
c) Using a 1-bank, 4-bank or 16-bank *LACache* connected directly to the L2-cache

Routing all the *LAMemUnit* requests to the main memory provided a baseline to compare against the *LACache* performance. Additionally, the *LACache* was evaluated with and without the L2-cache sitting under it. The Hwacha Vector Processor uses the approach of directly connecting the vector processing unit to the L2-cache, and skips the L1 cache completely (Lee Y. O., 2015), which is similar to the approach of connecting the *LACache* directly to the memory controller. Results are not shown for the this third configuration where the *LACache* is directly connected to the memory controller, because this configuration performed worst across the board for all applications. The remaining two configurations are examined in the following parameter sweep results.

The parameter sweep results for DGEMM average performance across all four variations is shown in Figure 4-23, and the results for the peak DGEMM performance is shown in Figure 4-24. In both cases, the highly-banked *LACache* design performs better than the Data-Cache design, especially as the matrix size grows.

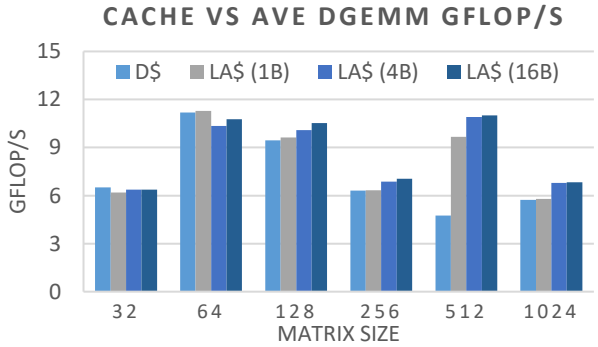

Figure 4-23: LACache configuration vs DGEMM average performance.

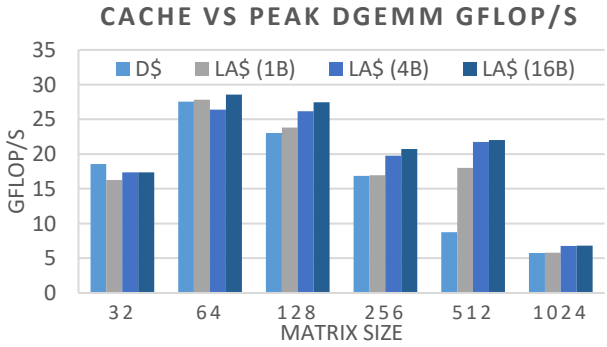

Figure 4-24: LACache configuration vs DGEMM peak performance.

Noticeably different results are seen with the DTRSM parameter sweeps, with the Data-cache design actually performing better than the `LACache` at small matrix sizes. The average DTRSM performance across all 16 variations is shown in Figure 4-25, while the peak performance is shown in Figure 4-26. In both cases, the Data-Cache design performs better for matrix sizes up to 128x128, and anything larger than that, the `LACache` design performs much better. The reason that the data-cache works better at small matrices is because some of the non-`LACore`-specific instructions may access the same data as the `LACore`-specific instructions. If the `LAMemUnits` route all requests through the Data-Cache, then both sets of instructions will enjoy cache-hits for that data. However, when the `LAMemUnits` route all their requests through the `LACache`, the non-`LACore` instructions will suffer from cache misses for the data that the `LAMemUnits` just accessed. As the matrix size grows, the data shared between both sets of instructions shrinks compared to the total data accessed by the `LAMemUnits`, and this data-sharing has a negligible effect on performance.

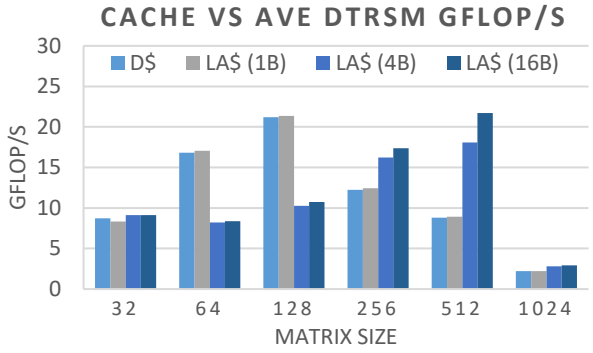

Figure 4-25: LACache configuration vs DTRSM average performance.

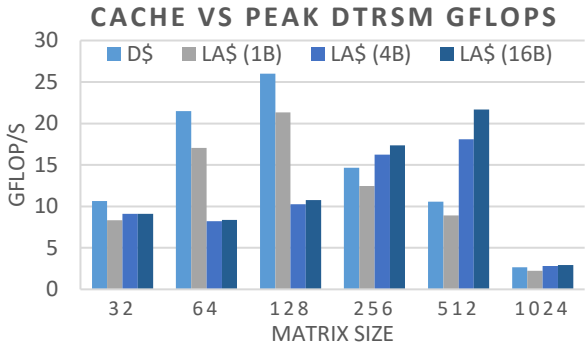

Figure 4-26: LACache configuration vs DTRSM peak performance.

Design Space Optimization: Cache Line Size

The cache line size was also chosen to be 128-bytes after extensive parameter sweeping with gem5, with the results shown in Figure 4-27. The advantage of a 32-byte or 64-byte cache-line size is that there are more entries in a given cache for a fixed cache size, which results in reduced conflicts. Larger caches, on the other hand, allow for larger bulk transfer of contiguous data, which can improve performance, especially for vector processing. However, when a larger cache-line size is used, proportionally larger caches need to be used as well, or else the cache-conflict rate will suffer, as there are less entries in the cache. The `LACache`, data-cache, and instruction-cache are all sufficiently sized for a 128-byte cache-line size according to the results presented in (Patterson D. A., 2013). An interesting decision was made by CRAY X1 designers to use a small cache-line size of 32-bytes, with the justification that smaller cache-line sizes are optimized for applications with low spatial locality (CRAY,

2016). Parameter sweeps with the *LACore* in gem5 showed adverse performance when using a 32-byte or 64-Byte cache line size compared to a 128-byte cache-line size for DGEMM, as shown in Figure 4-27.

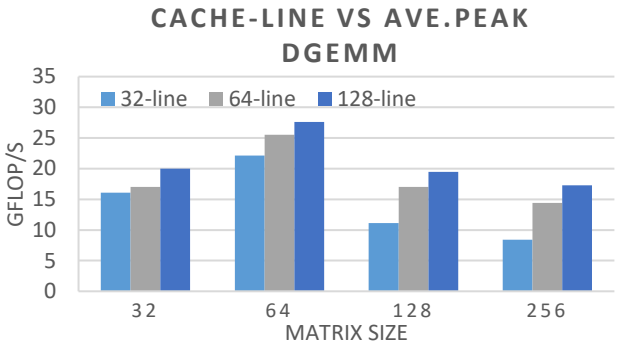

Figure 4-27: Cache Line Size vs DGEMM performance.

5 LACORE INSTRUCTION SET

5.1 LACORE ISA OVERVIEW

The full *LACore* extension to the RISC-V Instruction Set (Waterman A. a., 2016) is shown in Figure 5-1. A total of 68 instructions have been added using the Custom-0 extension space. RISC-V was chosen as a vehicle for the *LACore* architecture as opposed to other ISAs such as MIPS, ARM or x86, for a few reasons:

1) RISC-V is a modern, free, and open-source platform, with robust toolchains for developing both hardware and software;
2) a gem5 implementation and fully tested RISC-V RTL models exist for many CPU types, through the Sodor collection and BOOM (Celio C. P., 2015), allowing the *LACore* to only be implemented once in gem5 and in RTL to be used in these various models; and
3) RISC-V provides a light-weight scalar CPU footprint, which makes more sense than a heavyweight superscalar CPU as the *LACore* scales to manycore chip designs.

RISC-V LACORE ISA Extension

	inst	args	31 27	26 24	23 21	20 18	17	16 15	14 12	11 9	8 7	6 0
EXEC INSTS			LRXA	LRA	LRB	LRC	MST	*RDCT* LFUNC2	LRD	*DV SU T2* LFUNC3	*LOP*	OPCODE
	ldr{m|d}{a|s}{b|t}{n|x|s}	lrd, lra, lrb, lrc, lrxa	lrxa	lra	lrb	lrc	0	{00|01|10}	lrd	{000-111}	00	0001011
	ldrm{m|d}{a|s}{b|t}{n|x|s}	lrd, lra, lrb, lrc, lrxa	lrxa	lra	lrb	lrc	1	{00|01|10}	lrd	{000-111}	00	0001011
	ldv{m|d}{a|s}{b|t}	lrd, lra, lrb, lrc, lrxa	lrxa	lra	lrb	lrc	0	0	lrd	{000-111}	01	0001011
CFG INSTS			LRFA/ LRXA	LRXB	LRXC			*SPV TNS* *LRDLOC* LFUNC2	LRD	*DP VEC ALT* LFUNC3	LOP	OPCODE
	lcfgsx{s|d}	{lrd}, lrxa	lrxa	0	0			{00|01|10}	lrd	{0|1}00	10	0001011
	lcfgsf{s|d}	lrd, lrfa	lrfa	0	0			0	lrd	{0|1}01	10	0001011
	lcfgvadrv	lrd, lrxa	lrxa	0	0			0	lrd	010	10	0001011
	lcfgvadr{s|t}	lrd, lrxa, lrxb, lrxc	lrxa	lrxb	lrxc			1{0|1}	lrd	010	10	0001011
	lcfgvlay{s|d}	{lrd}, lrxa, lrxb, lrxc	lrxa	lrxb	lrxc			{00|01|10}	lrd	{0|1}11	10	0001011
XFER INSTS			LRXA	LRA	LZERO9				LRD	*CLR GET DAT* LFUNC3	LOP	OPCODE
	lxferdata	lrd, lra, lrxa	lrxa	lra	0				lrd	001	11	0001011
	lxfercsrget	lrxa	lrxa	0	0				0	010	11	0001011
	lxfercsrclear		0	0	0				0	100	11	0001011

Figure 5-1: LACore Instruction Set Overview.

5.1.1 Instruction Opcode

The *OpCode* field for the RISC-V ISA extension is 0001011 , and is composed of two parts – the Minor OpCode covering bits [6:5], and the Major Opcode covering bits [4:2], both of which are specified in the RISC-V ISA Manual (Waterman A. a., 2016). Bits [1:0] specify the instruction size, where *11* means 32-bit instruction. Bits [4:2] and Bits [6:5] together form a matrix of op-classes. As seen in Figure 5-2, there are four op-class spots in the 32-bit ISA space that are dedicated to custom ISA extensions.

LACore occupies the custom-0 extension slot, which means the Minor OpCode is 00 and the Major Opcode is 010. Note that this gives the *LACore* ISA a 25-bit encoding space.

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

Figure 5-2: RISC-V Instruction Set Map, Table 9-1 in (Waterman A. a., 2016).

5.1.2 Instruction Size

A design decision was made whether to use a 32-bit instruction space, a larger, 48-bit or 64-bit instruction space, or a variable length instruction space. Since the *LACore* ISA is not a typical load-store ISA, it has some unique instruction requirements that need to be taken into consideration when choosing a 32-bit, 64-bit, or variable instruction length:

- **Instruction Width Vs Number of Instructions:** There is a lot of configuration that needs to happen before a *ExecInst* or *XferInst* can start reading and writing data from memory correctly. The *LACore* ISA could either use wide instructions to fit all the operands, or use multiple consecutive, smaller instructions to accomplish the same task. The former option requires less instructions but complicates the decoder by requiring instruction fetch buffering (Patterson D. A., 1985), while the latter option has higher latency but is simpler and fits in the 32-bit RISC-V ISA space nicely.
- **Operand Layout Parameters:** Each input and output vector can be configured as a *Scalar*, *Vector* or *Sparse* data type. The *Scalar* type is simply a 32-bit or 64-bit floating point, so does not require many configuration parameters. The *Vector* data type requires more parameters in addition to whether it is a vector of 32-bit or 64-bit floating point values, such as the start address of the vector, and its stride, count and skip. This is a total of three 5-bit register indexes and a Boolean flag, so 16 bits are used. Since the *ExecInst* instructions use three inputs and one output, this would require 64 bits just for operand configuration, which would require an instruction size even larger than 64-bits. The problem becomes much worse with the *Sparse* data type, which requires six parameters per operand (and would require over 128-bit instruction).
- **Operand Locations:** The *LACore* is mainly used as a memory-memory ISA via the *ExecInsts* and *XferInsts*, but the data-source can also be in the *LACfg* configuration registers as well.

In addition to data stored in memory, there is actually two disjoint address spaces: Private Scratchpad Memory, and Main Memory. So we need the ISA to specify a location of the input and output data of the three possible locations in addition to the already large amount of configuration needed for the Layout Parameters discussed above.

The above discussion support why a multi-instruction configuration sequence is inevitable for the *LACore* ISA. It might be feasible to mix in 128-bit instructions with 32-bit instructions, but the decoding logic becomes more complicated and requires buffering, and the real cost of configuration just moves to filling a bunch of general purpose registers over many cycles, instead of configuring the *LACfgs* over a similar number of cycles.

5.2 CONFIGURATION INSTRUCTIONS

	inst	args	31 27	26 24	23 21	20 18	17	16 15	14 12	11 9	8 7	6 0
			LRFA/ LRXA	LRXB		LRXC		SPV TNS LRDLOC LFUNC2	LRD	DP VEC ALT LFUNC3	LOP	OPCODE
CFG INSTS	lcfgsx{s|d}	{lrd}, lrxa	lrxa	0		0		{00|01|10}	lrd	{0|1}00	10	0001011
	lcfgsf{s|d}	lrd, lrfa	lrfa	0		0		0	lrd	{0|1}01	10	0001011
	lcfgvadrv	lrd, lrxa	lrxa	0		0		0	lrd	010	10	0001011
	lcfgvadr{s|t}	lrd, lrxa, lrxb, lrxc	lrxa	lrxb		lrxc		1{0|1}	lrd	010	10	0001011
	lcfgvlay{s|d}	{lrd}, lrxa, lrxb, lrxc	lrxa	lrxb		lrxc		{00|01|10}	lrd	{0|1}11	10	0001011

Figure 5-3: LACore configuration ISA.

The 9 configuration instructions, illustrated in Figure 5-3, are called *CfgInsts*, and are single-cycle instructions that update a *LACfg* register specified by *lrd* with contents from general-purpose integer or float registers *lrxa*, *lrxb*, *lrxc*, and *lrfa*. In the RISC-V scalar CPU implementation of the *LACore* ISA, *lrxa*, *lrxb* and *lrxc* corresponds to registers x0-x31, and *lrfa* corresponds to registers f0-f31 from (Waterman A. a., 2016). The description and usage of each argument is presented in Table 5-1 below.

In addition to the instruction arguments, there are five configuration bitfields in *CfgInsts*: SPV, TNS, DP, VEC and ALT. The SPV bit is 1 if the data-source is a sparse matrix, and is only asserted during the *lcfgvadr{s|t}* instructions. The TNS field is 1 if the sparse matrix should be read transposed, and is only relevant for the *lcfgvadr{s|t}* instructions. The DP field is 0 to configure a single-precision data-source, and 1 to configure a double-precision data-source, and this flag applies to the *lcfgsx{s|d}*, *lcfgf{s|d}* and *lcfgvlay{s|d}* instructions, since the *lcfgvadrv* and *lcfgvlay{s|d}* do not configure the data precision. The VEC field is 1 if the data-source is a vector or sparse matrix, and 0 if it is a scalar; it applies to every configuration instruction. The ALT field, or

"alternate configuration" field, specifies that a scalar should be read from the general-purpose floating point registers instead of read from scratchpad or memory, OR that the *lcfgvlay{s|d}* instruction is run to update the layout fields for a vector or sparse matrix. The description and usage of each bitfield is also presented in Table 5-1 below.

Field	Field Type	Description	Used in Instructions
LRD	Argument	Destination LACfg register lr0-lr7	ALL
LRXA	Argument	Source RISC-V integer register x0-x31	
LRXB	Argument	Source RISC-V integer register x0-x31	*lcfgvadr{s|t}* *lcfgvlay{s|d}*
LRXC	Argument	Source RISC-V integer register x0-x31	
LRFA	Argument	Source RISC-V floating point register f0-f31	*lcfgsf{s|d}*
SPV	Bitfield	1 if sparse, 0 if not sparse	*lcfgvadrv* *lcfgvadr{s|t}*
TNS	Bitfield	1 if sparse is transposed, 0 otherwise	
LRDLOC	Argument	0x00, 0x01, 0x10 for *lrd*, *(lrd)*, *{lrd}*, meaning data-source is in LACfg, memory, or scratchpad respectively	*lcfgsx{s|d}* *lcfgsf{s|d}* *lcfgvlay{s|d}*
DP	Bitfield	1 if data-source is double-precision, 0 otherwise	ALL
VEC	Bitfield	1 if data-source is vector or sparse matrix, 0 otherwise	
ALT	Bitfield	1 if data-source is scalar and in float register or performing the lcfgvlay{s|d} instruction, 0 otherwise	
LOP	Bitfield	LACore sub-opcode, 0x10 for config instructions	
OPCODE	Bitfield	LACore opcode, 0x0001011 for config instructions	

Table 5-1: Configuration Instructions bitfields and arguments.

A subset of the scalar and vector configuration instructions are illustrated in Figure 5-4. Curly braces or parenthesis around *lrd* mean the data stream is in the scratchpad or memory, respectively; no braces or parenthesis mean the data stream is a scalar floating point value stored in the *LACfg* itself. Additionally, all scalars, vectors or sparse stream configurations can be single or double-precision. Configuring a scalar is done by either *lcfgsx{s|d}* to load an address into the *LACfg*, and *lcfgsf{s|d}* to load the floating-point value into the *LACfg*. Configuring a vector or sparse matrix takes two instructions: one to configure the addresses, *lcfgvadrv* or *lcfgvadr{s|t}*, and one to configure the layout of the vector or sparse matrix, *lcfgvlay{s|d}*. Each configured *LACfg* can be used by one or multiple *LAMemUnits* simultaneously during a *ExecInst* or *XferInst*.

```
// put x2 addr of 32 - bit float in scratch in LACfg 0.
// put x3 addr of 64 - bit float in mem in LACfg 1.
// put 32 - bit float value in f3 in LACfg 2.
lcfgsxs {lr0}, x2
lcfgsxd (lr1), x3
lcfgsfs lr2, f3

// put addr of 64 - bit float in mem into LACfg 2,
// then put its stride/count/skip into LACfg 2
lcfgvadrv lr2, x3
lcfgvlayd (lr2), x4, x5, x6
```

Figure 5-4: LACore configuration instruction examples.

An important note is that the scratchpad and memory address spaces are non-unified, so the address 0x10 in the scratchpad refers to byte offset 0x10 in the scratchpad, while memory addresses refer to normal memory address space. This means that scratchpad data is not directly accessible to the programmer, and must be explicitly transferred to memory through either a data-transfer or execution instruction in order to be used by the scalar CPU instructions.

5.3 Data Transfer Instructions

	inst	args	31 27	26 24	23 21	20 18	17 16	15 14 12	11 9	8 7	6 0
			LRXA	LRA	LZERO9			LRD	CLR GET DAT / LFUNC3	LOP	OPCODE
XFER INSTS	lxferdata	lrd, lra, lrxa	lrxa	lra	0			lrd	001	11	0001011
	lxfercsrget	lrxa	lrxa	0	0			0	010	11	0001011
	lxfercsrclear		0	0	0			0	100	11	0001011

Figure 5-5: LACore Data Transfer ISA.

Field	Field Type	Description	Used in Instructions
LRD	Argument	Destination LACfg register lr0-lr7	*lxferdata*
LRA	Argument	Source LACfg register lr0-lr7	*lxferdata*
LRXA	Argument	RISC-V integer register x0-x31, holds the 'count' for *lxferdata* operation and serves as the destination register for the *lxfercsrget* instruction	*lxferdata* *lxfercsrget*
CLR	Bitfield	1 if clearing the LACsrReg, 0 otherwise	ALL
GET	Bitfield	1 if copying LACsrReg to an integer register, 0 otherwise	
DAT	Bitfield	1 if performing a data transfer from a source LACfg to a destination LACfg	
LOP	Bitfield	LACore sub-opcode, 0x11 for all data moving instructions	
OPCODE	Bitfield	LACore opcode, 0x0001011 for config instructions	

Table 5-2: Data Movement Instructions bitfields and arguments.

The three *XferInsts*, shown in Figure 5-5, copy data-streams between any combination of memory, scratchpad and an *LACfg*. The bitfields and arguments for these instructions are shown in Table 5-2 above. The *lxferdata* instruction moves data from an arbitrary source specified by the *LACfgs*: *lrd* (destination) and *lra* (source). The total number of elements to copy is specified by general-purpose integer register *lrxa*. The *lxfercsrget* instruction copies the 64-bit *LACsrReg* (the control-status register) contents into the general-purpose integer register *lrxa*. The *lxfercsrclear* instruction sets the *LACsrReg* contents to zero. Figure 5-6 illustrates a subset of the *lxferdata* assembly instructions.

```
// transfer the scalar/vector/matrix in LAReg 1
// to the scalar/vector/matrix in LAReg 0.
// The number of elements to transfer is
// in integer register x2.
laxferdata lr0, lr1, x2
```

Figure 5-6: LACore Data Movement Instruction examples.

The *lxferdata* functionality is similar to the memcpy() extension to the RISC-V ISA proposed by (Mao, 2016), except the *LACore* version is much more flexible by providing mechanisms to:

1) convert between single-precision and double-precision streams;
2) transform streams between scalars, vectors, and sparse matrices; and
3) move data between scratchpad and memory.

5.4 Data Execution Instructions

	inst	args	31 27	26 24	23 21	20 18	17	16 15	14 12	11 9	8 7	6 0
			LRXA	LRA	LRB	LRC	MST	*RDCT* / LFUNC2	LRD	*DV* *SU* *T2* / LFUNC3	*LOP*	OPCODE
EXEC INSTS	ldr{m|d}{a|s}{b|t}{n|x|s}	lrd, lra, lrb, lrc, lrxa	lrxa	lra	lrb	lrc	0	{00|01|10}	lrd	{000-111}	00	0001011
	ldrm{m|d}{a|s}{b|t}{n|x|s}	lrd, lra, lrb, lrc, lrxa	lrxa	lra	lrb	lrc	1	{00|01|10}	lrd	{000-111}	00	0001011
	ldv{m|d}{a|s}{b|t}	lrd, lra, lrb, lrc, lrxa	lrxa	lra	lrb	lrc	0	0	lrd	{000-111}	01	0001011

Figure 5-7: LACore Data Execution ISA.

The 56 *ExecInsts*, illustrated in Figure 5-7, simultaneously configure the *LAMemUnits* to stream input and output data into the *LAExecUnit*'s datapath, and configure the datapath to perform one of the 24 possible operations on the data streams.

ExecInsts come in three output-modes: vector-output, scalar-output, and multi-stream output, which have instruction prefixes of *ldv*, *ldr*, and *ldrm* respectively. There are 8 variations of *ldv* instructions, and 24 variations of *ldr* and *ldrm* instructions each. The variations, seen in the inst column of Figure 5-7, are used to configure the various muxes and control circuitry in the *LAExecUnit*

for the duration of the instruction. The bitfields names and their functions are described in more detail in Table 5-3 below. The three input data-streams and one output data-stream are specified by instruction args *lra*, *lrb*, *lrc*, and *lrd* respectively, while the total number of input elements to operate on is specified by integer register *lrxa*. Note that the total number of output elements is dependent on whether a vector-output, scalar-output or multi-stream output instruction is selected, and not directly dependent on the value in *lrxa*. Figure 5-8 below shows a subset of *ExecInsts* for each of the three types of output-modes.

Field	Field Type	Description	Used in Instructions
LRD	Argument	Destination LACfg register lr0-lr7	ALL
LRA	Argument	Source A LACfg register lr0-lr7	
LRB	Argument	Source B LACfg register lr0-lr7	
LRC	Argument	Source C LACfg register lr0-lr7	
LRXA	Argument	RISC-V integer register x0-x31, holds the 'count', or number of input elements from each stream to process	
MST	Bitfield	1 if multi-stream output, 0 otherwise	ALL
RDCT	Bitfield	0x00, 0x01, 0x10 for min, max, sum for ReduceNode config	24 *ldr** instructions 24 *ldrm** instruction
DV	Bitfield	1 if divide, 0 if mutliply for VecNode configuration	ALL
SU	Bitfield	1 if subtract, 0 if add for VecNode configuration	
T2	Bitfield	1 if (A+-B)*/C and 0 if (A*/B)+-C for VecNode config	
LOP	Bitfield	LACore sub-opcode, 0x00 for scalar and multi-stream outputs, and 0x01 for vector outputs	
OPCODE	Bitfield	LACore opcode, 0x0001011 for config instructions	

Table 5-3: Execution Instructions bitfields and arguments.

The *VecNodes* in the datapath are configured by the LFUNC3 field in the *ExecInst*, while the *ReduceNodes* are configured by the LFUNC2 field. The RDCT field within the LFUNC2 field is 0x00 if the *ReduceNodes* perform sum(X, Y); RDCT is 0x01 if the *ReduceNodes* perform max(X, Y); RDCT is 0x10 if the *ReduceNodes* perform min(X, Y). the MST flag is 1 for multi-stream output only.

```
// scalar-output: output len is 1
// lr0 = sum( (lr1+lr2)*lr3 )
// lr0 = sum( (lr1*lr2)-lr3 )
ldrmats lr0, lr1, lr2, lr3, x2
ldrmsbs lr0, lr1, lr2, lr3, x2
// multi-stream-output: output length determined by
// x2 value and count field in LARegs 1 ,2 and 3
// lr0 = sum( (lr1+lr2)*lr3 )
// lr0 = sum( (lr1*lr2)-lr3 )
ldrmmats lr0, lr1, lr2, lr3, x2
ldrmmsbs lr0, lr1, lr2, lr3, x2
// vector-output: output length in x2
// lr0 = (lr1+lr2)/lr3
// lr0 = (lr1/lr2)-lr3
ldvdat lr0, lr1, lr2, lr3, x2
ldvdsb lr0, lr1, lr2, lr3, x2
```

Figure 5-8: LACore Data Execution Instruction examples.

The *VecNode* configuration flags within the LFUNC3 field are DV, SU and T2. DV is 1 for division and 0 for multiplication, SU is 1 for subtract and 0 for add, and T2, which stands for "multiply/divide first or second", is 1 if the multiply/divide operation should be used on the A and B input or on the (A+-B) and C inputs. Originally, the T2 field stood for "top or bottom", since the original *VecNode* operations were those in Table 5-4. However, as the *LACore* ISA evolved, the eight *VecNode* operations in Table 5-4 were replaced by those in Table 4-1.

Deprecated *VecNode* Ops	
(A+B)*C	(A+B)/C
(A-B)*C	(A-B)/C
A*(B+C)	A/(B+C)
A*(B-C)	A/(B-C)

Table 5-4: Deprecated VecNode operations.

6 LACoreAPI Framework

The *LACore*'s ISA was implemented in gcc, and a C programming, header-only library called the *LACoreAPI* was developed to raise the abstraction of programming the *LACore* from the assembly level to C. All benchmarks and programs targeting the *LACore* thus far have been built on top of the *LACoreAPI*. The API mimics the underlying assembly instructions fairly closely, providing configuration, data movement and execution function calls.

6.1.1 Configuration API

The configuration API is summarized in Figure 6-1. Lines 5-7 show how scalar data-streams are configured for any of the *LACfg*, scratchpad or memory locations. Lines 10-11 show a subset of the vector configuration API calls. There are many variations for setting the vector stride, count and skip, or using sane default values if only the address or the address and sub-vector count need to be specified.

```
double *value;
LAAddr addr = (LAAddr)value;

// set scalar in an LACfg, scratch, or mem
la_set_scalar_dp_reg(0, *value);
la_set_scalar_dp_sch(1, 0x10);
la_set_scalar_dp_mem(2, addr);

// set vector in mem or scratch
la_set_vec_dp_mem(0, addr, stride, count, skip);
la_set_vec_adr_dp_mem(0, addr);
la_set_vec_adrcnt_dp_sch(0, addr, count);
```

Figure 6-1: LACoreAPI configuration API examples.

6.1.2 Data Movement API

The *LACoreAPI* provides a simple API for moving data-streams between memory, scratchpad and the *LACfgs* through the *la_copy()* API call, illustrated in Figure 6-2, which takes a source and destination LACfg, and the number of elements in the streams to transfer. It blocks for an arbitrary number of cycles until the transfer has completed, and therefore is not an asynchronous interface call. It should be apparent from Figure 6-2 that much more sophisticated data movements can be performed, since the *LACfgs* can be configured for scratchpad or memory locations, with scalar, vector or sparse matrix configs, with either single or double-precision floating point elements.

```
// copies src->dst, convert double->float
void copy_d2s(double *src , float *dst , int cnt)
{
  la_vec_adr_dp_mem(0, (LAAddr)src);
  la_vec_adr_sp_mem(1, (LAAddr)dst);
  la_copy(1, 0, cnt);
}
```

Figure 6-2:LACoreAPI data movement API example.

6.1.3 Execution API

The *LACoreAPI* provides API calls for all variations of vector-output, scalar-output, and multi-stream output operations for data-streams of arbitrary length. Each API call takes 5 arguments: the 3 source *LACfg* indexes, the 1 destination *LACfg* index, and the number of input elements to operate on. Vector-output calls have the format *la_A<op>B<op>C(...)*, scalar-output calls have the format *la_A<op>B<op>C_<op2>(...)*, and multi-stream-output instructions have the format *la_A<op>B<op>C_<op2>_multi(...)*. In these formats, <op> is either add, sub, mul or div, which set the *LAExecUnit* datapath's *VecNode* configuration, while <op2> is sum, min or max, which sets the *LAExecUnit* datapath's *ReduceNode* configuration. Examples of the three variations of execution API calls are shown in Figure 6-3.

```
//D = (A+B)*C, vector output
la_AaddBmulC(D, A, B, C, count);

//D = (A/B)-C, scalar output
la_AdivBsubC_sum(D, A, B, C, count);

//D = (A*B)+C, multi - stream output
la_AmulBaddC_sum_multi(D, A, B, C, count);
```

Figure 6-3: LACoreAPI execution API examples.

7 BENCHMARKS AND EVALUATION

7.1 BENCHMARK METHODOLOGY

HPC Challenge benchmark suite (Luszczek, 2005) and the Sparse DGEMV routine were used to compare the performance of the *LACore* against three different architectures: a simple in-order pipelined RISC-V processor, a superscalar x86 processor with SSE2 enabled, and an NVIDIA Fermi GPU with two Streaming Multiprocessors (SMs). All platforms were simulated on gem5 (Binkert, 2011) in syscall-emulation mode, with the Fermi GPU simulated with gem5-gpu (J. Power, 2015).

The HPCC benchmark suite was selected to evaluate our platform primarily because it is designed for high performance linear algebra computations. It contains seven benchmarks that stress computation throughput, memory bandwidth, and communication bandwidth. Additionally, its benchmarks test performance with high-and-low spatial-and-temporal localities. We believe these reasons make the HPCC benchmark suite a strong candidate to holistically evaluate our architecture across a range of linear algebra applications. In addition to the HPCC benchmark suite, Sparse DGEMV, which stands for *Double-Precision Matrix Vector Multiplication*, was also used to evaluate the sparse-matrix manipulation capabilities of the *LACore* compared to the other platforms. Sparse DGEMV was chosen to evaluate the sparse capabilities because it is a common linear algebra routine with a high arithmetic complexity.

The main motivation for using a cycle-accurate simulator for benchmarking the x86 and GPU platforms instead of using real hardware is to provide a fairer comparison by removing process technology, clock speeds and cache configuration as variables, in order to focus on the merits of the processor architectures. gem5 provides the capability of setting the comparable processor frequencies and cache sizes, as well as control the workloads more directly than running them on top of an operating system.

Only a single thread on a single core was used in the *LACore*, RISC-V and x86 implementations, even though multithreading and multi-core were options. The reasoning is to compare the raw performance of a single hardware thread in each of these three platforms. Similarly, the GPU was run using only a single cluster, with two cores per cluster. Here, cluster is gpgpu-sim's equivalent of a collection of NVIDIA Streaming Multiprocessors (A. Bakhoda, 2009). Only two NVIDIA SMs were modelled since this is a fair comparison to an *LACore* in terms of area, as discussed in the Design Area

Estimates section. The most relevant gem5 configurations for the LACore, RISC-V, x86 and scaled-down Fermi GPU platforms are shown in Table 7-1.

LACore		**X86, RISC-V**		**CUDA**	
Scalar CPU Model	In-Order	X86 CPU Model	Out-of-Order	Scalar X86 CPU Model	Out-of-Order
Scalar CPU Clock	3 GHz	RISC-V CPU Model	In-Order	GPU Model	NVIDIA Fermi
LAExecUnit Clock	1 GHz	Scalar CPU Clock	3 GHz	Scalar CPU Clock	3 GHz
VecNode Count	8	Cache Line Size	128 Bytes	GPU Core Clock	1 GHz
SIMD Width	8	Inst-Cache Size	16 KB	Cache Line Size	128 Bytes
Scratchpad Size	64 KB	Data-Cache Size	64 KB	GPU Clusters	1
Inst-Cache Size	16 KB	L2-Cache Size	256 KB	GPU Cores-per-Cluster	2
Data-Cache Size	64 KB			L2-Cache Size	1 MB
LACache Size	64 KB				
L2-Cache Size	256 KB				

Table 7-1: gem5 and gem5-gpu configurations used for the HPCC benchmark suite.

The HPCC benchmarks used for evaluation were DGEMM, FFT, PTRANS, HPL, Random Access, and STREAM. This covers the four extreme combinations of high-and-low spatial-and-temporal locality. b_eff was not tested since it primarily evaluates multi-processor communication efficiency, which is not the focus of this thesis.

The standard HPCC software distribution was not used for the benchmarks. Instead, kernels tailored for each of the platforms were either hand-written, or used high-performance libraries, such as GNU Scientific Library, FFTW3, and Eigen (Gough, 2009) (Johnson, 1998) (G. Guennebaud, 2010). There are a few reasons for not using the standard distribution of HPCC:

1) gem5's syscall-implementation mode does not cover all syscalls, or all functionality of every ISA it implements, so many of the compiled x86 benchmarks do not even run
2) *LACore* and CUDA (NVIDIA's software framework) are special platforms that don't have specially tailored kernels in the HPCC distribution, so they need to be written anyways
3) this paper evaluates single-threaded functionality for x86, RISC-V and *LACore*, so greater control is needed over exactly how the benchmarks are internally running

Care was taken to follow the specification for each benchmark described in (Luszczek, 2005) as closely as possible, in order to provide reproducible results. For all benchmarks, we verified the *LACore* results to be correct against correct implementations.

7.2 DGEMM

7.2.1 Implementation

Double-Precision Dense Matrix-Matrix multiply is at the heart of many Linear-Algebra applications, and is a high spatial-and-temporal locality application. The operation count approaches $2n^3$ for square matrix sizes of n. For this paper, all 4 variations of $C = \alpha\ op(A)op(B) + \beta\ C$ were evaluated on all platforms.

The RISC-V platform used the GNU Scientific Library (GSL) implementation of DGEMM (Gough, 2009), while the x86 version used both the GSL and Eigen library implementations, with SSE2 enabled (G. Guennebaud, 2010). The Fermi GPU ran optimized hand-written kernels since cuBLAS was not available as a static library for NVIDIA Toolkit 3.2 and no good implementations for all four DGEMM variations that could run without runtime errors on gem5-gpu could be found. The `LACore` implementation used a handwritten algorithm leveraging the `LACoreAPI` framework.

The `LACore`'s algorithm for solving DGEMM uses a standard paneling technique, where sub-blocks of matrices A, B and C are copied into the scratchpad, and then several dot-products are performed for each sub-row in A with a sub-column in B and then updates the element in C. After all the dot-products using the `LAExecUnit`'s custom datapath, the block from C is copied back into main memory. The reason paneling is effective is because it improves temporal and spatial locality on average, which is important in cache-based memory hierarchies, which the `LACore` has.

When transposing A or B is required, as is for three of the four DGEMM variations, the `LACore`'s algorithm will first transpose the whole matrix before performing the paneled-DGEMM sub-algorithm discussed above. The reason that the whole matrix is transposed beforehand is to allow the highly-optimized paneled-DGEMM kernel to remain the same across all four variations, therefore requiring the conversion of all inputs to a common format beforehand. Figure 7-1 shows the C-language transpose sub-routine used in the `LACore`'s DGEMM kernel, where blocks are copied from memory to scratchpad to memory using the special `LAMemUnit` hardware through the `la_copy()` API call in the `LACoreAPI`.

```
//transposes an NxM matrix in src to an MxN matrix in dst
void dtranspose_la_core(double *src, double *dst, IDX N, IDX M) {
  const LACfgIdx src_mem_REG = 0, src_sch_REG = 1;
  const LACfgIdx dst_sch_REG = 0, dst_mem_REG = 1;
  IDX BLOCK = (IDX)(log2((double)SCRATCH_SIZE)/(double)sizeof(double));

  //perform transpose in panels of size (icount x jcount)
  for(IDX i=0; i<N; i+=BLOCK) {
    for(IDX j=0; j<M; j+=BLOCK) {
      IDX icount = MIN(BLOCK, (N - i));
      IDX jcount = MIN(BLOCK, (M - j));

      LAAddr src_addr = (LAAddr)(src + (i+N*j));
      LAAddr dst_addr = (LAAddr)(dst + (j+M*i));

      //configure LACfg with index 'src_mem_REG' for
      //source submatrix in memory
      la_set_vec_dp_mem(src_mem_REG, src_addr, 1, icount, (N-icount));
      //configure LACfg with index 'src_sch_REG' for
      //source submatrix in the scratchpad
      la_set_vec_dp_sch(src_sch_REG, 0, jcount,
                        icount, 1-(jcount*icount));
      //transfer source data from mem to scratchpad
      la_copy(src_sch_REG, src_mem_REG, icount*jcount);

      //configure LACfg with index 'dst_sch_REG' for
      //the submatrix we just transferred to the scratchpad
      la_set_vec_dp_sch(dst_sch_REG, 0, 1, jcount, 0);
      //configure LACfg with index 'dst_mem_REG' for
      //the submatrix within the dst matrix
      la_set_vec_dp_mem(dst_mem_REG, dst_addr, 1, jcount, (M-jcount));
      //transfer dst data from scratchpad to mem
      la_copy(dst_mem_REG, dst_sch_REG, icount*jcount);
    }
  }
}
```

Figure 7-1: DGEMM matrix transpose sub-routine in C for the LACore.

7.2.2 Results

The averaged performances of all four DGEMM variations are shown in Figure 7-2. The *LACore*'s performance asymptotically increases with the matrix size. At a matrix dimension of 128, the *LACore* achieves an average 14.3 GFLOP/s, which is a speedup of 31.7x, 1.6x, 9.8x and 5.73x over RISC-V, Fermi GPU, x86-GSL and x86-Eigen implementations. At a dimension of 1024, the average speedup is 80.4x, 1.52x, 47.3x and 6.7x respectively, which can be seen in Figure 7-3.

The large speedup over the RISC-V and x86 implementations is expected since DGEMM is a highly computation-bound application, and the *LACore* can keep its *LAExecUnit* constantly busy since memory accesses are regularly-strided with high localities. Also, the vector sizes that are accessed depend primarily on the large panel width in the paneling DGEMM algorithm, as opposed to HPL

(discussed later), which accesses vectors of size $1, 2, 3, \ldots, n$ when performing LU decomposition and forward/back substitution. This is why DGEMM performs better than HPL on the *LACore*.

The performance speedup over the Fermi GPU can be explained by the *LACore* not having to transfer data to and from device memory, while still benefiting from similar parallelism that the GPU provides. Additionally, the *LACore* does not require synchronization during the reduction phase of a dot-product, since only a single thread is running, and the *LAExecUnit* has a special reduction unit in it. The Fermi GPU, on the other hand, requires calls to syncthreads() after each iteration of a dot-product reduction accessing shared-memory, which causes additional overhead.

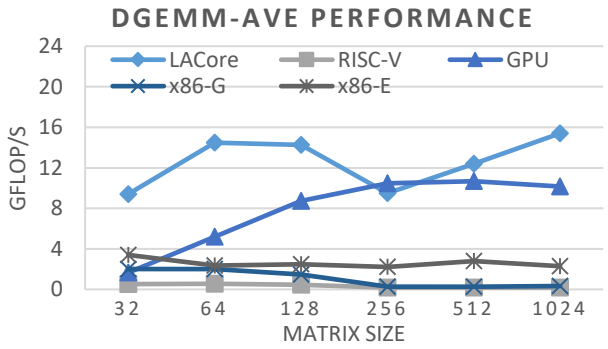

Figure 7-2: DGEMM average for LACore, RISC-V, Fermi GPU, x86-GSL and x86-Eigen.

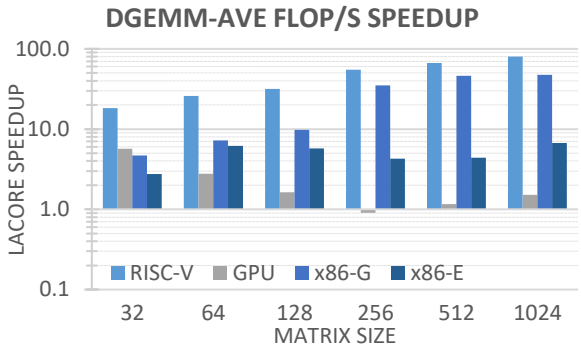

Figure 7-3: DGEMM average speedup of LACore over RISC-V, Fermi GPU, x86-GSL and x86-Eigen.

Similar trends can be seen for each platform's peak performance across all four DGEMM variations, with the *LACore* performing even better than in the DGEMM-average case. Figure 7-4 shows that the *LACore* achieves a peak of 30.5 GFLOP/s at a matrix dimension of 64, which is a speedup of 50x, 4.8x, 11.7x and 12.2x over RISC-V, Fermi GPU, x86-GSL and x86-Eigen implementations, as seen in Figure 7-5. The *LACore*'s peak performance also outperforms all other platforms asymptotically as the matrix size increases, with a performance of 23.8 GFLOP/s at a matrix dimension of 1024, and a speedup of 86x, 1.5x, 43x and 10x over RISC-V, Fermi GPU, x86-GSL and x86-Eigen implementations.

The reason the *LACore* has larger speedups in the peak DGEMM results vs the averaged DGEMM results is because the matrix transposes required in three of the four DGEMM variations cannot be performed as efficiently by the *LACore* as simple matrix-matrix multiplication. So the one DGEMM variation that does not have any transposes has a much better performance.

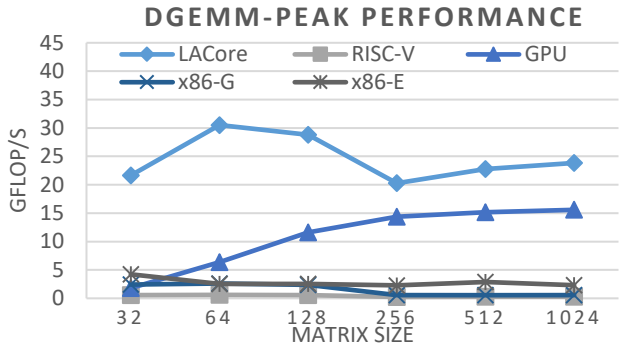

Figure 7-4: DGEMM peak for LACore, RISC-V, Fermi GPU, x86-GSL and x86-Eigen.

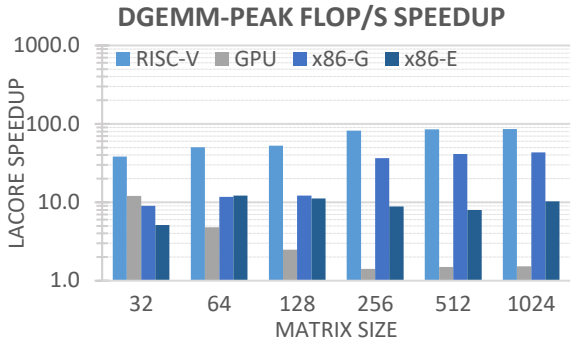

Figure 7-5: DGEMM peak speedup of LACore over RISC-V, Fermi GPU, x86-GSL and x86-Eigen.

The worst-case DGEMM performance across all four DGEMM variations has the *LACore* performing worse than the Fermi GPU implementation at intermediate matrix dimensions, but performing better than all the other platforms at small and large matrix dimensions. Figure 7-6 shows the *LACore* achieves its best worst-case performance at the largest matrix size tested of 1024, where it ran at 9.9 GFLOP/s, with a speedup of 455x, 2.8x, 47x and 4.3x over the RISC-V, Fermi GPU, x86-GSL and x86-Eigen implementations. As seen in Figure 7-7, the x86 and Fermi GPU implementations are fairly competitive with the *LACore*'s performance, at low-to-mid matrix sizes.

The poorer performance in the worst-case of the *LACore* compared to the other platforms is most likely due to the matrices being small enough to fit in the L1 caches, thus causing the *LACore*'s extra transpose sub-routine overhead to not be worth the time investment. If the matrices are small enough to fit into the caches, then the other implementations don't suffer any real spatial or temporal locality penalties when working with transposed matrices, but at larger matrices, this penalty becomes severe, and the *LACore*'s transpose sub-routine pays off tremendously, as seen in Figure 7-6.

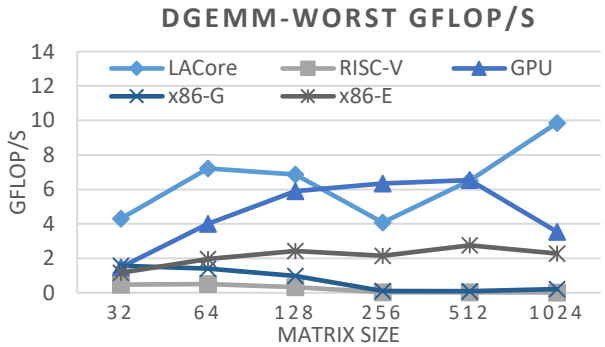

Figure 7-6: DGEMM worst-case for LACore, RISC-V, Fermi GPU, x86-GSL and x86-Eigen.

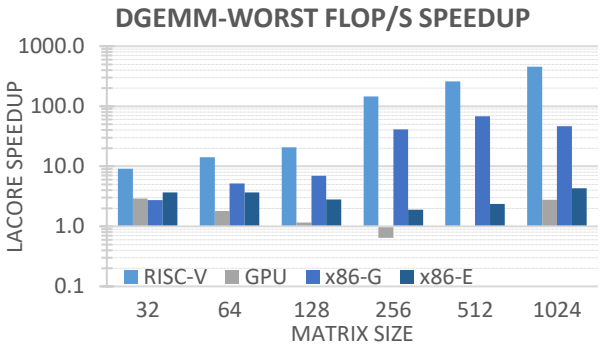

Figure 7-7: DGEMM worst-case speedup of LACore over RISC-V, Fermi GPU, x86-GSL and x86-Eigen

7.3 FFT

7.3.1 Implementation

The complex double-precision FFT tests computational throughput for low spatial-locality, high temporal-locality applications. The total operation count is $5m\ log2m$ and is measured in GFLOP/s (Luszczek, 2005). The RISC-V CPU used GSL for FFT (Gough, 2009). The x86 CPU used two different libraries, GSL and FFTW (Johnson, 1998), both compiled with SSE2 enabled. Since cuFFT was not available as a static library for NVIDIA Toolkit 3.2, a slightly modified FFT implementation was taken from the University of Illinois' Parboil benchmark suite included in the gem5-gpu distribution (J. A. Stratton, 2012). The *LACore* implementation was hand-written C code leveraging the *LACoreAPI*.

The *LACore* uses an iterative Radix-2 implementation of the Cooley-Tukey algorithm (Tukey, 1965). The algorithm is composed of 3 steps:

1) precompute all twiddle-factors, which are stride-accessed by the *LAMemUnits* during the FFT;
2) perform the bit-reverse algorithm; and
3) iteratively solve the FFT.

```
//'x' value stored in imag parts of 'data'
//'C' are the cos polynomial coefficients
void cosines(Complex *data , IDX count) {
  //NOT SHOWN: setup scalars Zero_REG and One_REG ...
  LAAddr in = (LAAddr)&data->im;
  LAAddr out = (LAAddr)&data->re;
  //tw_REG points to the imaginary parts of each item in 'data' vector
  la_set_vec_dp_mem(tw_REG, out, 1, 1, 1);
  //X_REG points to the real parts of each item in 'data' vector
  la_set_vec_dp_mem(X_REG, in, 1, 1, 1);
  //calculate square of vector: X_REG = X_REG*X_REG (see paper)
  la_AaddBmulC(X_REG, X_REG, Zero_REG, X_REG, count);

  for(IDX i =0; i <8; ++ i) {
    //set the next scalar and then perform
    //  tw_REG = (tw_REG+C[i])*(x^2)
    la_set_scalar_dp_reg(Const_REG, C[i]);
    la_AaddBmulC(tw_REG, tw_REG, Const_REG, X_REG, count);
  }
}
```

Figure 7-8: LACore FFT Twiddle Factor Pre-Computation.

Precomputing the twiddle-factors in step-1 would normally require $VectorSize/2\ sin()$ and $cos()$ calls, but the *LACore* implementation actually compute $cos()$ using the 18-degree polynomial coefficients from (Garrett, 2012), as illustrated in Figure 7-8. This approach allows the Twiddle-Factors to be computed in parallel as opposed to serial calls to $cos()$.

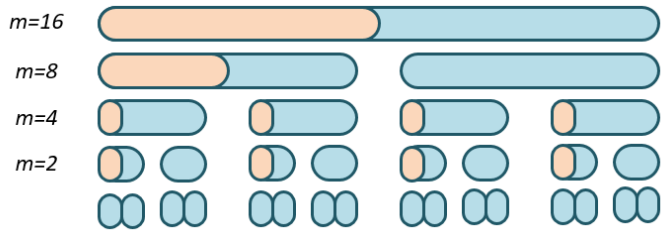

Figure 7-9: LACore's two FFT blocking strategies.

The second step of performing the bit-reversal for the FFT is done using multiple calls to the `la_copy()` function call provided by the *LACoreAPI*. The *LACore*'s actual FFT algorithm is composed of two different algorithms that perform blocks of computation differently depending on how many iterations have been performed. The current algorithm makes this jump after the 5th iteration, when the 'solved' sub-vectors grow to over 32 elements long. The main difference between the two algorithms is the order in which Twiddle-Factors are accessed and the sub-vector elements are accessed, mainly to improve spatial locality and maximize the size of the vectors being operated on at all times. Figure 7-9 demonstrates the pattern that the LACore solves for elements in the FFT: when the current iteration, m, is small, solve the first element in each sub-vector at a time (orange elements). When m is large, we solve for consecutive elements in the same vector. For each iteration in the FFT, the Twiddle-Factors are loaded into the scratchpad, and then a complex double-precision vector multiply is applied to the sub-vectors, and then the result sub-vectors are updated in memory. In Figure 7-9, this means for the initial iterations, when m is small, a single Twiddle-Factor is loaded in to the scratchpad and multiplied by each element from different sub-groups.

7.3.2 Results

The complex double-precision FFT results are shown in Figure 7-10, with the *LACore* significantly outperforming all other implementations for vector sizes up to 2^{14} (or 16384) elements. The *LACore*'s peak double-precision throughput at a vector size of 4096 is 1.88 GFLOP/s, which is a 4.23x, 7.98x, 2.40x and 3.34x speedup over the RISC-V, Fermi GPU, x86-GSL and x86-FFTW implementations. The throughput asymptotically approaches 0.55 GFLOP/s, which is a 2.1x, 1.36x, 1.56x and 0.79x speedup of the RISC-V, Fermi GPU, x86-GSL and x86-FFTW implementations. So the *LACore* is the optimal architecture for small to medium-sized FFTs and nearly as good as the highly-tuned FFTW framework at large-sized FFTs, which implements higher-radix transforms than the simplest Radix-2 (Johnson, 1998). The small performance difference at large input vectors can be explained by the higher-sophistication of the FFTW algorithms than what the *LACore* was running.

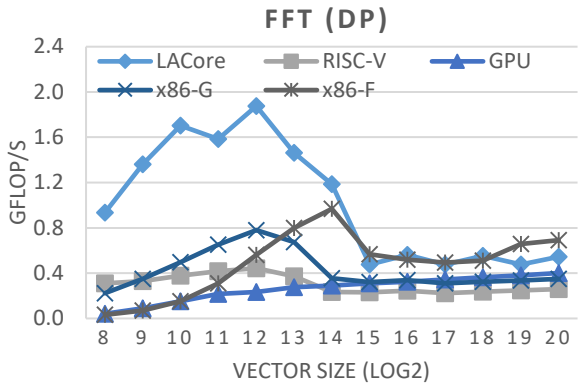

Figure 7-10: FFT Double-Precision on the LACore, RISC-V, Fermi GPU, x86-GSL and x86-FFTW.

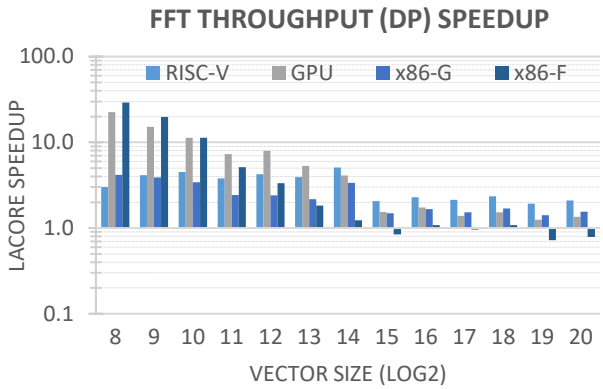

Figure 7-11: FFT Double-Precision Speedup of LACore over RISC-V, Fermi GPU, x86-GSL and x86-FFTW.

7.4 PTRANS

7.4.1 Implementation

Double-Precision Transpose and Add is a high spatial-locality and low temporal-locality application. The benchmark evaluated is $A = A^T + B$, where A and B are n-dimension matrices. The total operations is n^2. The RISC-V platform used the GSL implementation of DGEMM (Gough, 2009), while the x86 version used three implementations with SSE2 enabled: a hand-written kernel, the GSL library, and the Eigen library (G. Guennebaud, 2010). The Fermi GPU ran optimized hand-written kernels, since cuBLAS was not available as a static library for NVIDIA Toolkit 3.2. The `LACore` ran a hand-written kernel leveraging the `LACoreAPI` framework.

7.4.2 Results

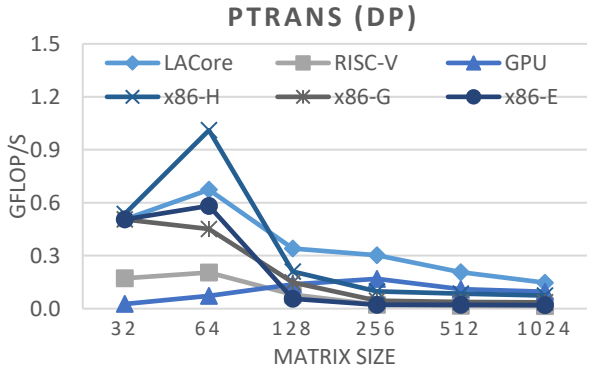

Figure 7-12: PTRANS (DP) on LACore, RISC-V, Fermi GPU, and x86 (hand-written, GSL and Eigen).

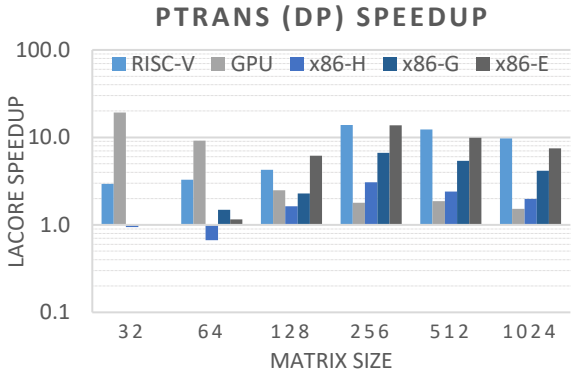

Figure 7-13: PTRANS (DP) Speedup of LACore over RISC-V, GPU, and x86 (hand-written, GSL, Eigen).

The PTRANS absolute performance results are shown in Figure 7-12, with the *LACore* outperforming all other implementations at large matrix sizes, asymptotically reaching 150 MFLOP/s. At this matrix size, the *LACore* achieves a speedup of 9.72x, 1.52x, 1.98x, 4.14x and 7.48x over the RISC-V, Fermi GPU, x86-Hand-Written, x86-GSL and x86- Eigen implementations, as seen in Figure 7-13.

7.5 HPL

7.5.1 Implementation

The High-Performance Linpack solves a linear system $Ax = b$, where A is a square matrix with dimension n, and x, b are column vectors. This is a high temporal and spatial locality application, similar to DGEMM. The total operations for the partial LU factorization and the linear system solving combined is $\frac{2}{3}n^3 + \frac{3}{2}n^2$, with the LU factorization portion of the computation dominating for large n.

The RISC-V platform used the GSL implementation for LU-factorization and Linear-System Solve (Gough, 2009). The x86 platform used two implementations, GSL and Eigen, both with SSE2 enabled (G. Guennebaud, 2010). Since cuSOLVER was not available as a static library for NVIDIA Toolkit 3.2, and no implementations could be found for GPU Linear-System Solve, the scaled Fermi GPU platform was not evaluated for this benchmark. It is worth noting that the HPL and DGEMM results are highly correlated (P. R. Luszczek, 2006), and a GPU implementation for DGEMM is provided, so the Fermi GPU HPL performance can be estimated from that. The *LACore* uses a hand-written algorithm leveraging the *LACoreAPI* library.

The *LACore* implementation has two algorithms, and which one to run depends on the input problem size. Figure 7-14 shows that at smaller matrix sizes (less than 128), using the *LAMemUnits* to swap rows (SWAP) of A in memory during the permutation step is faster than using a permutation vector (PVEC) for indirection and not physically swapping rows in memory. As the problem size increases, the cost of swapping rows becomes large, and the PVEC algorithm is more effective. The reason the PVEC algorithm is slower than SWAP at small problem sizes is because the indirection vector prevents the *LAMemUnits* from sequentially accessing items in columns, since each item in a column could potentially be in a different physical row. Accessing consecutive elements in a row is not a problem, though, which is why the *LACore* uses a row-major matrix for HPL.

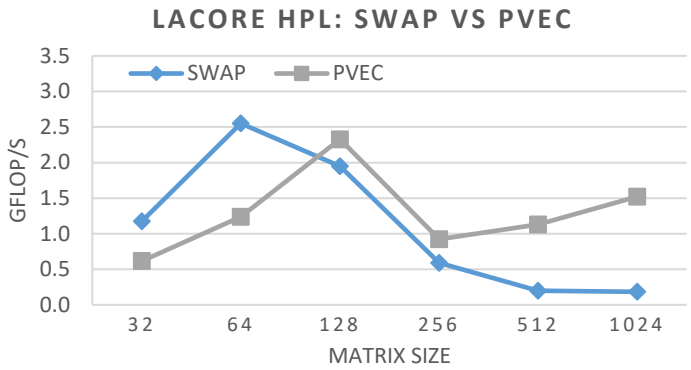

Figure 7-14: HPL performance when swapping rows (SWAP) vs using a permutation vector (PVEC).

A single L-iteration of the LU-decomposition step can be used to visualize how the `LACore` solves HPL. The U-iteration, and the forward and back substitution routines are very similar in structure, so will not be shown. Figure 7-15 shows the elements accessed when solving an L-iteration, which solves for the dark-blue column, and accesses all the elements in the light blue columns and rows. The expression for the elements accessed for each L_{ij} is given by equation 1:

$$L_{ij} = \frac{1}{U_{jj}}\left(A_{ij} - \sum_{k=0}^{j-1} L_{ik}U_{kj}\right) \quad (1)$$

where the U-vector is looped over and the dot-product with each light blue row in the L matrix is computed. The `LACore` LU decomposition solves L and U in place of A, meaning the dark blue column is the same physical memory as the light blue column in A. The actual code used in the HPL benchmark is shown in Figure 7-16.

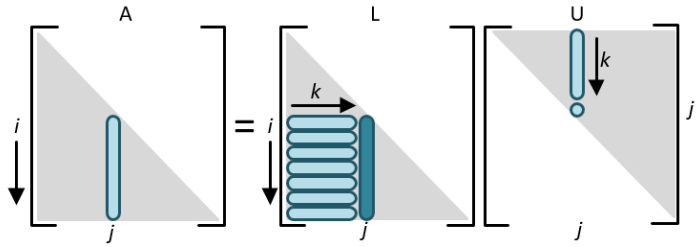

Figure 7-15: Elements in matrices A, L and U accessed during an L-iteration in LU decomposition.

It can be seen from the L-iteration code that if a permutation vector was used (PVEC), the loading of the U column into the scratchpad would take k instructions, since each element in the U column would have to be individually located in memory, while the SWAP version only has to issue a single `la_copy()` command with the appropriate stride to load the elements into the scratchpad. Similarly, the entire loop from $i = j + 1$ to $i = N$ in Figure 7-16 can be replaced by a single

la_AaddBmulC_sum_multi() command with the SWAP algorithm but not the PVEC algorithm. These are two of the reasons PVEC performs poorly at small matrix sizes.

```
//performs L-iteration during LU decomposition on the LACore
void lu_l_iter(double *A , IDX N , IDX j) {
  //NOT SHOWN: load (1 , - U0j .. - Ukj )/Ujj into scratchpad

  for(IDX i=j +1; i <N; ++ i){
    LAAddr Li0 = (LAAddr)&A[N*i];
    LAAddr Lij = (LAAddr)&A[N*i+j];

    //configures input vector Li0 of length j in memory
    la_set_vec_adr_dp_mem(Li0_mem_REG, Li0);
    //configures output scalar Lij in memory
    la_set_scalar_dp_mem(Lij_mem_REG, Lij);
    //performs dot product of length j: Lij = (U0j dot Li0)
    la_AaddBmulC_sum(Lij_mem_REG, U0j_sch_REG, Zero_REG, Li0_mem_REG, j);
  }
}
```

Figure 7-16: LACore LU-Decomposition L-iteration without using permutation vector.

7.5.2 Results

The HPL results are shown in Figure 7-17, with the *LACore* outperforming all other implementations for matrix sizes up to 2^8 (or 256) elements, at which point, x86- Eigen outperforms the *LACore*. The *LACore*'s peak double-precision throughput at a matrix size of 64 is 2.55 GFLOP/s, which is a 4.67x, 2.10x, and 2.81x speedup over the RISC-V, x86-GSL and x86-Eigen implementations. The *LACore*'s throughput asymptotically increases as the matrix size grows, with a throughput of 1.52 GFLOP/s at a 1024 matrix size, which is a 5.88x, 7.22x and 0.57x speedup of the RISC-V, x86-GSL and x86-Eigen implementations, as seen in Figure 7-18.

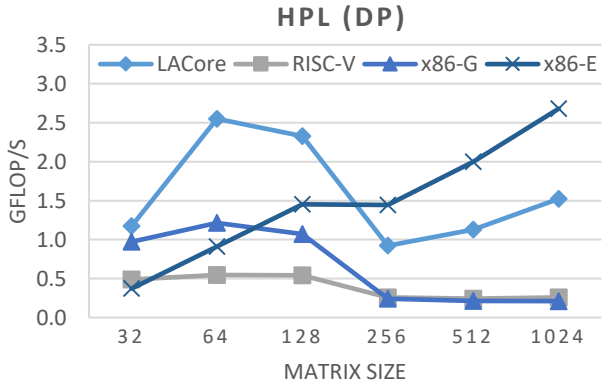

Figure 7-17: HPL Double-Precision on the LACore, RISC-V, x86-GSL and x86-Eigen.

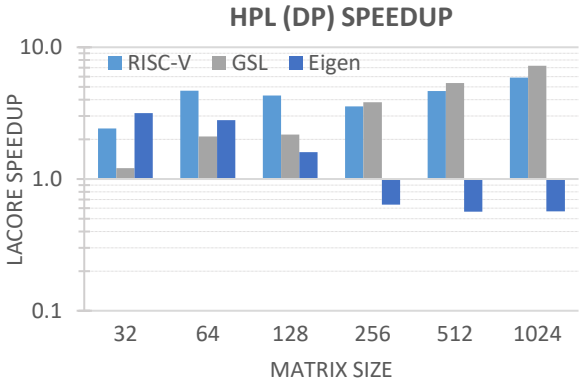

Figure 7-18: HPL Speedup of LACore over RISC-V, x86-GSL and x86-Eigen.

7.6 Random Access

7.6.1 Implementation

Although Random Access (RA) is a purely integer workload, it was included in the evaluated benchmarks in order fully test all four combinations of high-and-low temporal-and-spatial localities. RA tests how architectures perform with low-temporal and low-spatial localities by updating random locations in a large table in memory. The RISC-V and *LACore* platforms ran identical code, since the *LACore*-specific hardware could not be leveraged for this benchmark. The x86 workload was compiled with SSE2 enabled, and the Fermi GPU used a hand-written kernel.

7.6.2 Results

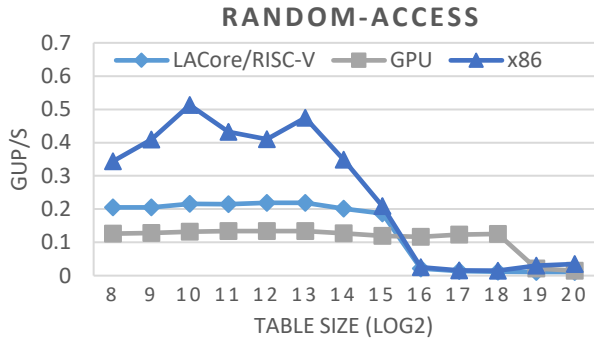

Figure 7-19: Random-Access on LACore/RISC-V, Fermi GPU and x86.

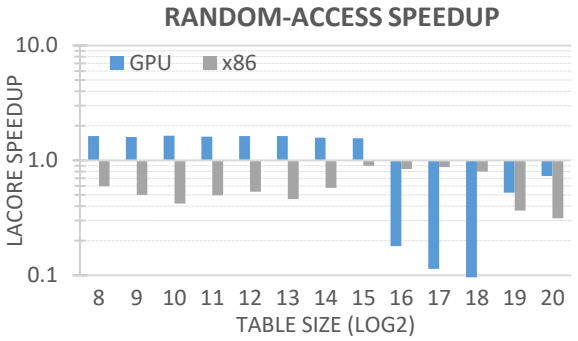

Figure 7-20: Random-Access Speedup of LACore/RISC-V over x86 and Fermi GPU.

As mentioned, the *LACore* ran an identical binary to the RISC-V platform, since none of the *LACore*-specific hardware could be leveraged. Figure 7-19 shows the absolute results for all four platforms, while Figure 7-20 shows the relative speedup of the *LACore* over the other platforms. The *LACore* achieves a 1x, 1.6x and 0.5x speedup over the RISC-V, Fermi GPU and x86 platforms at smaller table sizes, and achieves 1x, 0.7x and 0.3x speedups as the table size grows arbitrarily large.

A large reason x86 performed better than the *LACore* is due to the x86 running a superscalar (multiple-issue, out-of-order) CPU model in gem5, while the *LACore* was only using a single-issue pipelined CPU model. The Fermi GPU performed better at table sizes between $2^{16} - 2^{18}$ because its L2-cache was 1 MB while the *LACore*'s L2-cache was only 256 kB, and these two cache sizes are the upper and lower bounds of the region the GPU outperforms the *LACore*. The table in RA still fits in the whole L2 cache in the GPU's case, but not in the *LACore's* case within this envelope. Above a 2^{18} table size, the GPU starts to have L2 cache evictions as well, and its performance starts to match the *LACore* again.

7.7 STREAM-TRIAD

7.7.1 Implementation

This benchmark is high-bandwidth workload with high-spatial locality and low-temporal locality, since each element in the vectors are accessed sequentially one time. The RISC-V and x86 platforms ran no special libraries for the STREAM Triad benchmark, although the x86 workload was compiled and run with SSE2 enabled. The Fermi GPU implementation offloaded the vector computation as device kernels. The *LACore* implementation made use of the custom *LAExecUnit* and *LAMemUnits* for high-bandwidth memory transfer and vector arithmetic.

The full STREAM Triad C code for the *LACore*, using the *LACoreAPI* framework, is shown in Figure 7-21, and requires only 5 commands for any sized vectors. The first 4 commands configure the input and output streams, which in this case is a scalar and three vectors. The last instruction runs for count elements, using the configured *LACfgs*.

```
// a(i) = b(i) + q*c(i)
void dstream_triad_la_core(double *A, double *B, double q,
                           double *C, IDX count)
{
  //these are the LACfg indexes used by each input or output
  const LACfgIdx A_REG = 0;
  const LACfgIdx B_REG = 1;
  const LACfgIdx C_REG = 2;
  const LACfgIdx q_REG = 3;

  //configure the 3 inputs and 1 output for STREAM
  la_set_scalar_dp_reg(q_REG, q);
  la_set_vec_adr_dp_mem(A_REG, (LaAddr)A);
  la_set_vec_adr_dp_mem(B_REG, (LaAddr)B);
  la_set_vec_adr_dp_mem(C_REG, (LaAddr)C);

  //vector output: A_REG = C_REG*q_REG + B_REG
  la_AmulBaddC(A_REG, C_REG, q_REG, B_REG, count);
}
```

Figure 7-21: LACore STREAM-Triad Code.

7.7.2 Results

STREAM Triad provides two correlated results: memory bandwidth (Bytes/s) and computation throughput (FLOP/s). The absolute results for the bandwidth are shown in Figure 7-22. The chart for GFLOP/s looks nearly identical, so is not shown. The GB/s for Triad is $3 \cdot VectorSize$ according to (P. R. Luszczek, 2006), and the *LACore* peaks at 103 GB/s for a vector size of 4096, where it has 13x, 47x and 4.3x speedups over RISC-V, Fermi GPU and x86 implementations. As the vector size grows, the *LACore*

asymptotically approaches 7.25 GB/s, with speedups of 5.2x, 2.2x and 5.8x over RISC-V, Fermi GPU and x86 implementations, as seen in Figure 7-23. Similar results for the GFLOP/s absolute and relative performances are shown in Figure 7-24 and Figure 7-25 respectively.

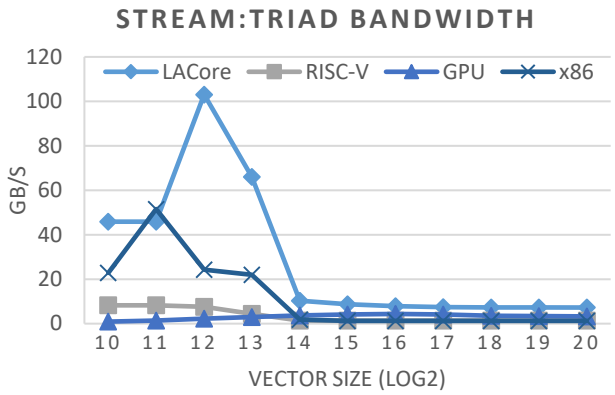

Figure 7-22: STREAM Triad Bandwidth comparison for LACore, RISC-V, Fermi GPU and x86.

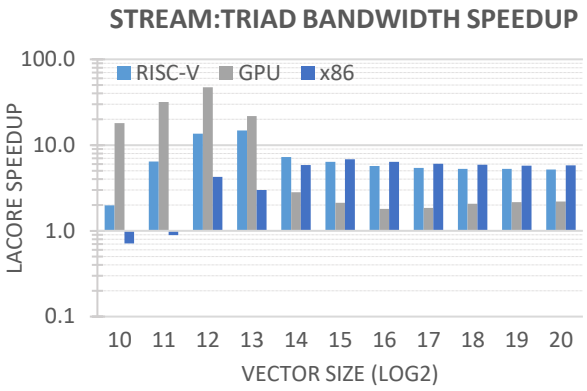

Figure 7-23: STREAM Triad, LACore's Bandwidth increase over RISC-V, Fermi GPU and x86.

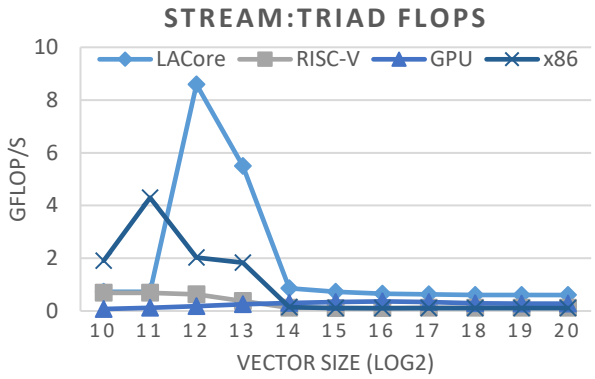

Figure 7-24: Figure 12 1: STREAM Triad GFLOP/s comparison for LACore, RISC-V, Fermi GPU and x86.

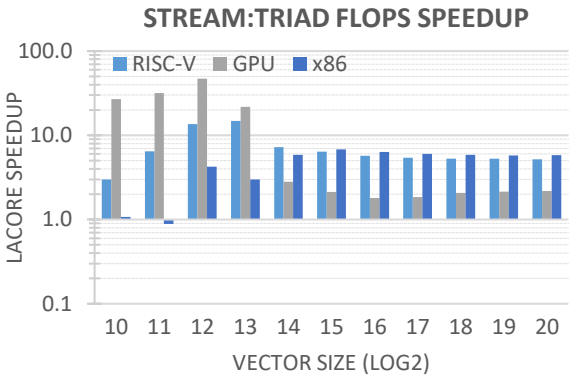

Figure 7-25: STREAM Triad, LACore's GFLOP/s increase over RISC-V, Fermi GPU and x86.

One reason why `LACore` outperforms the other platforms so significantly in STREAM-Triad is due to its previously described large-format vector-like architecture with the specialized streaming `LAMemUnits`, which can access multiple consecutive elements in a cache-line in the same cycle. In other words, the `LACore` can exploit the high spatial-locality of STREAM benchmarks.

7.8 SPARSE DGEMV

7.8.1 Implementation

Sparse DGEMV (SpMV) has an irregular memory-access pattern and a relatively high arithmetic complexity. It was chosen to evaluate the `LACore` because the HPCC benchmark suite does not have

any sparse applications, and the sparse-matrix data-source configuration is a major feature of the *LAMemUnits*. Performance for various matrix sizes and sparsities were collected, with sparsity values of 20%, 40%, 60% and 80% evaluated, corresponding to matrices that are 20%, 40%, 60% and 80% filled with non-zeros respectively. The performance of SpMV on the *LACore* was compared to three other implementations: a RISC-V in-order CPU using the GSL library, and an x86 with SSE2 enabled using both Eigen and GSL implementations. Since cuSPARSE was not available as a static library for NVIDIA Toolkit 3.2, and no implementations could be found for GPU Sparse DGEMV, the scaled Fermi GPU platform was not evaluated for this benchmark. The *LACore* implementation made use of the custom *LAExecUnit* and *LAMemUnits* for high-bandwidth memory transfer and vector arithmetic. The programming model for sparse matrices is straightforward with the *LACoreAPI*, and the full kernel used for the *LACore*'s SpMV evaluation is shown in Figure 7-26. The kernel loads the vector b into the scratchpad, then performs the matrix-vector multiplication using the multi-stream output mode, and then stores the resulting vector, x, from scratchpad back into memory.

```
//A in row major, b,x are column vectors
void dgemv_spv_la_core(double *A_data_ptr, uint32_t *A_col_ptr,
  uint32_t *A_row_ptr, double *b, double *x, IDX N, IDX M)
{
  //these are the LACfg indexes used by each scalar, vector or sparse matrix
  const LaRegIdx Zero_REG  = 0;
  const LaRegIdx A_mem_REG = 1;
  const LaRegIdx b_mem_REG = 2;
  const LaRegIdx b_sch_REG = 3;
  const LaRegIdx x_mem_REG = 4;
  const LaRegIdx x_sch_REG = 5;

  la_set_scalar_dp_reg(Zero_REG, 0.0);
  //configure the sparse matrix 'A' with its 6 fields
  la_set_spv_nrm_dp_mem(A_mem_REG, (LaAddr)A_data_ptr, (LaAddr)A_row_ptr,
                                   (LaAddr)A_col_ptr, N, M, 0);
  //configure the dense 'b' and 'x' vectors in scratchpad and memory
  la_set_vec_dp_mem(b_mem_REG, (LaAddr)b,      1, M, 0);
  la_set_vec_dp_sch(b_sch_REG, 0,              1, M, -M);
  la_set_vec_dp_mem(x_mem_REG, (LaAddr)x,      1, N, 0);
  la_set_vec_dp_sch(x_sch_REG, SCRATCH_SIZE/2, 1, N, 0);

  //copy 'b' vector from mem to scratchpad, since it is frequently accessed
  la_copy(b_sch_REG, b_mem_REG, M);
  //perform x = A*b, with result x being stored in scratchpad
  la_AmulBaddC_sum_multi(x_sch_REG, A_mem_REG, b_sch_REG, Zero_REG, M*N);
  //copy 'x' from scratchpad to memory, now that computation is over
  la_copy(x_mem_REG, x_sch_REG, N);
}
```

Figure 7-26: Sparse DGEMV kernel using LACoreAPI.

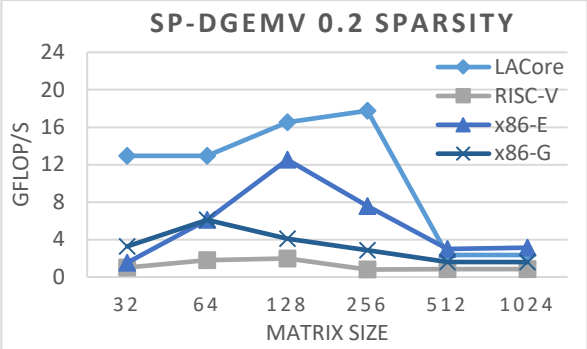

Figure 7-27: Sparse DGEMV (20%) GFLOP/s comparison for LACore, RISC-V, x86-Eigen and x86-GSL.

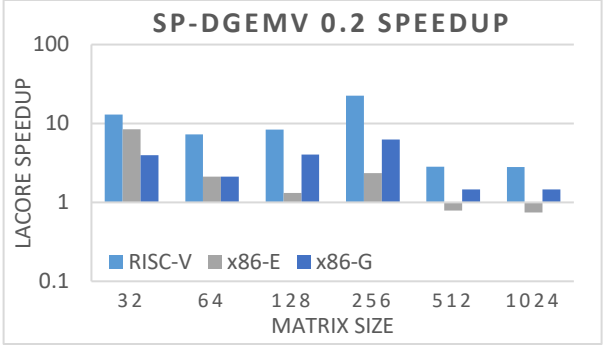

Figure 7-28: Sparse DGEMV (20%), LACore's GFLOP/s speedup over RISC-V, x86-Eigen and x86-GSL.

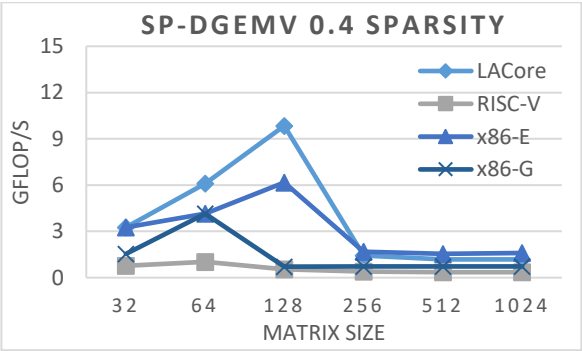

Figure 7-29: Sparse DGEMV (40%) GFLOP/s comparison for LACore, RISC-V, x86-Eigen and x86-GSL.

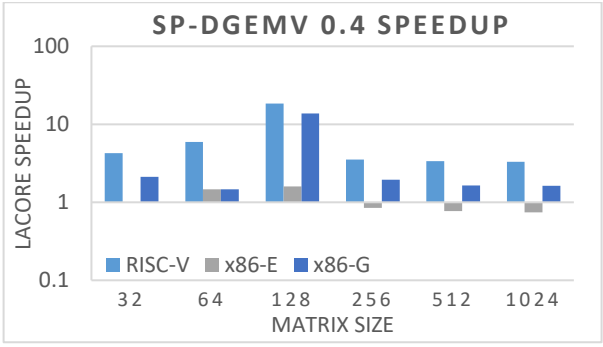

Figure 7-30: Sparse DGEMV (40%), LACore's GFLOP/s speedup over RISC-V, x86-Eigen and x86-GSL.

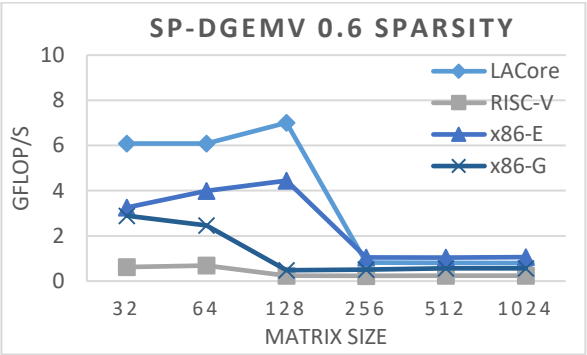

Figure 7-31: Sparse DGEMV (60%) GFLOP/s comparison for LACore, RISC-V, x86-Eigen and x86-GSL.

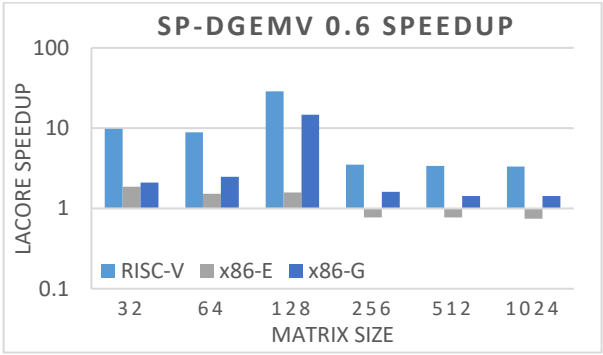

Figure 7-32: Sparse DGEMV (60%), LACore's GFLOP/s speedup over RISC-V, x86-Eigen and x86-GSL.

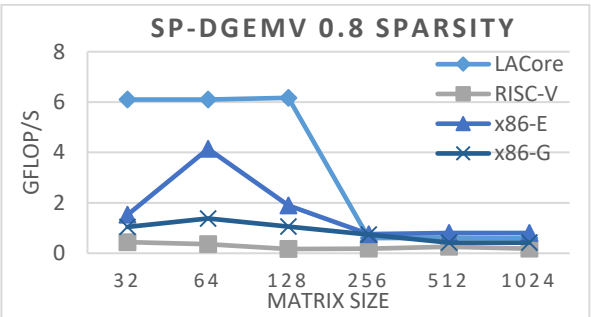

Figure 7-33: Sparse DGEMV (80%) GFLOP/s comparison for LACore, RISC-V, x86-Eigen and x86-GSL.

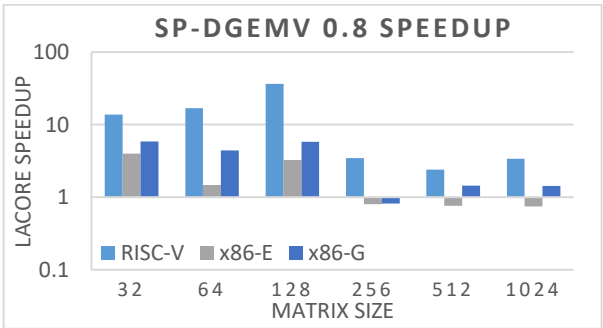

Figure 7-34: Sparse DGEMV (80%), LACore's GFLOP/s speedup over RISC-V, x86-Eigen and x86-GSL.

7.8.2 Results

The absolute performance for each platform at sparsities of 20%, 40%, 60% and 80% are shown in the left column of images above: Figure 7-27, Figure 7-29, Figure 7-31, and Figure 7-33 respectively. The relative speedups of the *LACore* over the other platforms for Sparse DGEMV are shown in images for the corresponding sparsities - Figure 7-28, Figure 7-30, Figure 7-32, and Figure 7-34 respectively.

GFLOP/s were calculated using the equivalent dense matrix sizes. For example, SpMV with a 20% sparsity matrix with side n, would have a GFLOP/s of $2n^3$. Same with 40%, 60% and 80% matrices. At a 20% sparsity, the *LACore* achieves a peak performance of 17.8 GFLOP/s at a matrix size of 256, which is a 22.4x, 2.3x and 6.2x speedup over the RISC-V, x86-Eigen and x86-GSL implementations respectively. After a matrix size of 256, all implementations suffer a substantial drop in throughput due to the problem size exceeding cache capacity. At a sparsity of 40%, the *LACore* achieves a peak performance of 9.8 GFLOP/s at a matrix size of 128, which is a speedup of 18x, 1.6x and 13.8x over the RISC-V, x86-Eigen and x86-GSL platforms. At a sparsity of 60%, the *LACore* achieves a peak performance of 7.0 GFLOP/s at a matrix size of 128, which is a speedup of 28x, 1.6x and 14.6x over the RISC-V, x86-Eigen and x86-GSL implementations. Finally, at a sparsity of 80% (which is the most-dense of all 4 sparsities tested), the *LACore* achieves a peak performance of 6.2 GFLOP/s, which is a 36x, 3.3x, and 5.8x speedup over the RISC-V, x86-Eigen, and x86-GSL implementations.

The most noticeable trend in all the SpMV results is that the *LACore* dominates all other platforms up until the problem size exceeds a threshold determined by the cache size. At this point, all platforms' performances drop by up to an order of magnitude, with the *LACore* and x86-Eigen asymptotically achieving very similar results, despite x86 eigen being a superscalar processor and *LACore* being a single-issue pipelined model in gem5. This severe drop in performance at matrix sizes of 256 to 512 is surprising because the three vectors accessed in SpMV are simply accessed sequentially in memory, so it is not immediately clear that this would strain the caches. But a closer analysis suggests that as the sparse matrix size grows, there will be more accesses to the $data$ vector and jdx vector for each access to the idx vector, which could result in the idx elements in the cache to get evicted more frequently, thus causing more cache misses. In other words, the three vectors making up the compressed sparse matrix will cause thrashing in the cache since they will be competing for cache-lines with each other.

Another view of the SpMV results can be seen in Figure 7-35, which overlays the 20%, 40%, 60% and 80% sparsity results for the *LACore* on top of each other. In addition, Figure 7-35 also has the

DGEMM averaged results from the HPCC benchmark suite overlaid as well. The relativity results are not too surprising, with the DGEMM performance being better than all the SpMV performances, and the performance of the sparse benchmarks improving with lower sparsity, since there are more zeros to skip over. One interesting observation is that DGEMM appears to scale well with problem size, while SpMV does not even degrade gracefully with problem size. The analysis for this phenomenon has already been given though.

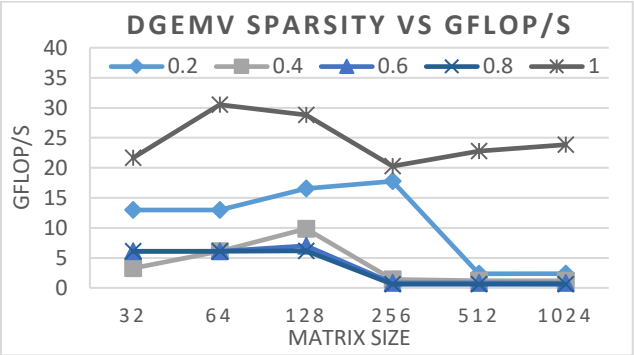

Figure 7-35: Sparse DGEMV, LACore performance vs matrix sparsity.

The overall sparse DGEMV benchmark results suggest that the `LACore` is a good platform for applications involving sparse-matrices, especially for matrix sizes less than 512x512. The `LACoreAPI` kernel for sparse DGEMV was demonstrated and clearly showed the high-programmability and simplicity of dealing with sparse matrices on the `LACore`.

8 ROOFLINE MODEL

8.1 LACORE ROOFLINE MODEL

A roofline model for the *LACore* has been created using the results from the HPCC benchmarks and Sparse DGEMV benchmarks. The roofline model attempts to envelope the upper bound GFLOP/s for any type of application on a given architecture. The envelope is limited by the maximum floating-point performance of the hardware at high operational complexity problems, and limited by memory bandwidth at low computation complexity problems, according to (Williams S. W., 2009), the original source of the Roofline model.

		Roofline Ridge
Max DP GFLOPs (Scalar)	184	1.78
Max DP GFLOPs (Vec)	128	1.24
STREAM-Triad max GB/s	103	

Table 8-1: LACore Roofline Model Parameters.

The *LACore*'s roofline model is composed from the elements in Table 8-1. The STREAM-Triad maximum bandwidth occurs at a vector size of 4096 elements, as mentioned in the HPCC results section. The maximum scalar-output double-precision floating-point performance is 184 GFLOP/s while the maximum vector-output double-precision floating-point performance is 128 GFLOP/s, as discussed in the LACore Architecture section. The intersection of the bandwidth envelope and the maximum hardware GFLOP/s forms the ridge at 1.78 operational complexity for scalar output and 1.24 operational complexity for vector output.

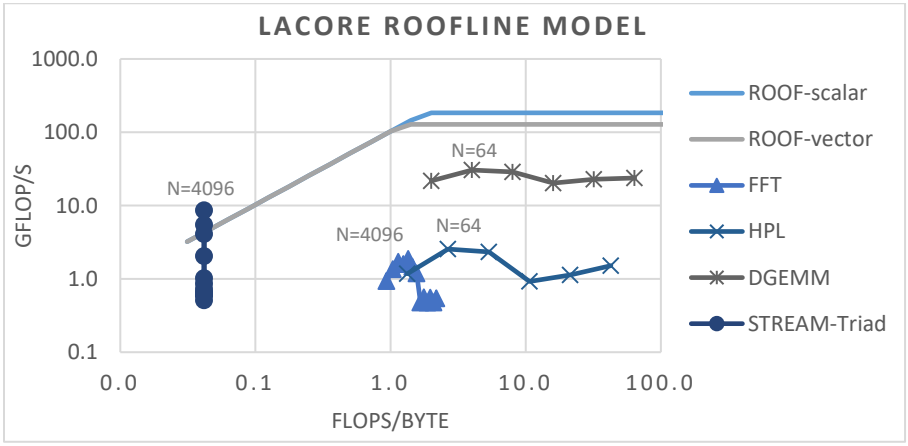

Figure 8-1: LACore Roofline Model with selected HPCC applications.

The *LACore* roofline model is presented in Figure 8-1, with the FFT, HPL, DGEMM and Stream-Triad results included as well. Each point for each benchmark corresponds to a different vector or matrix size, and the sizes that produced the highest performance for each benchmark were labeled on the chart: FFT and STREAM GFLOP/s peaked at a vector size of 4096, while DGEMM and HPL GFLOP/s peaked at a matrix size of 64x64. The applications' operational complexities were taken from a few sources: (Ofenbeck, 2014) was used for DGEMM and (Williams S. , 2016) was used for FFT and STREAM. No derivations for HPL's operational complexity could be found in literature, so it was derived by hand. The operational complexities for each benchmark are shown in Table 8-2. The FLOP/BYTE determines the x-axis position of the results at the given input size. For example, at a matrix size of 32, DGEMM would have a FLOP/BYTE ratio of 2, but at a matrix size of 128, DGEMM would have a FLOP/BYTE ratio of 8, which can be seen in the figure above as well. For STREAM-Triad, the FLOP/BYTE is independent of the problem size, so the plotted results are on a vertical line.

	DGEMM	STREAM-Triad	HPL	FFT-1D
Bytes Moved	$32n^2$	$24n^2$	$16n^2$	$48n$
FLOP/s	$2n^2$	n^2	$\frac{2}{3}n^3$	$5n\log(2n)$
FLOP/BYTE	$\frac{n}{16}$	$\frac{1}{24}$	$\frac{n}{24}$	$\frac{5}{48}\log(2n)$

Table 8-2: FLOP/Byte calculations for HPCC Applications.

8.2 LACORE ROOFLINE ANALYSIS

The roofline model and plotted results reveal a few interesting qualities of the *LACore* and the HPCC benchmark implementations. The first observation is that the *LACore* has a ***balanced*** roofline ridge at 1.24 and 1.78 for vector and scalar output modes. This means that neither the memory bandwidth of the system nor the theoretical computational performance of the system are the major limitation. If the memory bandwidth was low compared to the peak hardware performance, the ridge would be much farther to the right, at a FLOP/Byte of 4, 8 or even higher. This would mean that barely any applications except for the extremely computational-bound would be able to fully utilize the computational resource of the system. On the other hand, having a memory bandwidth that was much better relative to the system's computational power would result in a roofline peak at a FLOP/Byte of 0.5, 0.25 or even lower, and means that the memory system is never fully utilized, except for the extremely memory-bound applications. So, the *LACore*'s balanced roofline ridge means no part of the overall system was overdesigned, which reduces unnecessary complexity and area cost.

Another observation from the Roofline model is that HPL and FFT perform well below the roofline, which means that it is neither being bound by memory resources or computational resources, and are actually being limited by poor algorithmic design. DGEMM performs closer to the roofline limit, and STREAM-Triad actually outperforms the theoretical roofline at a vector size of 4096, which suggests the roofline model is just a theoretical model and does not impose hard restrictions on application performance, but mere "soft-envelopes". The other interpretation would be that the *LACore*'s architecture somehow violates some assumptions made by the authors of the roofline model.

9 DESIGN AREA ESTIMATES

9.1 AREA ESTIMATION

Dual Mode floating point circuits for Add, Subtract, Compare, Multiply and Divide are used in the `LAExecUnit`'s datapath. A 110nm dual-mode Adder, 90nm dual-mode Multiplier, and 90nm dual-mode 2-stage Divider are presented in (Akkaş, 2008), (Jaiswal, 2015), and (Jaiswal, 2016). Area for each of these circuits is shown in Table 9-1, along with the instance count of each in the `VecNodes`, `ReduceNodes` and the `AccumulateNode`. The total area footprint of the `LAExecUnit` datapath at the 32-nm technology node is found by multiplying the instance count of the Adders, Multiplier and Dividers by the scaled-down area of each from the 110 and 90 nm nodes to the 32 nm nodes, and turns out to be 4.37 mm^2.

	Count/SIMD	Adders	Multipliers	Dividers
VecNode	8/8	64	64	64
ReduceNode	7/8	56	0	0
AccumulateNode	1/8	15	0	0
Cycles		5	4	18/14
Fmax *(GHz)*		1.25	1	1
Node *(nm)*		110	90	90
Area *(mm^2)*		138300	138000	206701
Area *(mm^2)*		1.58	1.12	1.67
Total Area *(mm^2)*		**4.37**		

Table 9-1: Total area estimation of the LAExecUnit's datapath at the 32 nm node.

Using the design for a clock-crossing FIFO from (Cummings, 2002), the FIFOs in the `LAExecUnit`'s datapath were modeled as dual-port RAMs. Then, using the CACTI 5.3 web interface, an area estimation for a single FIFO was found, from which the total area used by FIFOs in the `LAExecUnit`'s datapath could be found. The number of FIFOs and their sizes are tabulated in Table 9-2. Each FIFO is 8 words deep, with 64 lines per word. This is equivalent to a 512-byte RAM, so there are 20 kB of FIFOs in the `LAExecUnit`'s datapath.

FIFO depth	8	**VecNode input FIFOs**	24
FIFO word width	64 bytes	**ReduceNode input FIFOs**	15
FIFO area *(mm^2)*	0.017	**AccumulateNode output FIFOs**	1
Total FIFO Area *(mm^2)*	**0.68**	**Total datapath FIFOs**	**40**

Table 9-2: LAExecUnit FIFO area calculations.

The other components in the *LAExecUnit* include 48 64- bit multiplexers and 4 *LAMemUnits*, which will all be grouped into the control portion of the *LACore*, and a conservative estimate of 25% of the total area and power of the *LACore* will be allocated to this control portion. Area estimates for the caches and scratchpad in the *LACore* were found using the CACTI 5.3 Web Interface (D. Tarjan, 2006) for the 32-nm process, and are summarized in Table 9-3. The configuration for the caches and scratchpad were the same as those shown in the gem5 configuration in Table 7-1. The scalar CPU area is taken from the design in (B. Keller, 2016), a RISC-V processor with a vector-extension unit implemented on a 28-nm node. The total area of the Scalar CPU, not including the vector extension was 0.461 mm^2.

Memory	**Area** *(mm^2)*
64 kB *LACache*	1.58
64 kB Data-Cache	1.58
16 kB Inst-Cache	1.10
256 kB L2-Cache	2.02
64 kB Scratchpad	0.50
Total	**6.78**

Table 9-3: Area usage of LACore caches and scratchpad at the 32-nm node.

9.2 Area Comparisons

The full *LACore* area estimation is given in Table 9-4, with the *LACore*-specific hardware taking up 60.4% of the total area. The *LACore* uses 2.62x the area resources as a simple RISCV CPU, including all the caches and scratchpad. This increase in size is justified by the massive performance improvements across the board in the HPCC benchmarks, such as an 85.8x improvement in DGEMM, a 5.8x improvement in HPL and up to a 5.8x improvement in FFT. In all cases, it is more area efficient to use an *LACore* vs multiple RISC-V cores to achieve the same performance. For example, with DGEMM, it would take 85 RISC-V cores to achieve the same result as a single *LACore* at a matrix size of 1024.

	Area *(mm^2)*
RISC-V Scalar	0.46
Inst-Cache/Data-Cache/L2-Cache	4.70
Scalar Total	**5.16**
LAExecUnit datapath/FIFOs	5.05
LAExecUnit control	1.26
LACache/scratchpad	2.08
LACore Total	**13.55**
LACore/Scalar Ratio	**2.62**

Table 9-4: Total RISC-V Scalar CPU and LACore Area Breakdown.

A comparison of the LACore's area to two NVIDIA P100 Streaming Multiprocessors (SMs) can similarly be drawn by using the statistics in (NVIDIA, 2016) for a single SM. For the global resources, such as the GPU L2 cache, the value is divided by 28, which is half the number of SMs on the GPU. The comparison of the *LACore* with the two SMs is shown in Table 9-5. The P100 SM has a larger area than the *LACore*, and also uses more memory resources. The similar area footprints of the two SMs and the *LACore* is the main reason why this paper chose to compare HPCC benchmarks for two SMs instead of the whole GPU. For a fair comparison to the entire P100, a 30-*LACore* manycore chip would have to be compared.

	2x P100 SM	**LACore**
Area *(mm^2)*	21.78	13.55
2x P100 SM/LACore Area	**1.61**	
L1-Cache (kB)	0	144
L2-Cache (kB)	146	256
Registers/FIFOs (kB)	512	20
Scratchpad (kB)	128	64
2x P100 SM/LACore Memories	**1.62**	

Table 9-5: 2 NVIDIA P100 SMs vs the LACore area and memory usage.

10 CONCLUSIONS AND FUTURE WORK

10.1 CONCLUSIONS

In this thesis, the *LACore*, a novel large-format vector architecture for linear algebra application was presented. The *LACore* architecture is designed to address many of the shortcomings of modern architectures for HPC linear algebra applications through unique architectural features such as the highly-configurable Large-Format *LAMemUnits* which can stream arbitrarily sized vectors and matrices to and from the scratchpad and memory, and the mixed-precision vector-reduction *LAExecUnit*, which can achieve up to 368 FLOP/cycle single-precision performance. Additionally, the *LACore* architecture is equipped to handle long-running complex linear algebra kernels with its 64 kB high-throughput, low-latency scratchpad, used to store intermediate results.

A main goal of the *LACore* architecture is to provide a medium ground between the too-general-purpose GPU accelerators, and the too-rigid ASIC/FPGA fixed function accelerators. To achieve this goal the *LACore*'s instruction set was carefully crafted to contain 68 powerful instructions for configuration, data movement and data execution. On top of the ISA, a C-programming framework called the *LACoreAPI*, was developed and demonstrated in this paper. The *LACoreAPI* raises the programming abstraction for the user and provides a flexible and intuitive interface for programming the *LACore*.

To provide a mechanism for design space exploration, and to provide a platform to evaluate benchmarks on, the *LACore* was implemented in the industry-standard cycle-accurate gem5 simulator using a single-issue, pipelined RISC-V processor as the scalar CPU. The *LACore* performance was evaluated against a RISC-V processor, an x86 processor and a scaled NVIDIA Fermi GPU using the gem5 simulator and the gem5-gpu simulator in the Fermi GPU's case.

The HPCC benchmark suite and the Sparse DGEMV kernel were used to evaluate the general purpose linear algebra capabilities and the sparse-matrix capabilities of the *LACore*. Within the HPCC benchmark suite, the *LACore* outperformed all other platforms in STREAM, FFT, HPL, DGEMM and PTRANS applications for a wide range of problem sizes. Additionally, the *LACore* outperformed the RISC-V and x86 platforms substantially in the Sparse DGEMV benchmark for the whole range of matrix sparsities tested. The *LACore* is demonstrated to be an optimal architecture for both high-and-low spatial-and-temporal locality applications through the HPCC and Sparse DGEMV benchmark results.

Finally, the *LACore*'s Roofline model and area footprint were presented. The Roofline model analysis revealed that the *LACore* had a balanced design, with neither the memory subsystem or the computational performance lagging behind the other by a substantial margin. The *LACore* area analysis revealed that the *LACore* used less area and memory resources than a scaled NVIDIA GPU, while offering higher-performing capabilities for linear algebra applications. This demonstrates that the *LACore* architecture can compete with industry-standard hardware in performance per area, which is a strong indicator in the scalability of the *LACore* to manycore designs.

10.2 Future Work

The *LACore* is a new and novel architecture with many areas of exploration and development underway. This paper focused mainly on the architecture of a single *LACore* processor, the initial design space exploration using gem5, the instruction set and the *LACoreAPI* programming model, and the performance of single-threaded applications compared to other platforms. Future and current work will expand and build on this foundational work.

There are many areas of the *LACore* architecture that have room for design-space exploration and performance evaluation, such as the mixed-precision *LAExecUnit* with the ability to coerce single-precision and double-precision data-streams to the other format on the fly. Additionally, more linear algebra applications with sparse-matrix data-sources will be evaluated, such as sparse LU decomposition and applications involving the transposition of sparse matrices.

Current work has been done to develop and evaluate multi-core and many-core *LACore* chips. A C-programming, task-scheduling framework for multi-core *LACore* designs, called the *LATaskSchedulingAPI*, has already been developed to provide a high-level API for programmers to target these multi-core *LACore* designs. Additionally, the gem5 simulator has already been modified to provide support for additional System Calls used by the *LATaskSchedulingAPI*. Future work will continue to build upon these pieces in order to explore effective NoC designs and programming patterns for multi-core *LACore* designs.

Another area of planned future work is an FPGA implementation of the *LACore* architecture discussed in this thesis, which we would then run our benchmarks on and compare to results on other real-hardware platforms for the RISC-V, x86 and NVIDIA GPU architectures. This will provide performance and area comparisons of a hardware implementation of the *LACore* against hardware

implementations of modern HPC acceleration hardware. The eventual goal after FPGA implementation and performance-tuning is an ASIC implementation and tape-out for a single-*LACore* processor, which will provide the most accurate area, power and performance numbers compared to other modern HPC architectures.

Most of the future work presented so far has discussed the architecture and hardware-related aspects of future work. However, there is a plethora of work to be done on the software side as well. There are current plans to develop a fully-BLAS compatible library targeting the *LACore*, which can then be linked against to allow running linear algebra frameworks, such as the GNU Scientific Library to run on the *LACore* without modification.

Additionally, kernels for a wider range of application domains will be explored using the *LACoreAPI* framework, in order to evaluate the full scope of applications that the *LACore* is suited for. There are current plans for implementing multi-layered neural networking algorithms, which we believe will be able to benefit from the vector-reduction datapath in the *LAExecUnit*. Additionally, more complex applications based in linear algebra will be developed, such as fully solving Partial Differential Equations using stencil computations.

REFERENCES

A. Bakhoda, G. L. (2009). Analyzing cuda workloads using a detailed gpu simulator. *IEEE International Symposium on Performance Analysis of Systems and Software*, (pp. 163-174).

Akkaş, A. (2008). Dual-mode floating-point adder architectures. *Journal of Systems Architecture, 54(12)*, 1129-1142.

Amdahl, G. M. (1967). Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (pp. 483-485). ACM.

Andrew Waterman, Y. L. (2011). *Spike, a RISC-V ISA Simulator*. Retrieved from Github: https://github.com/riscv/riscv-isa-sim

Asanović, K. &. (2014). *Instruction sets should be free: The case for risc-v. .* University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146.

Asanovic, K. A. (2016). *The rocket chip generator.* EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17.

Asanovic, K. B. (2006). *The landscape of parallel computing research: A view from berkeley (Vol. 2).* Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

B. Keller, M. C.-F. (2016). Sub-microsecond adaptive voltage scaling in a 28nm fd-soi processor soc. *European Solid-State Circuits Conference*, (pp. 269-272).

Banakar, R. S. (2002). Scratchpad memory: design alternative for cache on-chip memory in embedded systems. *Proceedings of the tenth international symposium on Hardware/software codesign* (pp. 73-78). ACM.

Beard, J. C. (2013). Analysis of a simple approach to modeling performance for streaming data applications. *Modeling, Analysis & Simulation of Computer and Telecommunication Systems* (pp. 345-349). IEEE.

Binkert, N. B. (2011). The gem5 simulator. *ACM SIGARCH Computer Architecture News, 39(2)*, (pp. 1-7).

Celio, C. (2014). *The Sodor Processor Collection.* Retrieved from github.com: https://github.com/ucb-bar/riscv-sodor

Celio, C. P. (2015). *The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor.* Tech. Rep. UCB/EECS-2015–167, EECS Department, University of California, Berkeley.

Ciricescu, S. E. (2003). The reconfigurable streaming vector processor (RSVP). *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture* (p. 141). IEEE Computer Society.

Corbal, J. E. (1999). MOM: a matrix SIMD instruction set architecture for multimedia applications. *Proceedings of the 1999 ACM/IEEE conference on Supercomputing* (p. 15). ACM.

CRAY. (1977). *CRAY-1 Computer System Hardware Reference Manual.* CRAY RESEARCH, INC.

Cray. (2003). *Cray Assembly Language (CAL) for.* Seattle, WA.

CRAY. (2016). *Optimizing Applications on the Cray X1TM System.* Retrieved from docs.cray.com: http://docs.cray.com/books/S-2315-50/html-S-2315-50/x5129.html

Cummings, C. E. (2002). Simulation and synthesis techniques for asynchronous FIFO design. *SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers.*

D. Tarjan, S. T. (2006). *"Cacti 4.0," Technical Report HPL-2006-86.* Palo Alto, Tech. Rep: HP Laboratories .

Dennis, J. B. (1980). Data Flow Supercomputers. *IEEE computer, 13(11)*, pp. 48-56.

Duff, I. S. (1989). Sparse Matrix Test Problems. *ACM Trans. Math. Softw.*, 1-14.

Esmaeilzadeh, H. a. (2011). Dark Silicon and the End of Multicore Scaling. *Proceedings of the 38th Annual International Symposium on Computer Architecture* (pp. 365-376). ACM.

Fetzer, E. S. (2006). The Parity protected, multithreaded register files on the 90-nm itanium microprocessor. *IEEE Journal of Solid-State Circuits*, 246-255.

G. Guennebaud, B. J. (2010). *Eigen v3*. Retrieved from http://eigen.tuxfamily.org

Garrett, C. K. (2012). *Fast polynomial approximations to sine and cosine.*

Gough, B. (2009). *GNU scientific library reference manual.* Network Theory Ltd.

Hisamoto, D. L. (2000). FinFET-a self-aligned double-gate MOSFET scalable to 20 nm. *IEEE Transactions on Electron Devices, 47(12)*, (pp. 2320-2325).

Intel. (2017). *Intel Xeon processor E7 v4 Family.* Retrieved from ark.intel.com: https://ark.intel.com/products/96900

J. A. Stratton, C. R.-J.-W.-m. (2012). Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing, vol 127*.

J. Power, J. H. (2015). gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters, vol. 14, no. 1*, 34-36.

Jaiswal, M. K. (2015). Dual-mode double precision/two-parallel single precision floating point multiplier architecture. *Very Large Scale Integration (VLSI-SoC), 2015 IFIP/IEEE International Conference* (pp. 213-218). IEEE.

Jaiswal, M. K. (2016). *Area-Efficient Architecture for Dual-Mode Double Precision Floating Point Division.* IEEE Transactions on Circuits and Systems I: Regular Papers.

Johnson, M. F. (1998). Fftw: An adaptive software architecture for the fft. *IEEE International Conference on Acoustics, Speech and Signal Processing, vol. 3*, (pp. 1381-1384).

Jouppi, N. P. (2017). *In-Datacenter Performance Analysis of a Tensor Processing Unit.* Retrieved from arxiv.org: arXiv:1704.04760.

Kurzak, A. B. (2016). Mixed Precision Iterative Refinement Techniques for the Solution of Dense Linear Systems. *The International Journal of High Performance Computing Applications* , 457-466.

Lawson, C. L. (1979). Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 308-323.

Lee, Y. O. (2015). *The Hwacha Microarchitecture Manual, Version 3.8.* EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-263.

Lee, Y. S. (2015). *The Hwacha Vector-Fetch Architecture Manual, Version 3.8.* EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-262.

Lindholm, E. N. (2008). NVIDIA Tesla: A unified graphics and computing architecture. *IEEE micro, 28(2).*

Luszczek, P. D. (2005). *Introduction to the HPC challenge benchmark suite.* Lawrence Berkeley National Laboratory.

Mao, H. K. (2016). *Hardware Acceleration for Memory to Memory Copies.*

Momose, S. H. (2014). The brand-new vector supercomputer, SX-ACE. *International Supercomputing Conference* (pp. 199-214). Sprint International Publishing.

Moore, G. E. (1998). Cramming more components onto integrated circuits." 86.1 (1998): 82-85. *Proceedings of the IEEE 86(1)*, (pp. 82-85).

Morris, G. R. (2005). An FPGA-based floating-point Jacobi iterative solver. *Parallel Architectures, Algorithms and Networks, 2005. ISPAN 2005. Proceedings. 8th International Symposium* (p. 8). IEEE.

Nepal, K. H. (2016). Automated High-Level Generation of Low-Power Approximate Computing Circuits. *IEEE Transactions on Emerging Topics in Computing*.

NVIDIA. (2016). *NVIDIA Tesla P100 Whitepaper.* Retrieved from nvidia.com: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

Ofenbeck, G. S. (2014). Applying the roofline model. *Performance Analysis of Systems and Software (ISPASS)* (pp. 76-85). IEEE.

P. R. Luszczek, D. H. (2006). The hpc challenge (hpcc) benchmark suite. *ACM/IEEE conference on Supercomputing*, (p. 213).

Patterson, D. A. (1985, Jan). Reduced Instruction Set Computers. *Commun. ACM, 28*(1).

Patterson, D. A. (2013). *Computer organization and design: the hardware/software interface.* Newnes.

Rennich, S. (2011). *CUDA C/C++ Streams and Concurrency.* Retrieved from gputechconf.com: http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf

Rivers, J. A. (1997). On High-bandwidth Data Cache Design for Multi-issue Processors. *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (pp. 46-56). IEEE Computer Society.

Russell, R. M. (1978). The CRAY-1 computer system. *Communications of the ACM 21.1*, (pp. 63-72).

Sanders, J. (2010). *Introduction to CUDA C.* Retrieved from nvidia.com: http://www.nvidia.com/content/gtc-2010/pdfs/2131_gtc2010.pdf

Shao, Y. S. (2016). Co-designing accelerators and soc interfaces using gem5-aladdin. *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium* (pp. 1-12). IEEE.

Shaw, D. E. (2014). Anton 2: Raising the Bar for Performance and Programmability in a Special-purpose Molecular Dynamics Supercomputer. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 41-53). IEEE Press.

Smith, J. E. (1982). Decoupled access/execute computer architectures. *ACM SIGARCH Computer Architecture News (Vol. 10, No. 3)* (pp. 112-119). IEEE Computer Society Press.

Sumita, M. &. (2005). A 32b 64-word 9-read-port/7-write-port pseudo dual-bank register file using copied memory cells for a multi-threaded processor. *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005* (pp. 384-605). IEEE.

Taylor, M. B. (2012). Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. *Design Automation Conference (DAC)* (pp. 1131-1136). IEEE.

Thimmannagari, C. (2004). *CPU design: answers to frequently asked questions.* Springer Science & Business Media.

Tukey, J. W. (1965). An algorithm for the machine calculation of complex fourier series. *Mathematics of computation, vol. 19, no. 90*, 297-301.

Van Zee, F. G. (2015). BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software (TOMS), 41(3).*

Waterman, A. (2016). *Design of the RISC-V Instruction Set Architecture.* EECS Department, University of California, Berkeley.

Waterman, A. a. (2016). *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1.* EECS Department, University of California, Berkeley.

Williams, S. (2016). *Roofline Performance Model*. Retrieved from Berkeley Lab Computational Research: https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/

Williams, S. S. (2006). The potential of the cell processor for scientific computing. *In Proceedings of the 3rd conference on Computing frontiers* (pp. 9-20). ACM.

Williams, S. W. (2009). Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM, 52(4)*, (pp. 65-76).