Technical Report

Towards a High-Performance, Low-Power Linear Algebra Processor

Ardavan Pedram

Andreas Gerstlauer

Robert van de Geijn

The University of Texas at Austin

UT-CERC-10-03

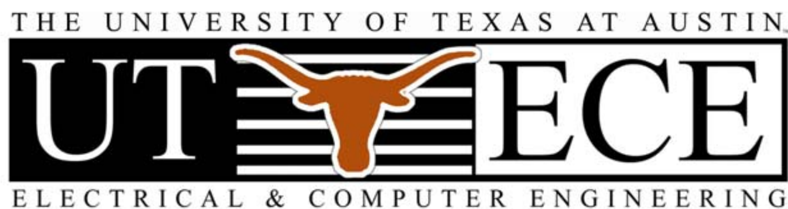
September 1, 2010

Computer Engineering Research Center
The University of Texas at Austin

1 University Station, C8800
Austin, Texas 78712-0323

Telephone: 512-471-8000
Fax: 512-471-8967

http://www.cerc.utexas.edu



Towards a High-Performance, Low-Power Linear Algebra Processor

FLAME Working Note #49

Ardavan Pedram
Andreas Gerstlauer
Robert A. van de Geijn
The University of Texas at Austin
Austin, TX 78712

September 2, 2010

Abstract

Achieving high-performance while reducing power consumption is the key question as technology scaling is reaching its limits. It is well-accepted that application-specific custom hardware can achieve orders of magnitude improvements in efficiency. The question is whether such efficiency can be maintained while providing enough flexibility to implement a broad class of operations. In this paper, we aim to answer this question for the domain of matrix computations. We propose a design of a novel linear algebra processor and demonstrate that it can achieve orders of magnitude improvements in efficiency for matrix-matrix multiplication, an operation that is indicative for a broad class of matrix computations. A feasibility study shows that 46 double- and 113 single-precision GFLOPS/W can be achieved in 13.6 and 11 GFLOPS/mm^2, respectively with current components and standard 45nm technology.

1 Introduction

It is predicted that advances in semiconductor technology will allow for many billions of transistors on a single chip while power concerns will limit the number of transistors that can be active at any given time. The key question going forward is how to minimize, or at least greatly reduce, the power consumption while retaining or improving the achieved performance per unit area. Required efficiencies and optimality requires specialization. At the same time, dark silicon provides us with the opportunity to include such heterogeneous cores on a chip that can be effectively utilized only when needed.

It is well known that full custom, application-specific design of on-chip hardware accellerators can provide orders of magnitude improvements in efficiencies for a wide variety of application domains [11, 32]. The flexibility provided by programmable general purpose machines comes at the expense of inherent instruction handling inefficiencies. Both control and data paths are designed to process an unknown, sequential stream of general fine-grain operations. To achieve performance, aggressive architectural optimizations, such as deep caching and pipelining with associated speculation, reordering and prediction are applied in an effort to dynamically recover inherent parallelism and locality. However, these techniques tend to incur tremendous overheads.

By contrast, in application-specific designs, the order and type of operations to be performed is known at design time. Both control and data paths are hardwired to directly realize the desired

computation. This is possible in domains, such as multimedia or signal processing, where applications are standardized. There, functionality can be realized into fixed hardware, and exponentially growing costs of chip design can be reaped across a large volume of units. The question is whether these concepts can be applied to a broader class of other, more general applications. If in the future neither fine-grain programmable computing nor full custom design are feasible, can we design specialized on-chip cores that maintain the efficiency of full custom hardware while providing enough flexibility to execute whole classes of coarse-grain operations?

In this paper, we aim to answer these questions for the domain of matrix computations, which build the basis for many algorithms in communications, control and scientific computing. It is well understood that linear algebra problems can be efficiently reduced down to a canonical set of Basic Linear Algebra Subroutines (BLAS), such as matrix-matrix and matrix-vector operations [6]. Highly efficient realization of matrix computations on existing general-purpose processors have been studied extensively. Among the highest profile efforts is the currently fastest method for (general) matrix-matrix multiplications (GEMM) [10], which was later generalized to the broader set of BLAS operations [9]. These results have shown that a single approach can achieve high performance across this important set of operations on a broad range of traditional processors.

By contrast, the long-term vision of our project is to design high-performance, low-power linear algebra processors by essentially aiming to realize this method directly in specialized hardware. In the present paper we examine how this can be achieved for GEMM, with an eye on keeping the resulting architecture sufficiently flexible to compute all operations in this class. Our analysis suggests that it should be possible to achieve a performance of 45 double- and 110 single-precision GFLOPS/W in 11-13 GFLOPS/mm^2 with currently available components and technologies as published in literature. This represents a two order of magnitude improvement over current general purpose architectures and a one order of magnitude improvement over current GPUs.

The paper is organized as follows: after a brief discussion and reexamination of deficiencies in related work, we develop our proposed matrix processor architecture aimed at removing such inefficiencies in Section 3. In Section 4 and Section 5 we show the mapping of matrix multiplication onto this processor and we analyze both its theoretical performance and performance characteristics of a realistic implementation based on current technology. The paper concludes with a summary and an outlook on future work in Section 6.

2 Related Works

GEMM implementation on traditional general-purpose architectures has received a lot of attention. Modern CPUs include SIMD units that provide data parallelism without an increased instruction count, which can be exploited for high performance in matrix computations [9, 2, 30]. However, general instruction handling overhead remains and even with SIMD instructions, long computations have to be split into multiple operations that exchange data through a wide register file.

In recent years, GPUs have become a popular target for acceleration. Originally, GPUs were developed as specialized hardware for graphics processing that provided massive parallelism but was not a good match for matrix computations [7]. More recently, GPUs have shifted away from specialization back towards general-purpose architectures. Such GPGPUs essentially replicate a large number of SIMD processors on a single shared memory chip. GPGPUs can be effectively used for matrix computations [3, 28] with throughputs of more than 300 GFLOPS for single-precision GEMM (SGEMM), utilizing around 30-60% of the theoretical peak performance. Since early GPGPUs only included a limited number of double-precision units, their DGEMM performance is less

than 100 GLFOPS (at utilizations of 90-100%). In the latest GPGPUs, single-precision units can be configured as half the number of double-precision ones, achieving up to 600 or 300 GFLOPS at around 60% utilization, respectively [20]. In all cases, however, utilization and achievable performance will drop for smaller kernel sizes (e.g. matrix sizes less than 256).

Over the years, many other parallel architectures for high-performance computing have been proposed and in most cases benchmarked using GEMM as a prototypical application. Systolic arrays were popularized in the 80s [19]. With increasing memory walls, recent approaches have brought the computation units closer to memory, including hierarchical clustering of such combined tiles [23, 16]. Despite such optimization, utilizations for GEMM range from 60% down to less than 40% with increasing number of tiles. Instead of a shared memory hierarchy, the approach in [26] utilizes a dedicated network-on-chip interconnect with associated routing flexibility and overhead. It seems closest to our design, but it is not specifically designed for matrix multiplication and is reported to only achieve around 40% utilization for this application. More recently, Intel developed the Single-chip Cloud Computer (SCC) research processor with 48 Pentium cores [13] arranged as a 2D array with local memory. This processor intended to study question about programmability more than that it targets high performance.

As utilization numbers indicate in all these cases, inherent general-purpose characteristics of data paths and interconnects, coupled with associated instruction inefficiencies make it difficult to fully exploit all available parallelism and locality. By contrast, while we will build on the SIMD and GPU concept of massive parallelism, we aim to provide a natural extension that is further targeted at leveraging the specifics of matrix operations. We can recognize that our class of linear algebra operations essentially consists entirely of multiply-accumulate (MAC) computations with regular and predicable access patterns. As such, we can design a data path that consists of specialized MAC units, which include local accumulators that avoid the need for unnecessary shared storage accesses between basic arithmetic operations. Similarly, partitioned and distributed memories and interconnects can be specifically designed to realize available locality and required access patterns. Finally, control can be predominantly hardwired with a minimal set of micro-coded commands to switch between different coarse-grain matrix processing modes. As such, we focus our work on improved linear algebra processors that can replace traditional SIMD cores for the domain of matrix computations. In the same manner as in GPGPUs, these basic cores can in the future be replicated and dropped into a larger linear algebra processor arrangement.

Specialized hardware implementations of GEMM on FPGAs have been explored before, either as dedicated hardware implementations [34, 33] or in combination with a flexible host architecture [17]. Such approaches show promising results (up to 99% utilization) but are limited by the performance and size restrictions in FPGAs. By contrast, we target an ASIC implementation that will allow us to fully exploit state-of-the-art technologies. Within this context, our goal is to develop a fixed architecture that is flexible yet specialized enough to optimally execute many matrix operations.

3 Basic Design of a Linear Algebra Processor

A high-level design for a *Linear Algebra Processsor* (LAP) is shown in Figure 1. It consists of a 2D array of $n_r \times n_r$ processing elements (PEs), each of which has a MAC unit with a local accumulator, local storage, simple distributed control, and bus interfaces to communicate data within rows and columns. For illustrative purposes we will focus our discussion on the case of a mesh with $n_r \times n_r = 4 \times 4$ PEs.

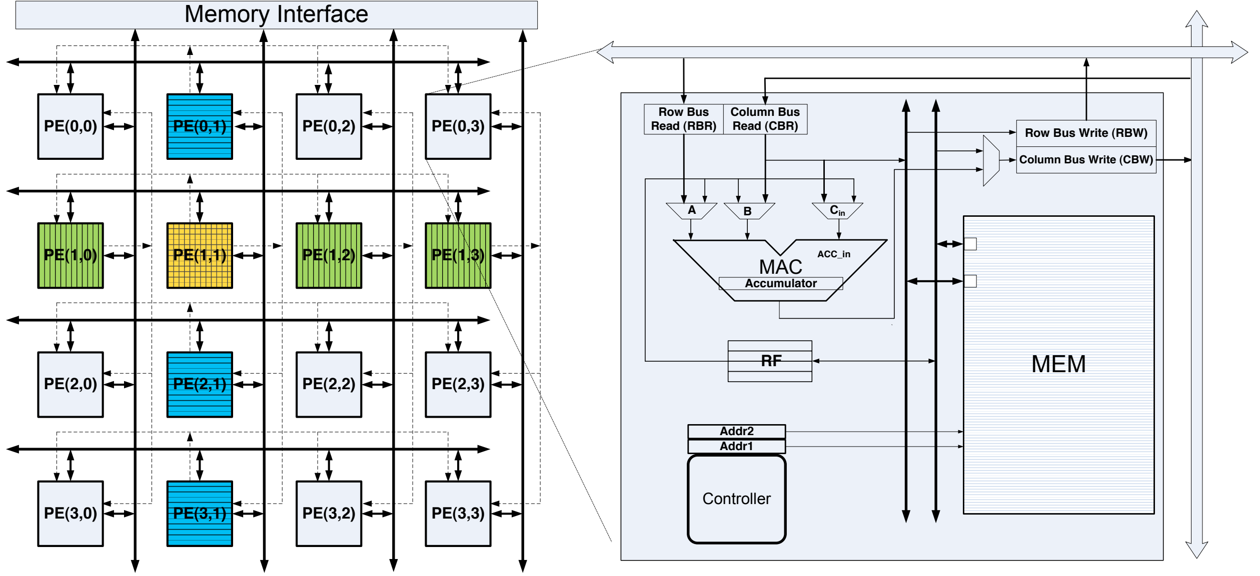


Figure 1: LAP Architecture. The highlighted PEs on the left illustrate the PEs that own the current column of $4 \times k_c$ matrix A and the current row of $k_c \times 4$ matrix B for the second rank-1 update ($p = 1$). It is illustrated how the roots (the PEs in second columns and row) write elements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

3.1 Basic Operation

A special case of GEMM will be used in this section to describe the Linear Algebra Processor: Let C, A, and B be 4×4, $4 \times k_c$, and $k_c \times 4$ matrices, respectively[1]. Then $C \mathrel{+}= AB$ can be computed as

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

so that C is updated in the first iteration with

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

and the second iteration with

$$\begin{pmatrix} \gamma_{0,0} + \alpha_{0,1}\beta_{1,0} & \cdots & \gamma_{0,3} + \alpha_{0,1}\beta_{1,3} \\ \vdots & \ddots & \vdots \\ \gamma_{3,0} + \alpha_{3,1}\beta_{1,0} & \cdots & \gamma_{3,3} + \alpha_{3,1}\beta_{1,3} \end{pmatrix}, \tag{1}$$

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

Let us assume that $4 \times k_c$ matrix A and $k_c \times 4$ matrix B are distributed to the array in a 2D cyclic round-robin fashion, much like one distributes matrices on distributed memory architectures [12, 5].

[1] The choice of parameter labes like n_r and k_c mirrors those used in [10].

In other words, $\alpha_{i,j}$ and $\beta_{i,j}$ are assigned to PE $(i \mod 4, j \mod 4)$. Also, element $\gamma_{i,j}$ of matrix C is assumed to reside in an accumulator of PE (i, j). Then a simple algorithm for performing this special case of GEMM among the PEs is to, for $p = 0, \ldots, k_c - 1$, broadcast the pth column of A within PE rows, the pth column of B within PE columns, after which a local MAC operation on each PE updates the local element of C.

3.2 LAP Architecture

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

Details of the PE-internal architecture are shown in Fig. 1 (right). At the core of each PE is a MAC unit to perform the computations $\gamma_{i,j} \mathrel{+}= \alpha_{i,p}\beta_{p,j}$. Each MAC unit has a local accumulator register that holds the intermediate and final values of one inner dot product of the result matrix C being updated. Apart from preloading accumulators with initial values of γ, all accesses to elements of C are performed directly inside the MAC units, avoiding the need for any register file or memory accesses. We utilize pipelined units that can achieve a throughput of one MAC operation per cycle. Such throughputs can be achieved by postponing normalization of results until the last accumulation [27]. Being able to leverage a fused MAC unit with delayed normalization will also significantly decrease power consumption while increasing precision.

As outlined in Section 3.1, we store the $4 \times k_c$ matrix A and the $k_c \times 4$ matrix B distributed among the PEs in local memories. It is well-understood for dense matrix operations [5, 12] that communication is greatly simplified and its cost is reduced if it is arranged to be only within PE rows and columns. When considering $\gamma_{i,j} \mathrel{+}= \alpha_{i,p}\beta_{p,j}$, one notes that if $\alpha_{i,p}$ is stored in the same PE row as $\gamma_{i,j}$, it only needs to be communicated within that row. Similarly, if $\beta_{p,j}$ is stored in the same column as $\gamma_{i,j}$, it only needs to be communicated within that PE column. This naturally leads to the choice of a 2D round-robin assignment of elements, where $\alpha_{i,p}$ is assigned to PE $(i, p \mod n_r)$ and $\beta_{p,j}$ to PE $(p \mod n_r, j)$.

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

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

Column busses in the PE mesh are multiplexed to both perform column broadcasts and transfer elements of A, B and C to/from external memory during initial preloading of input data and writing back of results at the end of computation. For the latter purpose, PEs can internally read and

write column bus values from/to the MAC accumulator or local memory. In regular operation, row and column busses carry $\alpha_{i,p}$ and $\beta_{p,j}$ values that continuously drive PE-internal MAC inputs in a pipelined fashion. Sending PEs $(i, p \mod n_r)$ and $(p \mod n_r, j)$ drive the busses in each row and column with values out of their local memories, where diagonal PEs $(i = j)$ simultaneously load two values from local memory onto both busses. For simplicity and regularity, sending PEs receive their own broadcasted values back over the busses into the MAC inputs like all other PEs. In such a setup, no additional registers or control are necessary.

Alternatively, we can consider a setup in which all elements $\beta_{p,j}$, $p = 0, \ldots, k_c - 1$ of B are replicated among all PEs in each row j. This eliminates the need to broadcast these values across columns. Instead, elements of B are always accessed locally through an additional register file[2]. Trading off storage for communication requirements, this setup avoids all column transfers, freeing up column busses for prefetching of subsequent input data in parallel to performing computations (see Section 4).

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

3.3 LAP Control

LAP control is distributed and each PE has a state machine that drives a predetermined sequence of communication, storage and computation operations. Local controllers in each PE are equally smart and all agents operate in parallel and in lock step. PE executions are implicitly coordinated and synchronized without any additional handshaking. Instead, inter- and intra-PE data movement is predetermined, and each PE implicitly knows when and where to communicate. Global control and handshaking is limited to coarse-grain coordination for simultaneous triggering or stalling of all PEs at the start of operation or in combination with external memory accesses. State machines are microprogrammed via a few external control bits to select the type of linear algebra operation that the PE should perform. Using only these control signals and counter presets, we expect to be able to support the full flexibility we want for executing, for example, all level-3 BLAS (matrix-matrix operations) [6].

The state machine for local PE control is shown in Figure 2. In the *Init* and *Fetch* states, the PE is initialized, and input data is loaded from external memory and distributed across accumulators and PE memories. The external memory interface in combination with global control drives column busses with 4-element vectors of C, A and B in a row-by-row fashion. Based on the row indices and distribution patterns, individual PEs independently determine which values to latch and where to store them internally. Note that as shown in the example here, the A panel is fetched in 4×4 blocks by looping over its k_c columns in 4-column chunks[3]. Furthermore, note that if B is replicated, every PE will locally store all four row values of every complete column it receives.

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

[3] When extending the approach to larger matrix multiplications in Section 4, we will locally keep multiple panels of a complete A matrix that can then be fetched in different patterns.

After preloading of input data, the following *BC* and *Update* states perform the actual computations. The broadcast state *BC0* fills the pipeline by re-initializing the address registers and letting the sending PEs write the set of elements (for $p = 0$) out onto the row and column bus write latches and busses. From there, values will be latched by all bus read registers with one cycle delay. In the case of replicated B, column busses are bypassed. Instead, values of B are loaded from local memory directly into the local register file.

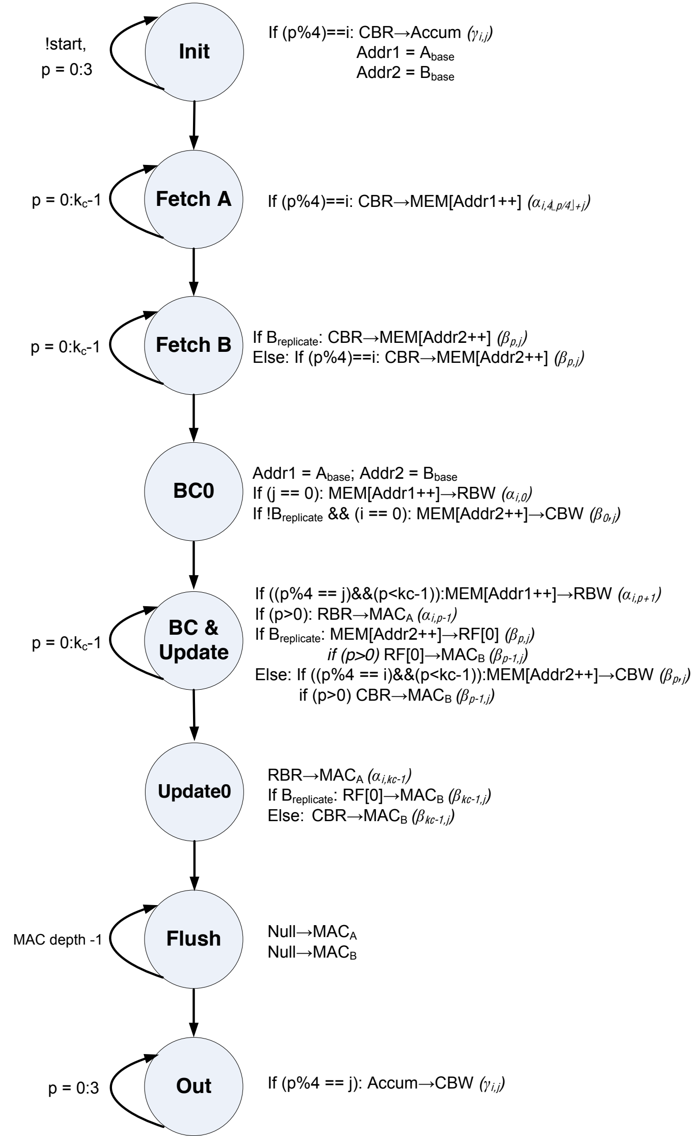


Figure 2: State machine for basic operation of PE (i, j).

Once the first set of MAC inputs are available on the busses, the *BC&Update* state operates the pipeline by performing subsequent broadcasts in parallel with MAC computations for a total of k_c rank-1 update cycles. Once the last set of items has been written into the bus read latches, it is processed in the *Update0* state. Finally, after applying the last set of elements to the MAC inputs, the state machine transitions to the final *Flush* and *Out* states in which the MAC pipeline is flushed and final accumulator values of γ are written back to the column busses and external memory.

The basic state machine in each PE requires eight states, two address registers and one loop counter. In the following sections, we will discuss LAP and PE operation for bigger matrix multiplications that are broken into a sequence of basic rank-k updates using a hierarchical blocking of input matrices. Each additional level of blocking will require an additional loop and loop counter. Since there are no loop-carried dependencies, we pipeline the outer loops to effectively overlap the rank-k computation of the current kernel with prefetching of the next kernel's input data and writeback of the previous kernel's results. As such, functionality of *Init*, *Fetch* and *Out* is merged into the *BC*, *Rank-1* and *Flush* states. With B replicated and all of a larger A local, the resulting state machine has a combined inner core state that runs all operations in a single-cycle loop with full parallelism and essentially 100% sustained LAP utilization. With three levels of blocking, such PE control only requires a total of four counters and ten states.

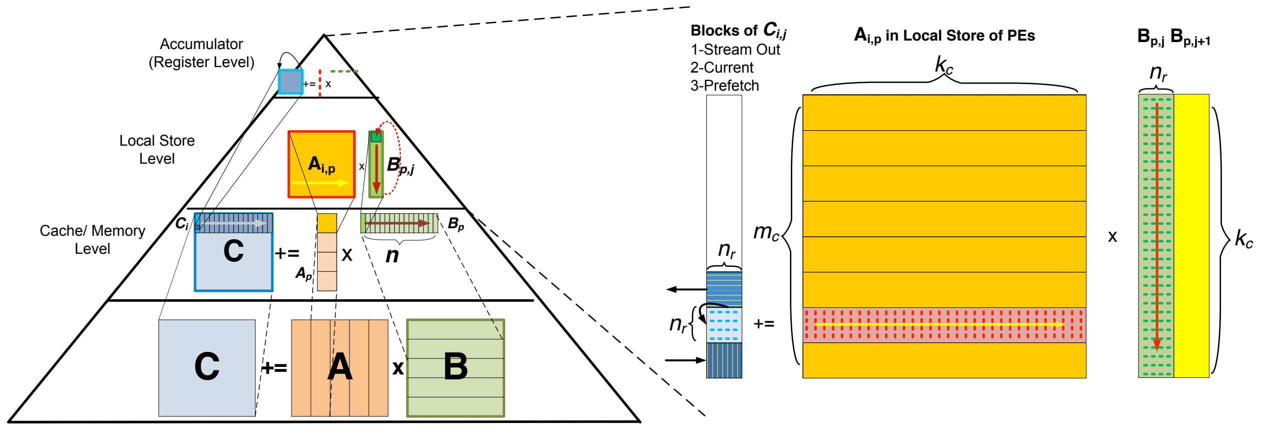


Figure 3: Memory hierarchy while doing GEMM, resident blocks are shown with thick lines in the pyramid.

4 Mapping GEMM to the LAP

In the previous section, we showed how a LAP can easily compute with data that already resides in its memory. The question is now how to compose the GEMM $C \mathrel{+}= AB$ for general (larger) matrices from the computation that can occur on the LAP. The key is to amortize the cost of moving data in and out of the LAP. We describe that in this section with the aid of Figure 3, which depicts the proposed design and use of the memory hierarchy.

4.1 Algorithm

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

$$C \mathrel{+}= \begin{pmatrix} A_0 & \cdots & A_{K-1} \end{pmatrix} \begin{pmatrix} B_0 \\ \vdots \\ B_{K-1} \end{pmatrix} = A_0 B_0 + \cdots + A_{K-1} B_{K-1}$$

so that the main operation to be mapped to the LAP becomes $C \mathrel{+}= A_p B_p$. This partitioning of matrices is depicted in the bottom layer in Figure 3.

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

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

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

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

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

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

Finally, $A_{i,p}$ is partitioned into panels of four rows and $C_{i,j}$ into squares of 4×4, which are processed from top to bottom in a blocked row-wise fashion across i. The multiplication of each row panel of $A_{i,p}$ with $B_{p,j}$ to update the 4×4 block of $C_{i,j}$ is accomplished by the LAP via the rank-1 updates described in Section 3. What is still required is for the 4×4 blocks $C_{i,j}$ to be brought in from main memory.

The described blocking of the matrices facilitates reuse of data, which reduces the need for high bandwidth between the memory banks of the LAP and the external memory:

- Fetching of a 4×4 block $C_{i,j}$ is amortized over $4 \times 4 \times k_c$ MAC operations (4×4 of which can be performed simultaneously).
- Fetching of a $k_c \times 4$ block $B_{p,j}$ is amortized over $m_c \times 4 \times k_c$ MAC operations.
- Fetching of a $m_c \times k_c$ block $A_{i,p}$ is amortized over $m_c \times n \times k_c$ MAC operations.

Note that when this approach is mapped to a general purpose architecture, $A_{i,p}$ is stored in the L2 cache, $B_{p,j}$ is kept in the L1 cache, and the equivalent of the 4×4 block of C is kept in registers [10].

4.2 Architecture

We now translate the theoretical insights about the hierarchical implementation of GEMM into a practical implementation in hardware. In doing so, we derive formulas for the size of the local store, the bandwidth within the LAP, and the bandwidth between the external memory and the LAP. Note that in our subsequent discussion 4×4, the size of the submatrices of $C_{i,j}$, is generalized to $n_r \times n_r$.

The local memory requirements for the LAP are that matrices $A_{i,p}$ and $B_{p,j}$ must be stored in the aggregate memories of the PEs. It was decided to keep duplicates of $B_{p,j}$ within all PEs of a column. It was also decided that computation with the current submatrix of $C_{i,j}$ was to be overlapped with the prefetching of the next such submatrix. Thus, the size of the local store, aggregated over all PEs, is given by

- $m_c \times k_c$ elements for $A_{i,j}$, and
- $2 \times k_c \times n_r \times n_r$ elements for the current and next $B_{p,j}$ and $B_{p+1,j}$.

In total, the local memory must be able to hold $m_c k_c + 2k_c n_r^2 = (m_c + 2n_r^2)k_c$ single or double precision floating point numbers. Note that the $n_r \times n_r$ submatrix of $C_{i,j}$ is always in the accumulators and never stored. However, concurrent prefetching and streaming out of the next and previous such submatrix, respectively, occupies two additional entries in the register file of each PE. Together with a register each for internal transfers of locally replicated $\beta_{p,j}$, every PE requires a register file of size 4 (rounded up to the next power of two).

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

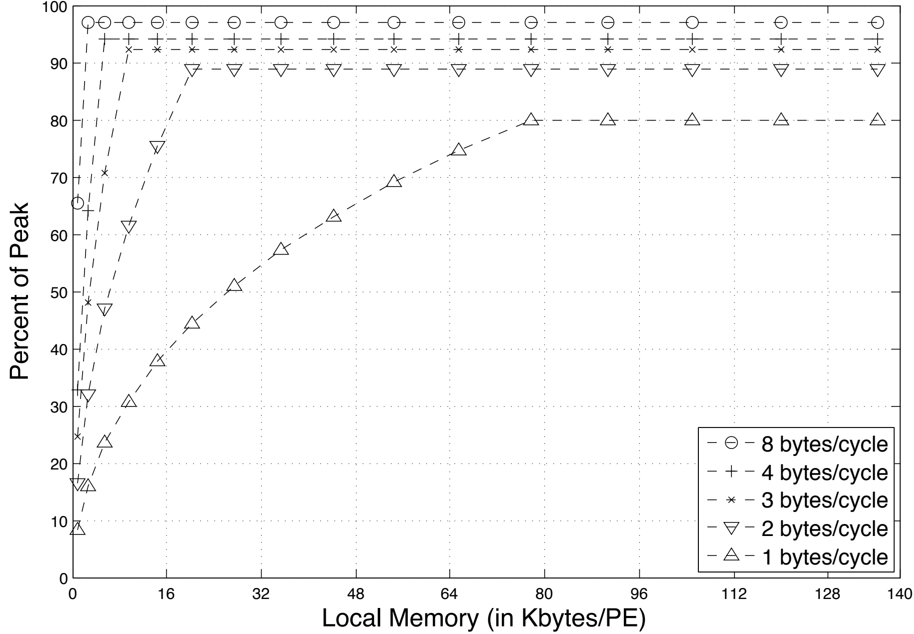


Figure 4: Estimated performance as a function of the external memory bandwidth and the size of local memory with $n_r = 4$, $m_c = k_c$, and $n = 512$.

C_i += $A_{i,p}B_p$ assuming peak performance requires $(m_c k_c n)/n_r^2$ cycles. Overlapping the communication of C_i and B_p with the computation of C_i gives us an estimate for computing C_i += $A_{i,p}B_p$ of

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

Given that at theoretical peak this computation would take $m_c k_c n$ cycles, the attained efficiency is estimated as

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

Notice that the complete computation C += AB requires loops around this "inner kernel" for one C_i and thus it is the performance of this inner kernel that dictates the performance of the overall matrix multiplication.

Figure 4 reports performance as a function of the size of the local memory and the bandwidth to external memory. Here we use $n_r = 4$, $m_c = k_c$ (the submatrix $A_{i,p}$ is square) and $n = 512$ (which is relatively small). This graph clearly shows that a trade-off can be made between bandwidth and the size of the local memory, which in itself is a function of the kernel size (k_c, m_c and n_r).

5 LAP Implementation

To investigate and demonstrate the performance and power benefits of the LAP, we have studied the feasibility of a LAP implementation in current, standard 45nm bulk CMOS technology using publicly available components and their characteristics as published in literature. We stress that the point of this section is not to present the ultimate design. Rather, we are giving evidence that high-performance and low power consumption can be attained by our design using reasonable technology and component choices.

We validated LAP operation and its theoretical performance analysis presented in the previous section by developing a cycle-accurate LAP simulator. The simulator is configurable in terms of PE pipeline stages, bus latencies, and memory and register file sizes. Furthermore, by plugging in power consumption numbers for MAC units, memories, register files and busses, our simulator is able to produce an accurate power profile of the overall execution. We accurately modeled the cycle-by-cycle control and data movement for GEMM, and we verified functional correctness of the produced results. The simulator provides a testbed for investigation of other linear algebra operations, and we were already able to successfully realize Cholesky Factorization with minimal changes to the LAP control and data paths.

5.1 Component Selection

MAC Units: State of the art implementations of Fused Multiply Add (FMA) units use many optimizations techniques to reduce latency, area and power consumption [21]. Fused Multiply Accumulate (FMAC) units use similar architectures but can have delayed normalization to achieve a throughput of one, accumulation per cycle [26, 27]. This technique can also save around 15% of total power since it eliminates two stages of the pipeline for the bulk of operation [14].

Most current designs support floating-point addition, multiplication, multiply-add and multiply-accumulate operations with varying latencies, and pipeline depths, where the number of pipeline stages typically ranges between 5 and 9. Note that these same units can also do integer operations and can be reconfigured to support either single- or double-precision operations [24].

A precise and comprehensive study of different FMA units across a wide range of both current and estimated future implementations, design points and technology nodes was presented in [8]. The authors report efficiencies of 120 GFLOPS/W for a standalone double-precision FMA unit in 45nm technology. Furthermore, paired with a 3-port register file, efficiencies of 90 GFLOPS/W are obtained. These numbers give us an indication of the upper limits that can be achieved.

In the superthreshold regime, frequency and voltage scaling leads to a cubical drop in power consumption while performance only decreases linearly. When aiming for the best possible performance over power ratio, it is therefore beneficial to operate the design at a low voltage and frequency point. In doing so, however, we will also keep in mind that we want to maintain or even exceed the raw performance per unit area of existing processors.

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

Local Storage: Our design utilizes around 16 KBytes of dual-ported SRAM per PE with no tags and no associativity. Given the sequential nature of access patterns to 64-bit wide double-precision numbers, we carefully selected memories with one or two banks to minimize power consumption. Using CACTI [22] with low-power ITRS models and aggressive interconnect projection, we ob-

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

Table 1: 45nm scaled performance and area for LAP with 16KBytes of dual-ported SRAM.

tained area estimates of around $0.13mm^2$ and we calculated the dynamic power of local SRAM at frequencies over 2.5 GHz to be around 13.5mW per port. For the overall system estimation (see Section 5.2), we project the dynamic power results reported by CACTI to the target frequencies of the MAC units. According to the CACTI results, leakage power is estimated to be negligible in relation to the dynamic power.

Interconnect: To estimate latencies and power consumption of row and column busses, we use data reported in [31] and [18]. Since we do not have any of the complex logic for bus arbitration and address decoding, we only consider the power consumption of the bus wires themselves as reported in the papers. With a $n_r \times n_r$ 2D array of PEs, our design contains a total of $2 \times n_r$ 32-bit (single precision) or 64-bit (double-precision) row and column busses. The numbers reported in [18] are for a 32-bit wide AMBA AHB data bus only and are around 1.5 mW. [31] reports around 1.2-2 mW for the same scenario. However, per PE we only have $2/n_r$ of the power consumption of a single bus. Hence, the power consumption of the bus wires is around 1.5-3 mW per PE, where we take the upper limit and double it to account for larger bus widths.

5.2 System Comparison

Overall area, power and performance estimates for our LAP design at various operating points are shown in Table 1. We compare single- and double-precision realizations of our design against other state-of-the-art academic and commercial systems, such as CPUs and GPUs[4]. With efficiency as the primary optimization goal going forward, we compare raw performance, raw power consumption and the critical ratios for performance per unit power and unit area. In relation to efficiency, it is crucial to not only analyze peak performance and power, but to rather consider processor utilization when running a particular application as a key factor. With GEMM being an operation that exhibits amble parallelism and locality and that has been studied extensively through careful and tedious hand-tuning on conventional architectures, many systems, including our LAP, are able to achieve close to peak performance. In contrast to other architectures, however, we expect to be able to sustain such utilization rates for almost all other, more complex linear algebra operations.

For overall comparison of peak performance and power, we extended the analysis presented in [16] by including estimates for our LAP design, the 80-tile network-on-chip architecture from [26], the Power7 processor [29], and a NVidia Fermi GPU (C2050) [1, 20] (Table 2) all scaled to 45nm

[4]Note that comparisons have to be interpreted considering that our analysis uses component numbers available in the public domain, which typically lag several generations behind the state-of-the-art. As such, we can expect even further improvements when transferring our design into a commercial industry-setting in the future.

Architecture	Power density [W/mm^2]	Performance density [GFLOPS/mm^2]	Efficiency [GFLOPS/W]
Cell (SP)	0.3	1.8	6.0
Nvidia GTX280 (SP)	0.3	3.3	11.0
ATI R700 (SP)	0.6	6.4	10.7
Rigel (OPs)	0.3	8.0	26.6
80-Tile @ 0.8V (SP)	0.2	3.3	17.7
Fermi C2050 (SP)	0.6	2.5	4.3
LAP (SP)	0.1	13.6	113.0
Intel Quad-Core (DP)	0.5	0.4	0.8
IBM Power7 (DP)	0.5	0.5	1.0
Fermi C2050 (DP)	0.5	1.3	2.1
LAP (DP)	0.2	11.0	46.4

Table 2: 45nm scaled performance and area of various systems.

technology. For the Fermi GPU, we base our estimates on a reported power consumption of 238 Watts for 224/448 double/single precision cores operated at 1.15 GHz with a performance of 1030 single-precision or 515 double-precision GFLOPS.

We note that for a single-precision LAP at around 1GHz clock frequency, the estimated performance/power ratio is an order of magnitude better than GPUs. The double-precision LAP design shows around 58 times better efficiency compared to CPUs. The power density is also significantly lower as most of the LAP area is used for local store. Finally, the performance/area ratio of our LAP is in all cases equal to or better than other processors. All in all, with a double-precision LAP we can get up to 27 times better performance in the same area as a complex conventional core but using less than half the power.

6 Conclusions and Future Directions

This paper provides initial evidence regarding the benefits of custom hardware for linear algebra computations. The basic conclusion is that, as had been postulated [11], one to two orders of magnitude improvement in power and performance density can be achieved. The paper also suggests many possible extensions some of which we discuss now. For example, Figure 4 clearly shows the tradeoff between the size of the local memory and bandwidth to external memory. One question that remains is the careful optimization of this tradeoff across the multi-dimensioanl power, performance, utilization and area design space. Using a combination of simulations and further physical prototyping, we plan to address these questions in our future work.

The GEMM operation is in and by itself a sufficiently important operation to warrant the proposed hardware support. GEMM indirectly enables high performance for the level-3 Basic Linear Algebra Subprograms (BLAS) [6, 15] as well as most important operations in packages like LAPACK [4] and `libflame` [25]. For this purpose, we plan to investigate integration of our proposed LAP with such libraries.

The choice of the size of the LAP, $n_r = 4$ is arbitrary: it allows our discussion to be more concrete. A natural study will be how to utilize more PEs yet. As n_r grows, the busses that connect the rows and columns of PEs units will likely become a limiting factor. This could be overcome by pipelining the communication between PEs. Furthermore, bandwidth to an external

memory or host may become a bottleneck. Finally, we can envision a hierarchical clustering of multiple LAPs and second- or third-level memory into large arrays on a single chip.

We started out research by initially designing a LAP for Cholesky factorization, an operation that requires the square root and inversion of scalars. As such, our LAP simulator is already able to simulate both matrix multiplication and Cholesky factorization. It is well-understood that an approach that works for an operation like Cholesky factorization also works for GEMM and level-3 BLAS. Additional evidence that the LAP given in this paper can be extended to other such operations can be found in [9], in which the techniques on which our GEMM is based are extended to all level-3 BLAS. The conclusion, which we will pursue in future work, is that with the addition of a square-root unit, a scalar inversion unit, and some future ability to further program the control unit, the LAP can be generalized to accommodate this class of operations.

Acknowledgments

This research was partially sponsored by NSF grants CCF-0540926, CCF-0702714, and OCI-0850750. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

The authors would like to thank Mr. Jeff Diamond, Dr. Eric Quinnel, and Prof. Earl Swartzlander for the valuable insights they provided regarding this research.

References

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

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

[3] V Allada, T Benjegerdes, and B Bode. Performance analysis of memory transfers and GEMM subroutines on NVIDIA Tesla GPU cluster. *CLUSTER '09*, pages 1 – 9, 2009.

[4] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[5] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Comput. Soc. Press, 1992.

[6] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[7] K Fatahalian, J Sugerman, and P Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. *HWWS '04:ACM SIGGRAPH/EUROGRAPHICS*, Aug 2004.

[8] S Galal and M Horowitz. Energy-efficient floating point unit design. *IEEE Transactions on Computers*, PP(99):1 – 1, 2010.

[9] Kazushige Goto and Robert van de Geijn. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Softw.*, 35(1):1–14, 2008.

[10] Kazushige Goto and Robert A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Trans. Math. Soft.*, 34(3):12, May 2008. Article 12, 25 pages.

[11] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. *ISCA '10*, Jun 2010.

[12] B. A. Hendrickson and D. E. Womble. The Torus-Wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Stat. Comput.*, 15(5):1201–1226, 1994.

[13] J. Howard et al. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the International Solid-State Circuits Conference*, February 2010.

[14] S Jain, V Erraguntla, S.R Vangal, Y Hoskote, N Borkar, T Mandepudi, and V.P Karthik. A 90mw/gflop 3.4ghz reconfigurable fused/continuous multiply-accumulator for floating-point and integer operands in 65nm. *VLSID '10.*, pages 252–257, 2010.

[15] B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Soft.*, 24(3):268–302, 1998.

[16] John Kelm, Daniel Johnson, Matthew Johnson, Neal Crago, William Tuohy, Aqeel Mahesri, Steven Lumetta, Matthew Frank, and Sanjay Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. *ISCA '09*, Jun 2009.

[17] G Kuzmanov and W van Oijen. Floating-point matrix multiplication in a polymorphic processor. *ICFPT 2007*, pages 249 – 252, 2007.

[18] Kanishka Lahiri. Power analysis of system-level on-chip communication architectures. pages 1–6, May 2010.

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

[20] Rajib Nath et al. An improved MAGMA GEMM for Fermi GPUs. Technical report, LAPACK WN #227, 2010.

[21] E Quinnell, E Swartzlander, and C Lemonds. Floating-point fused multiply-add architectures. *ACSSC 2007*, pages 331 – 337, 2007.

[22] Thoziyoor Shyamkumar et al. CACTI:5.0 an integrated cache timing, power, and area model. Technical Report HPL-2007-167, HP Laboratories Palo Alto, 2007.

[23] Chikafumi Takahashi, Mitsuhisa Sato, Daisuke Takahashi, Taisuke Boku, Akira Ukawa, Hiroshi Nakamura, Hidetaka Aoki, Hideo Sawamoto, and Naonobu Sukegawa. Design and power performance evaluation of on-chip memory processor with arithmetic accelerators. *IWIA2008*, pages 51 – 57, 2008.

[24] D Tan, C Lemonds, and M Schulte. Low-power multiple-precision iterative floating-point multiplier with simd support. *IEEE Transactions on Computers*, 58(2):175 – 187, 2009.

[25] Field G. Van Zee. `libflame`*: The Complete Reference*. `www.lulu.com`, 2009.

[26] S Vangal, J Howard, G Ruhl, S Dighe, H Wilson, J Tschanz, D Finan, A Singh, T Jacob, S Jain, V Erraguntla, C Roberts, Y Hoskote, N Borkar, and S Borkar. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *IEEE Journal of Solid-State Circuits*, 43(1):29 – 41, 2008.

[27] S.R Vangal, Y.V Hoskote, N.Y Borkar, and A Alvandpour. A 6.2-gflops floating-point multiply-accumulator with conditional normalization. *IEEE Journal of Solid-State Circuits*, 41(10):2314–2323, 2006.

[28] V Volkov and J Demmel. Benchmarking gpus to tune dense linear algebra. *SC 2008*, pages 1 – 11, 2008.

[29] Malcolm Ware, Karthick Rajamani, Michael Floyd, Bishop Brock, Juan C Rubio, Freeman Rawson, and John B Carter. Architecting for power management: The IBM® POWER7™ approach. *HPCA 2010*, pages 1 – 11, 2010.

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

[31] Wolkotte. Energy model of networks-on-chip and a bus. *System-on-Chip, 2005.*, pages 82 – 85, 2005.

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

[33] Ling Zhuo and V Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 18(4):433 – 448, 2007.

[34] P Zicari et al. A matrix product accelerator for field programmable systems on chip. *Microprocessors and Microsystems 32*, 2008.