Technical Report

Co-Design Tradeoffs for High-Performance, Low-Power Linear Algebra Architectures

Ardavan Pedram
Andreas Gerstlauer
Robert A. van de Geijn

UT-CERC-12-02

October 5, 2011

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

Co-Design Tradeoffs for High-Performance, Low-Power Linear Algebra Architectures

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

Abstract

As technology is reaching physical limits, reducing power consumption is the key issue on our path to sustained performance. In this paper, we study fundamental tradeoffs and limits in efficiency (as measured in energy per operation) that can be achieved for an important class of kernels, namely the level-3 Basic Linear Algebra Sub-rountines (BLAS). It is well-accepted that specialization is the key to efficiency. This paper establishes a baseline by studying general matrix-matrix multiplication (GEMM) on a variety of custom and general-purpose CPU and GPU architectures. Our analysis shows that orders of magnitude improvements in efficiency are possible with relatively simple customizations and fine-tuning of memory hierarchy configurations. We argue that these customizations can be generalized to perform other representative linear algebra subroutines. In addition to indicating the sources of inefficiencies in current CPUs and GPUs, our results show our prototype linear algebra processor (LAP), double-precision GEMM (DGEMM) can achieve 600 GFLOPS consuming less than 25 Watts in standard 45nm technology, which is up to 50× better than CPUs in terms of energy efficiency.

1 Introduction

We have arrived at a point in time when power consumption is becoming the limiting factor for continued semiconductor technology scaling. While one could view this as a roadblock on the way to exascale computing, we would like to view it as an opportunity. In particular, it is now likely that a future chip may combine heterogeneous cores while having to cope with "dark silicon" [17]. By this, we mean that regions of a chip can be dedicated to highly-specialized functionality without constituting wasted silicon. If only part of the chip can be powered at any given time, those regions can simply be turned off when not in use. This allows us to propose cores that are highly customized for inclusion in such heterogeneous chip multiprocessor designs.

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 [32, 78]. However, full custom design is expensive in many aspects. Hence, the question is whether such concepts can be applied to a broader class of other, more general applications to amortize the cost of custom design by providing multiple functionalities. 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 address these questions for the domain of matrix computations, which are at the core of many applications in scientific, high-performance 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 [53, 14, 13]. 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) [30]. This operation is the building block for other matrix-matrix operations (level-3 BLAS) [39]. In [29], it is shown that this approach can be specialized to yield high-performance implementations for all level-3 BLAS on a broad range of processors.

However, rather than driving microarchitectural design, all these solutions *react* to any hardware changes in order to exploit or work around any new architectural features. We pursue instead the design of high-performance, low-power linear algebra processors that realize algorithms in specialized architectures. We examine how this can be achieved for GEMM, with an eye on keeping the resulting architecture sufficiently flexible to compute all level-3 BLAS operations and to provide facilities for operations supported by LAPACK, like LU factorization with pivoting, QR factorization, and Cholesky factorization. Hence, although we focus our explanation on GEMM, we do so with confidence that modest modifications to the design (e.g., addition of a scalar inversion and/or square-root units with a modified floating point unit) will support all level-3 BLAS and operations beyond.

The main questions when designing these accelerators are as follows: What are the upper limits on performance/power ratios that can be achieved in current and future architectures? What is the algorithm-architecture co-design of optimal accelerator cores? What are the parameters of the memory hierarchy to achieve both high efficiency and high utilization? What are the sources of under utilization and inefficiency in existing general purpose systems?

Previously [62], we introduced a custom micro-architecture design for a linear algebra core (LAC). In this paper, we extend the LAC design with a more general memory hierarchy model to evaluate different trade-offs in system design, including number of cores, bandwidth between layers of memory hierarchy, and the memory sizes in each layer. The results of these analyses are consolidated in a framework that can predict the utilization limits of current and future architectures for matrix computations. Finally, we introduce a prototypical implementation to demonstrate fundamental limits in achievable power consumption in current CPUs and GPUs as compared to an ideal architecture.

Our analysis framework suggests that with careful algorithm/architecture co-design and the addition of simple customizations, it should be possible to achieve efficiencies 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 $50\times$ improvement over current general-purpose architectures and a one order of magnitude improvement over current GPUs.

The rest of the paper is organized as follows: In the next section we briefly discuss related work. Next, Section 3 provides a review of the matrix processor core microarchitecture in our design. In Section 4, we build the memory hierarchy around such cores, show the mapping of matrix multiplication onto this system, and analyze the existing trade-offs in the design space exploration. Section 5 presents performance characteristics of a realistic implementation based on current technology in comparison to other, existing architectures. A summary and outlook on future work is given in Section 6.

2 Related Work

Implementation of GEMM on traditional general-purpose architectures has received a lot of attention. Modern CPUs exploit vector extension units [19, 20, 18, 43] for high performance matrix computations [29, 2, 76]. However, general instruction handling overhead remains. Three main limitations of conventional vector architectures are known to be due to the complexity of the central register file [64], implementation difficulties of precise exception handling, and expensive on-chip memory [42].

In recent years, GPUs have become a popular target for acceleration. Originally, GPUs were specialized hardware for graphics processing that provided massive parallelism but were not a good match for matrix computations [23]. More recently, GPUs have shifted back towards general-purpose architectures. Such GPGPUs replicate a large number of SIMD processors on a single shared-memory chip. GPGPUs can be effectively used for matrix computations [6, 74] 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 units, achieving more than 600 or 300 GFLOPS at around 60% utilization, respectively [59]. In all cases, however, achievable performance will drop for smaller matrix sizes (e.g. matrix sizes less than 512).

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 [47, 49, 48]. Different optimizations and algorithms for matrix multiplication and more complicated matrix computations are compared and implemented on both 1D [73, 68, 45] and 2D systolic arrays [31, 35, 68, 56]. In [38], the concept of a general systolic array and a taxonomy of systolic array designs is presented.

With increasing memory walls, recent approaches have brought the computation units closer to memory, including hierarchical clustering of such combined tiles [66, 41]. Despite such optimization, utilizations for GEMM range from 60% down to less than 40% with increasing numbers of tiles. Instead of a shared-memory hierarchy, the approach in [72] utilizes a dedicated network-on-chip interconnect with associated routing flexibility and overhead. It only achieves around 40% utilization for matrix multiplication. Finally, ClearSpeed CSX700 is an accelerator that specifically targets scientific computing with BLAS and LAPACK library facilities. It delivers up to 75 DGEMM GFLOPS at 78% of its theoretical peak [4].

As utilization numbers indicate in general-purposes cases, inherent characteristics of data paths and interconnects coupled with associated instruction inefficiencies make it difficult to exploit fully 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 leverages the specifics of matrix operations.

In the domain of custom design, recent FPGAs [61, 26] have moved towards Tera-FLOPS peak performance, achieving both high performance and power efficiency. However, FPGAs offer limited on-chip logic capacity, and at slow clock frequencies (100-300 MHz), they can reach high efficiencies but peak performance is limited. According to FPGA vendors, an FPGA with 40nm technology can achieve at most 100 GFLOPS performance at 7 GFLOPS/Watt of power efficiency [60]. Specialized hardware realizations of GEMM and other BLAS routines on FPGAs have been explored, either as dedicated hardware implementations [80, 79] or in combination with a flexible host architecture [50].

Such approaches show promising results (up to 99% utilization), but are limited by the performance and size restrictions in FPGAs [51, 15, 44, 37].

Existing solutions for dedicated realization of matrix operations mostly focus on 1D and 2D arrangements of processing elements [35]. In early FPGA designs with limited logic blocks on the chip, most of the approaches targeted an array arrangement of PEs that pipelines the data in and out of the PEs [46, 79]. Nowadays, with sufficient area on the chip, the design choice between 1D and 2D arrangement of PEs becomes again valid. There are three major benefits of a 2D solution versus a 1D solution: scalability, addressing, and data movement. The 2D arrangement is proven to be scalable with regard to the ratio of problem size to local store memory size for BLAS level operations [16]. Furthermore, address computations and data accesses in local stores of PEs becomes simpler with fewer calculations as compared to a 1D arrangement. This is especially true for more complicated algorithms. Finally, with 2D arrangements, different types of interconnects can be explored, yielding various types of algorithms for BLAS operations. Here, we focus on matrix multiplication.

A taxonomy of matrix multiplication algorithms on 2D grids of PEs and their interconnect requirements is presented in [54]. The algorithms for matrix multiplication are based on three basic classes: Cannon's algorithms (roll-roll-multiply) [9, 57], Fox's algorithm (broadcast-roll-multiply) [24, 11, 21, 22, 54], and SUMMA (broadcast-broadcast-multiply) [5, 69]. Cannon's algorithm shifts the data in two of the three matrices circularly and keeps the third one stationary. Required initial and final alignment of the input matrices needs extra cycles and adds control complexity. In addition, a Taurus interconnect is needed to avoid data contention. Fox's algorithms and its improvements broadcast one of the matrices to overcome alignment requirements. However, a shift operation is still required and such algorithms may show poor symmetry and sub-optimal performance. Finally, the SUMMA algorithm does not need any initial or post-computation alignment. The broadcast is a simple and uniform, single communication primitive, and does not have any bandwidth contention as in circular shifting. In addition, SUMMA is much easier to generalize to non-square meshes of processing units.

The flexibility of the SUMMA algorithm has made it the most practical solution for distributed memory systems [69] and FPGAs [15], and the SUMMA class of algorithms builds the basis for our design. A broadcast operation is an efficient way of data movement to achieve high performance in other BLAS and LAPACK operations. We will see that the cost and latency of broadcast operation does not add extra overhead in our cores.

In most of the previous implementations of dedicated matrix multiplications on systolic arrays and FPGAs, the memory hierarchy was not explored. To study scalability demands, we start by building our system from an inner computation core that is a highly optimized matrix multiplier [62] and build the memory hierarchy around it. In the process, partitioned and distributed memory hierarchies and interconnects can be specifically designed to realize available locality and required access patterns.

3 Design of The Linear Algebra Core (LAC)

In this section, we briefly review the design of a Linear Algebra Core (LAC) [62], as 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.

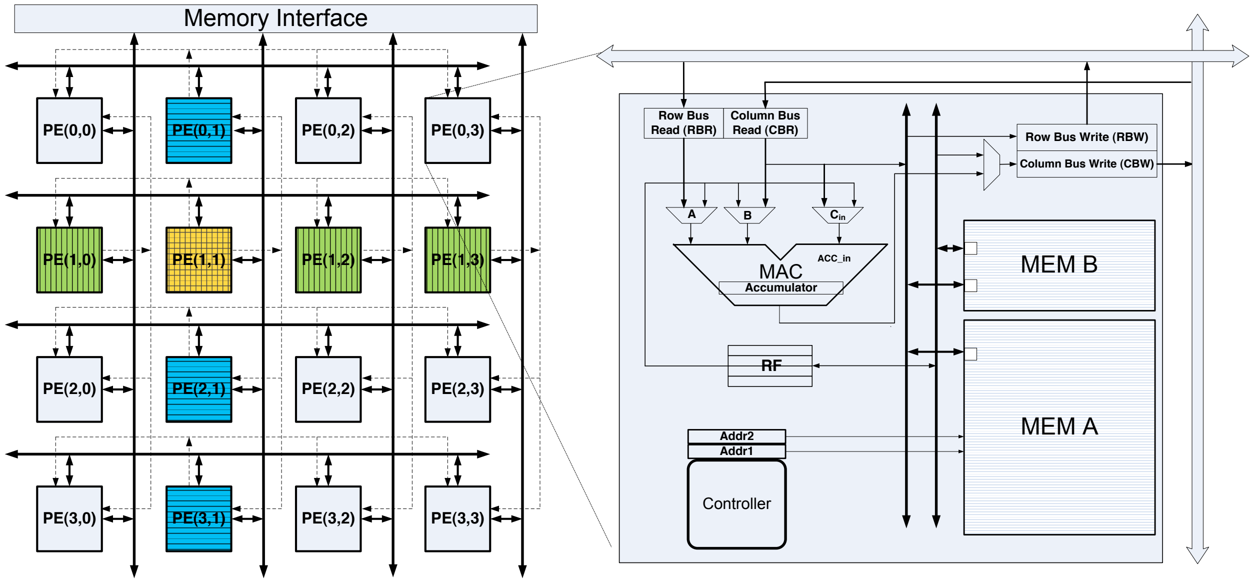

Figure 1: Core architecture. The highlighted PEs on the left illustrate the PEs that own the current column of $4 \times k_c$ matrix A and the current row of $k_c \times 4$ matrix B for the second rank-1 update ($p = 1$). It is illustrated how the roots (the PEs in second column and row) write elements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

3.1 Basic Operation

In the following, we will use a special case of GEMM to demonstrate the basic operation of the LAC. Let C, A, and B be 4×4, $4 \times k_c$, and $k_c \times 4$ matrices, respectively. Then C += AB can be computed as

$$\begin{pmatrix} \gamma_{0,0} \cdots \gamma_{0,3} \\ \vdots \quad \ddots \quad \vdots \\ \gamma_{3,0} \cdots \gamma_{3,3} \end{pmatrix} \mathrel{+}= \sum_{i=0}^{k_c-1} \begin{pmatrix} \alpha_{0,i} \\ \vdots \\ \alpha_{3,i} \end{pmatrix} \begin{pmatrix} \beta_{i,0} \cdots \beta_{i,3} \end{pmatrix}$$

so that C is updated in the ith iteration with

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

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

Let us assume that $4 \times k_c$ matrix A and $k_c \times 4$ matrix B are distributed to the array in a 2D cyclic round-robin fashion, much like one distributes matrices on distributed memory architectures [33, 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, for $p = 0, \ldots, k_c - 1$, to broadcast the pth column of A within PE rows, the pth rows of B within PE columns, after which a local MAC operation on each PE updates the local element of C.

3.2 PE Micro-Architecture

The prototypical rank-1 update given in (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 micro-architecture are shown in Figure 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. Pipelined units are employed 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 [71]. 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, $4 \times k_c$ matrix A and the $k_c \times 4$ matrix B are stored distributed among the PEs in local memories. It is well-understood for dense matrix operations [10, 33] 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 broadcasts of elements $\alpha_{i,p}$ from PE $(i, p \mod n_r)$ within PE rows and of elements $\beta_{p,j}$ from PE $(p \mod n_r, j)$ within PE columns. This is illustrated for the $p = 1$ update in Figure 1. In our design, we connect PEs by horizontal and vertical broadcast busses. The 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.

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 broadcast values back over the busses to feed MAC inputs. In such a setup, no additional registers or control are necessary.

Alternatively, one 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[1].

[1] We include a small, general register file that carries little additional overhead but provides the flexibility of

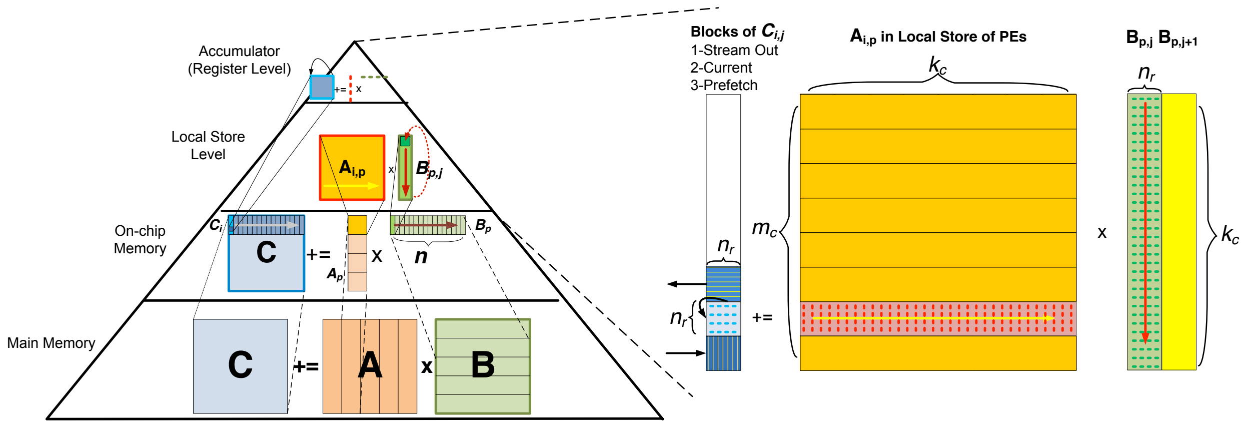

Figure 2: Memory hierarchy while doing GEMM. In each of the top three layers of the pyramid, the largest matrix is resident, while the other matrices are streamed from the next layer down.

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).

3.3 GEMM Algorithm

In designing a complete Linear Algebra Processor (LAP), we not only need to optimize the core, but also describe how data can move and how computation can be blocked to take advantage of multiple layers of memory. In order to analyze the efficiency attained by the core itself, we first need to describe the multiple layers of blocking that are required. We do so with the aid of Figure 2. For now it suffices to think of the LAP as consisting of one of the described cores plus on-chip memory. Later, we will generalize this to one with multiple cores.

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

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

so that the main operation to be mapped to the LAP becomes $C \mathrel{+}= A_p B_p$. This partitioning of matrices is depicted in the bottom layer in Figure 2.

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 $C = \begin{pmatrix} C_0 \\ \vdots \\ C_{M-1} \end{pmatrix}$ 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.

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.

Partition C_i and B_p into panels of $n_r (= 4)$ columns:

$$C_i = \left(C_{i,0} \cdots C_{i,N-1} \right) \text{ and } B_p = \left(B_{p,0} \cdots B_{p,N-1} \right).$$

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

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

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

This approach is very similar to how GEMM is mapped to a general purpose architecture [30]. There, $A_{i,p}$ is stored in the L2 cache, $B_{p,j}$ is kept in the L1 cache, and the equivalent of the 4×4 block of C is kept in registers. The explanation shows that there is symmetry in the problem: one could have exchanged the roles of A_p and B_p, leading to an alternative, but very similar, approach. Note that the description is not yet complete, since it assumes that, for example, C fits in the on-chip memory. Even larger matrices can be accommodated by adding additional layers of blocking, as will be described later (see Section 4.2.3).

3.4 Core Architecture

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

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

The goal is to overlap computation of the current submatrix of $C_{i,j}$ with the prefetching of the next such submatrix. This setup can achieve over 90% of peak performance. Thus, the size of the local store, aggregated over all PEs, is given by $m_c \times k_c$ elements for $A_{i,p}$, and by $2 \times k_c \times n_r \times n_r$ elements for the current and next $B_{p,j}$ and $B_{p+1,j}$. In total, the local memory must be able to

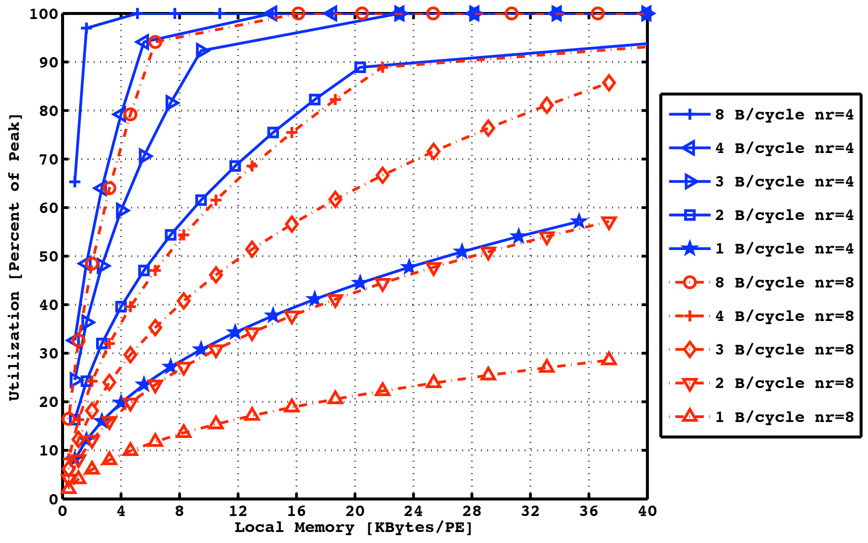

Figure 3: Estimated core performance as a function of the bandwidth between LAC and on-chip memory, and the size of local memory with $n_r = 4$ and $n_r = 8$, $m_c = k_c$, and $n = 512$.

hold $m_c k_c + 2k_c n_r^2 = (m_c + 2n_r^2)k_c$ single or double precision floating point numbers. Note that the $n_r \times n_r$ submatrix of $C_{i,j}$ is always in the accumulators and never stored. However, concurrent prefetching and streaming out of the next and previous such submatrix, respectively, occupies two additional entries in the register file of each PE. Together with a register each for internal transfers of locally replicated $\beta_{p,j}$, every PE requires a register file of size 4 (a size of 3, rounded up to the next power of two).

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

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

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

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

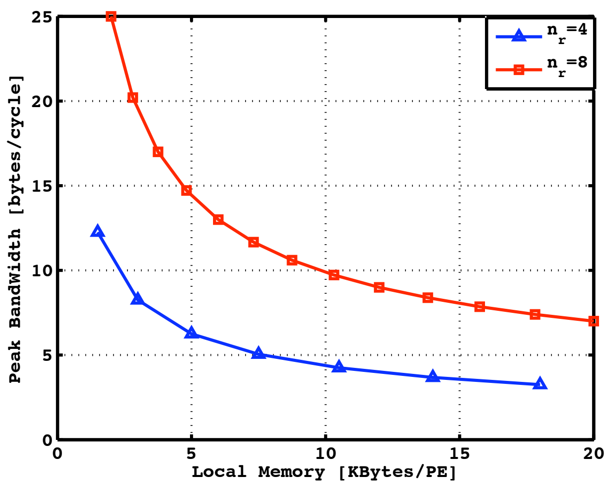

Figure 4: Core Performance vs. bandwidth between LAC and on-chip memory for peak performance with $n_r = 4$ and $n_r = 8$, $m_c = k_c$, and $n = 512$.

3.5 Core-Level Exploration

Figure 3 reports performance of a single core as a function of the size of the local memory and the bandwidth to the on-chip memory. Here we use $n_r \in \{4, 8\}$, $m_c = k_c$ (the submatrix $A_{i,p}$ is square), and $n = 512$ (which is relatively small). This graph clearly shows that a trade-off can be made between bandwidth and the size of the local memory, which in itself is a function of the kernel size (k_c, m_c, and n_r). The graph also shows under what conditions we can achieve 100% utilization.

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

4 Linear Algebra Processor

In the previous section, we showed how a LAC can easily compute with data that already resides in on-chip memory. The question is now how to compose the GEMM $C \mathrel{+}= AB$ for general (larger) matrices from the computations that can occur on a (larger) Linear Algebra Processor (LAP) that is composed of multiple cores. The key is to amortize the cost of moving data in and out of the cores and the LAP. We describe that in this section again with the aid of Figure 2. This framework will allow us to generally study tradeoffs in the memory hierarchy built around the execution cores.

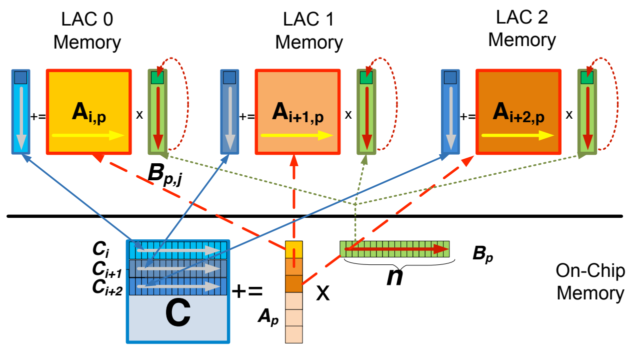

Figure 5: Memory hierarchy with multiple cores in a LAP system.

4.1 LAP Architecture

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

Figure 5 shows the use of the memory hierarchy for a larger matrix multiplication distributed across multiple cores. As discussed previously, each core locally stores a $m_c \times k_c$ (or $2m_c \times k_c$ to allow for prefetching to achieve peak performance) block of $A_{i,p}$, a n_r^2 subblock of $C_{i,j}$ and a $k_c \times n_r$ panel of $B_{p,j}$ (replicated across PEs), where different row blocks and panels of A and C are assigned to different cores. Bigger panels and blocks A, B and C are then stored at the next higher level of the memory hierarchy. Since elements of C are both read and written, we aim to keep them as close as possible to the execution units. Hence, the shared on-chip memory is mainly dedicated to storing a complete $n \times n$ block of matrix C. In addition, we need to share the current $k_c \times n$ row panel of B among the cores. With S cores in the LAP system and space for prefetching of blocks and panels of A and B, the total size of the on-chip shared memory therefore becomes $n^2 + S \times m_c \times k_c + 2k_c \times n$. This on-chip memory size does not reflect full overlapping of computations with communication in the chip level.

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

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

When computation dominates (the second term in the "max" dominates) the peak performance is independent of m_c, i.e. independent of the granularity at which C and the A panel are split into row chunks. Thus, m_c can be chosen to optimize memory bandwidth and the size of local store.

Core	Local Memory size [Words/PE]	Intra-core BW [Words/Cycle]	Core-chip BW [Words/Cycle]
partial overlap	$(n_r^2)(m_c k_c/n_r^2 + 2k_c)$	$n_r(1 + (\frac{2}{k_c} + \frac{1}{m_c}))$	$(\frac{2}{k_c} + \frac{1}{m_c})n_r^2$
full overlap	$(n_r^2)(2m_c k_c/n_r^2 + 2k_c)$	$n_r(1 + (\frac{2}{k_c} + \frac{1}{m_c} + \frac{1}{n}))$	$(\frac{2}{k_c} + \frac{1}{m_c} + \frac{1}{n})n_r^2$
Chip	Memory Size [Words]	Intra-chip [Words/Cycle]	Off-chip BW [Words/Cycle]
partial overlap	$n^2 + Sm_c k_c + 2k_c n$	$(\frac{2S}{k_c} + \frac{1(S)}{m_c})n_r^2$	$\frac{2Sn_r^2}{n}$
full overlap	$2n^2 + Sm_c k_c + 2k_c n$	$(\frac{2S}{k_c} + \frac{1(S)}{m_c} + \frac{S}{n})n_r^2$	$\frac{4Sn_r^2}{n}$

Table 1: Bandwidth and memory requirements of different layers of memory hierarchy based on problem breakdown and hardware parameters.

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

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

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

4.2 Chip-Level Exploration

The overall system design is an optimization and exploration problem that strives to minimize the size of and bandwidth between layers of the memory hierarchy, while optimizing the performance and utilization of the cores. Given specific restrictions, e.g. on memory bandwidth or input matrix size, this yields the number of PEs in each core, the number of cores on a chip and the sizes and organization of the different levels of the memory hierarchy.

Table 1 summarizes the bandwidth and sizes of different layers of the memory hierarchy. This table shows the demands of the partially overlapped and the fully overlapped versions of the algorithm as a function of the number of cores, block sizes, and matrix size when $m = n = k$. In the core level analyses, the partially overlapped version assumes that bringing blocks of $A_{i,p}$ to the core is not overlapped with computation.At the chip level, partially overlapped versions assume that transferring of matrix C to and from off-chip memory is not overlapped with computation.

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

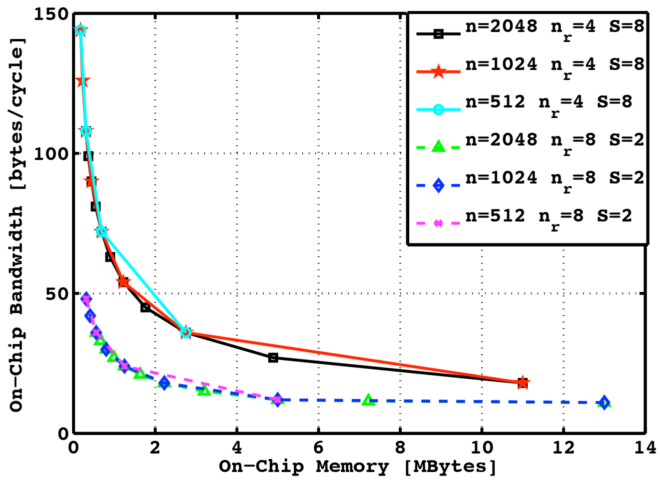

Figure 6: On-chip bandwidth vs. memory size for different core organizations, and problem sizes for fixed number of total PEs, and $m_c = k_c$. The utilization in all cases is over 93%.

4.2.1 Memory size vs. bandwidth

Based on our analytical model, we can evaluate the trade-off between the size of the on-chip memory and the intra-chip bandwidth between cores, and the on-chip memory, as shown in Figure 6. The resulting utilization in all cases is over 90%. We explore this trade off for $S = 8, n_r = 4$ and $S = 2, n_r = 8$ with a total number of PEs on the chip $(S \times n_r^2)$ equal to 128 in both cases. We can note that bandwidth demands grow exponentially as the size of available on-chip memory is reduced. This graph also demonstrates that bigger but fewer cores on the chip demand much less on-chip bandwidth. However, for a fixed problem size of C, bigger cores will require a bigger size of the on-chip memory, leading to a tradeoff between on-chip memory size and bandwidth. This extra space requirement is due to wider panels of A and B that must be stored in the shared memory.

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

We analyze the overall performance of the design when the number of cores is increased for different on-chip memory sizes and on-chip memory bandwidths. The curves in Figure 7 show the percentage of performance compared to a single 4×4 core for different numbers of cores and available on-chip bandwidths. The graph contains four sets of four curves where each set has the same ratio for the number of cores to available on-chip bandwidth S/BW, (indicated by same marker type). We observe that for small memory sizes different points of the same set with the same S/BW ratio all exhibit similar performance. Although the on-chip bandwidth is increased linearly with the number of cores, there is no performance improvement. To achieve performance gains when increasing the number of cores, the bandwidth has to grow superlinearly. However, as the size of memory increases, there is more benefits in using more cores to gain performance even with linear bandwidth increases.

For configurations with the same number of cores S, (indicated by the same line style or color) we observe that, as the bandwidth increases, the curves reach a peak eventually. The point in each curve with the smallest on-chip memory and peak performance is the optimal design point. Note that such a point is on the optimal design curve in Figure 6, too. For example, for S=8 cores, a bandwidth of 4 bytes or words/cycle, with an on-chip memory size of 13 Mbytes, and a bandwidth

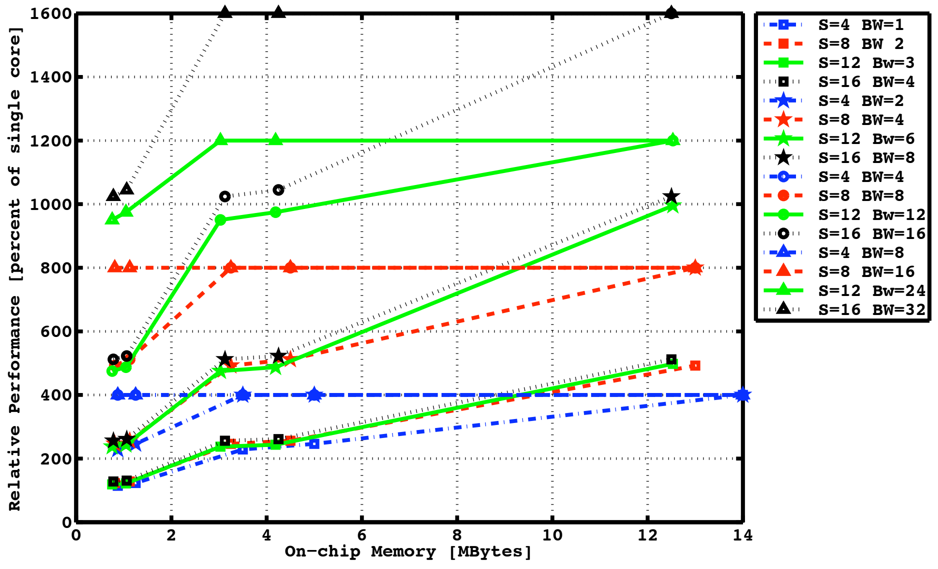

Figure 7: LAP performance for different on-chip memory sizes, different number of cores, and different total on-chip bandwidths with $n_r = 4$ and s=4, 8, 12, 16.

of 8 bytes/cycle with an with on-chip memory size 2.5 MBytes are both optimal design points.

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

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

Finally, we analyze the tradeoff between the size of the on-chip memory and the external, off-chip bandwidth. We assume that the problem size and number of cores are fixed, and initially the optimal local store size is allocated in the cores and PEs on the chip. Next, we shrink the available on-chip memory and compute the external bandwidth demands to keep the performance over 90%. The algorithmic solution to this problem is adding another layer of blocking as shown in Figure 8. The matrix dimension of the original problem size is is n and the new block size is n_s. We call this ratio $d = \frac{n}{n_s}$. After shrinking the available on-chip memory, the solution assumes that a single (Figure 8-(a)) or $k \leq d$ (Figure 8-(b,c) k=d) sub-blocks of the original matrix C can fit on the new on-chip memory. Then, the algorithm performs all operations and data movements necessary to compute these k sub-blocks of C. The new off-chip bandwidth for the new smaller on-chip memory and a sub-problem size $k \times (n_s \times n_s)$ as part of the original $n \times n$ matrix multiplication can be computed as

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

Figure 9 shows the external bandwidth demands for three different problem sizes and how the increase as the size of on-chip memory decreases. With growing original problem sizes $n \times n$, for the same on-chip memory size, the external bandwidth drops. We observe that as the original problem

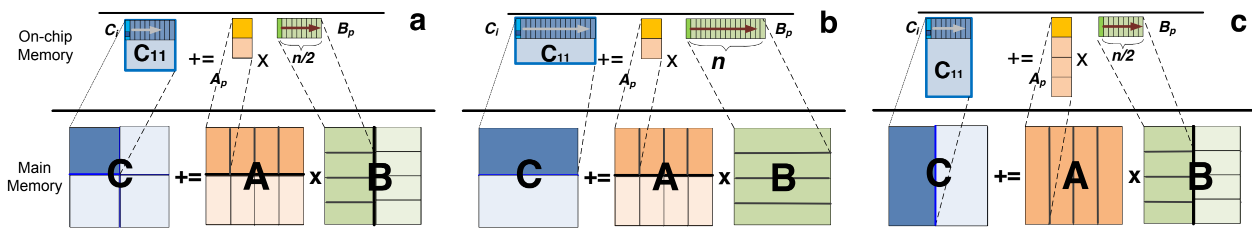

Figure 8: Blocking algorithm to map a big problem on a small on-chip memory. a) blocking for quarter size b,c)blocking for half size.

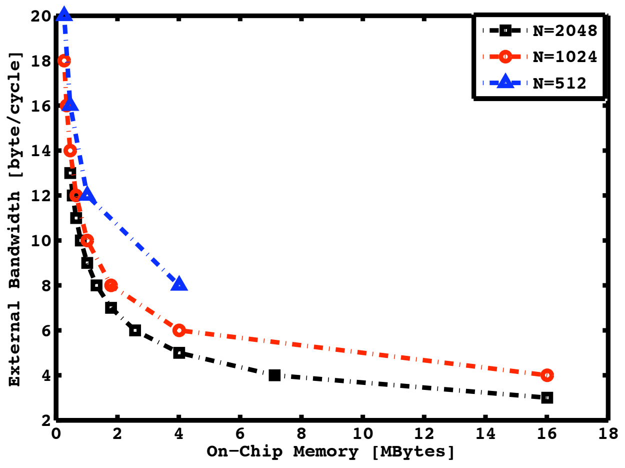

Figure 9: External Bandwidth vs. Size of on-chip memory tradeoff for different original problem sizes. All utilization numbers are over 92%.

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

Figure 10 summarizes overall performance of a 1.4GHz LAP as a function of the size of the on-chip memory (dictating the possible kernel size), the number of cores, and the external bandwidth to the off-chip memory. Here we use $n_r = 4$, $m_c = k_c$ (the submatrix $A_{i,p}$ is square) and $n = 256, 512, 768$ or 1024 as the dimension of matrix C (kernel size, which translates into a corresponding on-chip memory size). As we increase the available core parallelism, the needed off-chip bandwidth increases for the same problem size[2]. Also when problem size grows, with same off-chip bandwidth we get better performance. This graph shows that a small L2 memory size, e.g. as is the case in GPUs, which determines the possible on-chip problem size, limits the achievable peak utilization (”exploitable parallelism”). Overall, with 16 cores, 5 Mbytes of shared on-chip memory and an external bandwidth of 16B/cycle, we can achieve 600 GFLOPS.

4.3 Model Validation and Comparative Performance Prediction

The analytical models that we presented so far can help designers verify performance and utilization of their architecture for class of matrix operations in the early stages of the design process. In this section, we demonstrate the benefits and feasibility of our analytical models for early performance

[2]Note that the needed on-chip memory size also increases slightly due to additional storage required for prefetching across more cores.

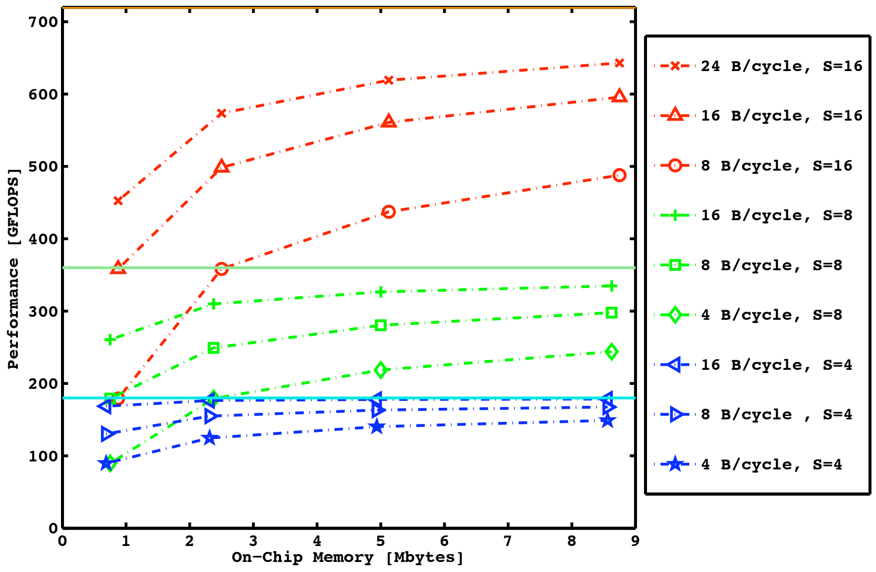

Figure 10: LAP performance as a function of external off-chip bandwidth and the size of on-chip memory with $n_r = 4$, $m_c = k_c$.

prediction by using them to discuss common sources of performance waste in existing architectures, and we specifically study examples of state-of-the-art GPU and other accelerated architectures.

There are two common limitations in parallel architectures that restrict their performance and efficiency. First, the core architectural and micro-architectural features can limit the accesses to local register files and number of instructions executed in each cycle. Second, the memory hierarchy organization that include sizes of layers and bandwidths between them might not be able to sustain data movement from/to the computation cores.In the following, we assume that the cores are perfectly designed. The main metric affected by core-level design issues is the achievable peak efficiency in terms of both energy spent per operation (GFLOPS/W) and achievable utilization. We have shown how to design such ideal core in Section 3. A further study of core-level micro-architectural tradeoffs is outside of the scope of this paper. Instead we focus on analysis of the memory hierarchy. The main efficiency metric affected by the memory hierarchy trade-off is achievable utilization. In the following we will specifically show how we can apply our analytical memory hierarchy model to predict limitations in Nvidia's Fermi and Clearspeed's CSX architectures.

The Nvidia Fermi C2050 architecture has 14 cores with 16 double precision MAC units in each core. The size of the onchip cache is 728 KBytes. The clock frequency is 1.15 GHz. Let us assume that cores are designed to achieve up to peak performance. With 728 KBytes, the dimension of the block of matrix C that fits in the onchip memory is $n_s = 256$ filling 512 KBytes of onchip L2 cache. Dividing the block C into row panels among the 14 cores results in $m_c = n_s/S = 256/14 = 16$. Hence, the size of each row panel of C is $m_c \times n_s = 16 \times 256$. Thus, the parameters of the design are as follows: $m_c = k_c = 16, S = 14, n_s = 256$. Assuming full overlapping, the maximum required off-chip bandwidth according to Table 1 is $(\frac{4\times14\times4^2}{256})\times$1.15GHz$\times$8Bytes= 32GBytes/second, which is within the 144 GBytes/ that Fermi offers. The on-chip required bandwidth is $(\frac{2S}{k_c}+\frac{S}{m_c})n_r^2 = (\frac{2\times14}{16}+\frac{14}{16})4^2\times$ 1.15GHz$\times$8Bytes= 386GBytes/second, which is much more than the 230 GBytes/second that Fermi offers. To calculate theoretically achievable utilization using such a configuration, we divide the available bandwidth by the demanded bandwidth: 230/386 = 60%. In reality, implementations of GEMM on C2050 achieve 58% [59] of peak performance. Hence, our model accurately predicts that the on-chip bandwidth of Fermi does not overcome the needs of matrix multiplication. We can over come this underutilization by increasing the on-chip bandwidth (see above), or by increase the on-chip memory size. If the size of on-chip memory is doubled in the previous case, the required on-chip bandwidth can drop to half or 200 GBytes/second using the solution in Figure 9-c.

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

5 LAP Implementation

We have developed both simulation and analytical power and performance models of the LAP in comparison with other architectures. The analytical performance model was presented in previous sections, and we will describe our power model next. In addition, we validated the performance model and LAP operation in general by developing a cycle-accurate LAP simulator. The simulator is configurable in terms of PE pipeline stages, bus latencies, and memory and register file sizes. Furthermore, by plugging in power consumption numbers for MAC units, memories, register files and 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.

5.1 Component Selection

To investigate and demonstrate the performance and power benefits of the LAP, we have studied the feasibility of a LAP implementation in current bulk CMOS technology using publicly available components and their characteristics as published in the literature.
State-of-the-art implementations of Fused Multiply Add (FMA) units use various optimization techniques to reduce latency, area and power consumption [63]. Fused Multiply Accumulate (FMAC) units with delayed normalization achieve a throughput of one accumulation per cycle [72, 71]and save around 15% of total power [36]. The number of pipeline stages typically ranges between 5 and 9and the same FPMAC units can be reconfigured to perform either integer, single-, or double-precision operations [67]. 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 [25]. For our analysis, we use the same data. 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.

Our design utilizes around SRAM with no tags and no associativity. Given the sequential nature of access patterns to 64-bit wide double-precision numbers, we carefully selected memories

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

Table 2: 45nm scaled performance and area for a LAP PE with 16KBytes of dual-ported SRAM.

with one or two banks to minimize power consumption. Using CACTI [65] with low-power ITRS models and aggressive interconnect projection, we obtained area estimates of around $0.13mm^2$ and we calculated the dynamic power of the local SRAM at frequencies over 2.5 GHz to be around 13.5mW per port. For the overall system estimation (see Section 5.4), 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.

To estimate latencies and power consumption of row and column busses, we use data reported in CACTI. 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. 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. However, per PE we only have $2/n_r$ of the power consumption of a single bus. CACTI reports three different classes of wires (fast local, semi-global, and global) for different layers of the memory hierarchy. For intra-core communication, we assume fast local wires. For wires with 30% overhead, the distance between repeaters is a maximum of more than 1.62mm. According to our area estimates, each PE will not be wider than 0.4 mm. Hence, for $n_r = 4$, broadcast bus will not require any overhead (no wire repeaters and even less power consumption) compared to a point-to point connectivity. The wire model suggests that with any type of wire, we can reach over 2.2 GHz or over 1.4GHz bus frequency on the broadcast bus for nr = 4,8 or $n_r = 16$, respectively. The area of the bus per PE is 0.023 mm^2 and the worst case bus power is negligible.

Overall area, power and performance estimates for our PE design at various operating points are summarized in Table 2. Running at a clock frequency of 1 GHz, a 4×4 LAP core is estimated to achieve an efficiency of 110 single-precision or 45 double-precision GFLOPS/W. We stress that the point of this section is not to present the ultimate design.

To find the best combination of components and the best operating frequency we used energy-delay W/$GFlops^2$ [28], as well as GFLOPS/W and GFLOPS/mm^2 efficiency metrics. The best design choice has a lower energy-delay value and maintains high efficiency. Figure 11-(left) shows the power/throughput and the energy-delay for different PE frequencies. At 1.8 GHz there is no much deduction in energy-delay while power/throughput increases significantly. At the left side of the spectrum low frequency designs have high efficiency but with high energy-delay and low area efficiency. A good tradeoff is achieved at a frequency of around 1 GHz, where energy-delay is still decreasing and there is high area and power efficiency. Figure 11-(right) shows the trade-off between area/throughput, power/throughput and energy-delay. Low frequency designs are on the right side of spectrum. At 1 GHz, more than twice the area efficiency and energy-delay (0.1 mm^2/GFlop and 10 mW/$GFLOPS^2$) is achieved when compared to a design at 0.3 GHz. Also, compared to 1.8 GHz core, while having almost the same energy-delay, the power efficiency is 40% better.

5.2 Power Modeling of Architectures

We developed a general analytical power model that builds on existing component models (e.g. for FPUs and memories) described in the previous section. The model is derived from methods described in [55, 8] and we applied it to both our LAP and various existing architectures. Our power model computes the total power as the sum of the dynamic power and idle power over all components in the architecture.

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

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

We calibrated our power model and its parameters against power and performance numbers presented for the NVidia GTX280 Tesla GP-GPU running matrix multiplication [34, 77]. We used the sizes of different GPU memory levels reported in [77] together with numbers from [34] and [3] to match logic-level, FPU, CACTI and leakage parameters and factors in order to achieve consistent results across published work and our model. We then applied this model to other architectures, such as the NVidia GTX480 Fermi GP-GPU [1, 40] or the Intel Penryn [27] dual-core processor. To the best of our knowledge, there are no detailed power models yet for these architectures. We adapted our model to the architectural details as far as reported in literature using calibrated numbers for basic components such as scalar logic, FPUs or various memory layers. In all cases, we performed sanity checks to ensure that total power numbers match reported numbers in literature.

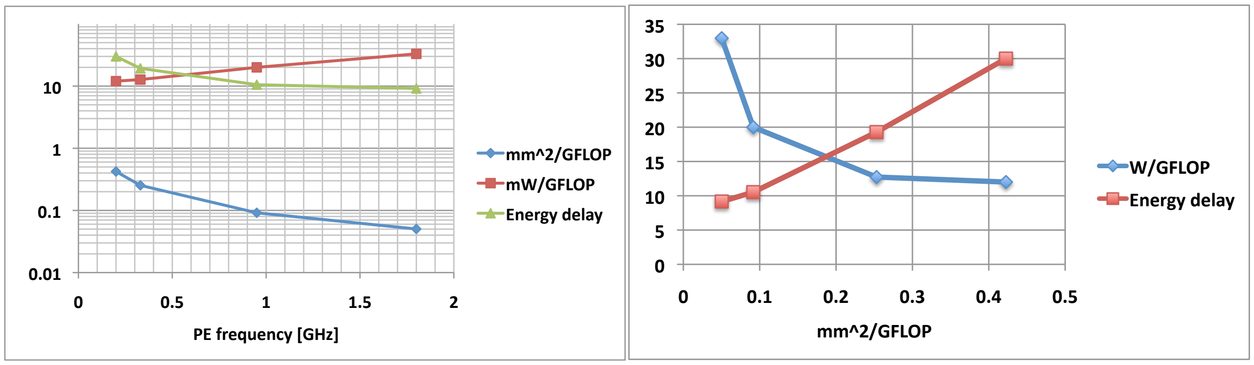

Figure 11: Efficiency metrics of PE (left), and power efficiency and energy-delay vs. area efficiency at different frequencies (right).

5.3 Power and Area Exploration

In this section, we use power and area models to study the design space that we created in Section 4. We explore various trade-offs and how each design feature can affect the power and area consumption of the whole system. We use analytical results from Section 4 and apply representative power and area numbers to each point in the design space. This will allow us to evaluate how size and bandwidth of different layers of the memory hierarchy affect the overall performance and efficiency of the design.

At the core level, the goal is to have enough bandwidth and local store to maintain peak performance (equivalent to Figure 4) . We select the size of the core to be $n_r = 4$, and show the core-level area and power consumptions. Figure 12-(left) illustrates the area of different components within PE. With a local store size of 18 KByte, the local store occupies at most 2/3 of the PE, which exhibits a a linear relation to the local store capacity size. The power/throughput ratio of the PE, the local store, and the total leakage is shown in Figure 12-(right). The graph suggests that with smaller local stores and even with higher bandwidths still less power is consumed in each PE. The overall PE power consumption is dominated by the FPU. These graphs advocate smaller local store sizes in terms of power and area consumption. However, there are three reasons that force larger PE local stores. First, the power density increases if local store size is reduced, which may limit the overall performance. Second, although decreasing the local store size does not affect the core power consumption, the on-chip bandwidth will increase exponentially, which decreases the utilization and also results in a significant increase of the total power consumption. Finally, for algorithms like Cholesky factorization in which all the data is in-core, a bigger local store per PE yields to the ability of handling bigger kernels and amortizing more of the irregular computations over the available parallelism.

At the chip level, we estimate the effect of on-chip memory size on overall power and area while maintaining peak performance (similar to Figure 9). For each on-chip memory size, there are different options in terms of core configuration. We choose the biggest possible local store size to minimize intra-chip traffic and hence power consumption. Here, the power consumption due to external accesses is not included. Figure 13-(left) shows the area consumption of the cores and on-chip memory. Figure 13-(right) shows that with our domain specific design of on-chip SRAM memory almost all of the power of the chip is used by the eight cores and memory trade-offs are negligible.

In order to get a better sense of memory trade-offs in more general systems, we performed the same analysis using the NUCA [58] memory simulator of CACTI and replacing the SRAM design by Nuca caches. Here, the effects of increased bandwidth with smaller memory sizes are seen more realistically. In our LAP design, we use single-ported memory banks in low-power technology and with low clock frequencies. In a Nuca cache based design, either multi-ported caches or high-performance, high-power banks have to be used to maintain the same high bandwidths at small memory sizes. We chose high-performance, high-power caches since they require less area and power compared to multi-ported designs. As shown in Figure 14-(left), in all cases the on-chip Nuca memory occupies more space than the computation cores do. Furthermore, a design with small capacity, high bandwidth banks ends up occupying more space than a larger, slower on-chip memory. Higher bandwidth also affects the power consumption of the system. Figure 14-(right) shows that at lower capacities, on-chip Nuca memory consumes more power than the computation cores. In other words, a design with larger simpler on-chip Nuca cache size is both more power and

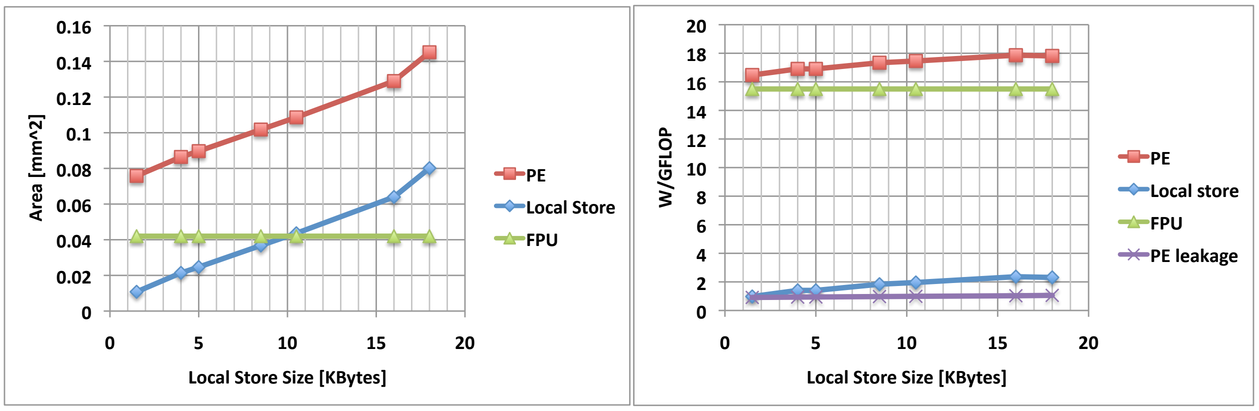

Figure 12: Area of a single PE in a 4x4 core for different local store sizes (left), and leakage, local store, and total power efficiency of a PE at in a 4x4 core at 45nm (right).

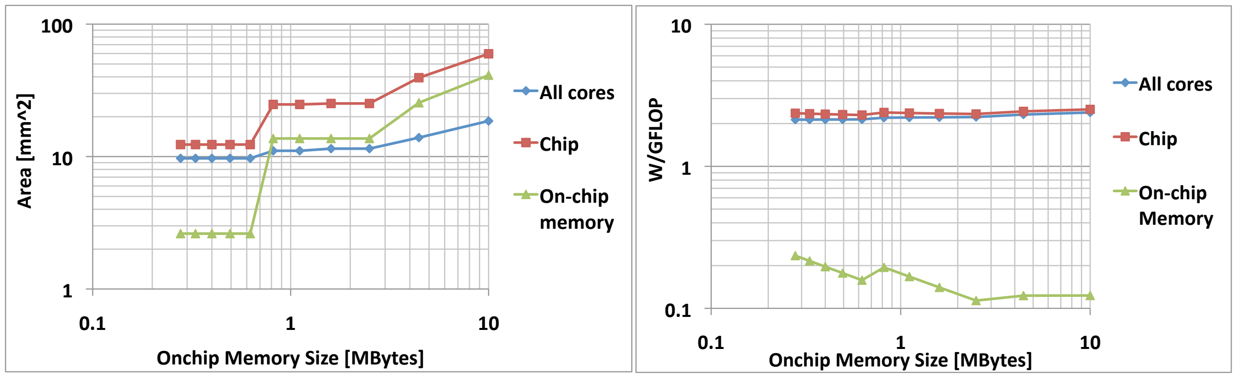

Figure 13: Area (left) and power efficiency (right) of cores, on-chip memory and a total 128 MAC unit system with S=8 4x4 cores, different on-chip SRAM memory sizes, and n=2048.

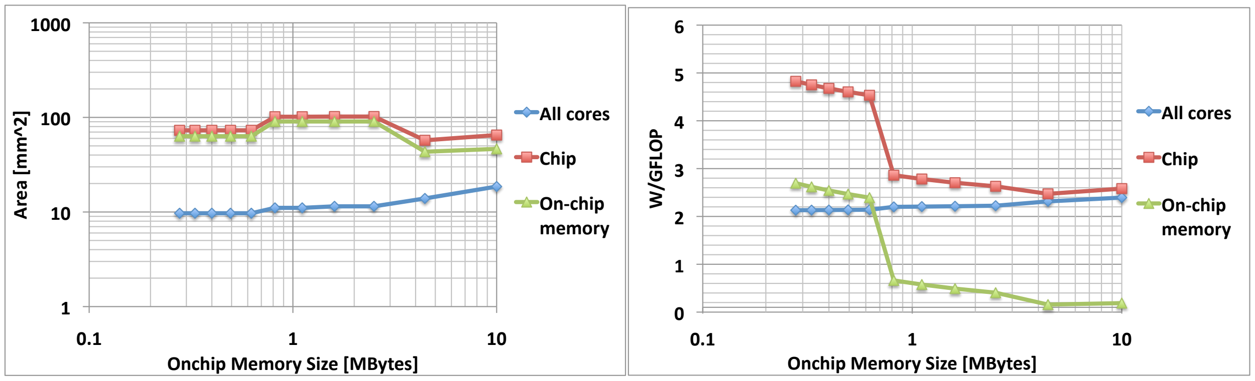

Figure 14: Area (left) and power efficiency (right) of cores, on-chip memory and a total 128 MAC unit system with S=8 4x4 cores, different on-chip Nuca memory sizes, and n=2048.

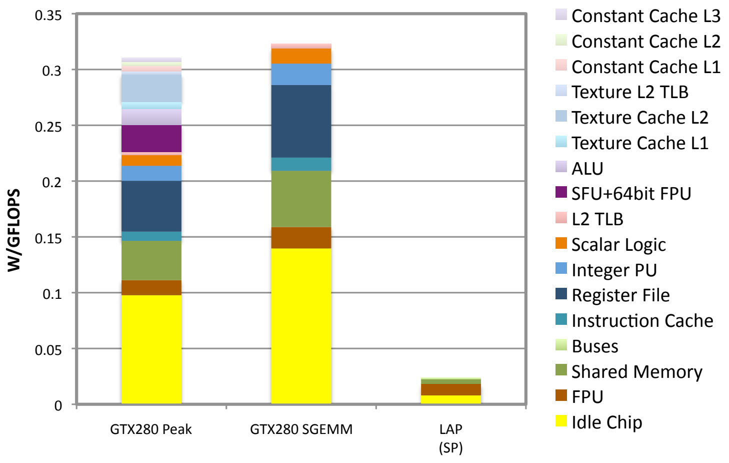

Figure 15: Normalized power breakdown of Nvidia Tesla GTX280 versus LAP at 65nm.

more area efficient.

5.4 Comparative Power and Performance Analysis

Figure 15, 16, and Figure 17 show a breakdown of performance-normalized power consumption for current high-performance GP-GPU and multi-core architectures as compared to single- or double-precision versions of a prototypical LAP with an equivalent number of cores (i.e. Shared Multiprocessors, SMs, in GPUs[3]) running at equivalent raw single FMAC performance (1.3GHz or 1.4GHz). In the case of GPUs (Figures 15 and 16), we show efficiencies for both peak operation and when running GEMM. Current GPUs run single- or double- precision GEMM (SGEMM or DGEMM) at only around 60% of their theoretical peak FPU performance [6, 74, 59]. As the graphs show, reduced utilization has a significant effect on achievable efficiencies, even when considering that unneeded components, such as constant caches, texture caches, extra ALUs or special functional units (SFUs) can be turned off. By contrast, the Intel Penryn dual-core processor and a LAP with two 4×4 cores running at 1.4GHz, i.e. at around half of the Penryn's 2.66GHz clock speed, achieve near peak utilization at a moderate performance of 20 and 90 double- precision GFLOPS, respectively (Figure 17).

Breakdowns show that traditional architectures include significant overhead. The only units that are really useful for performing matrix multiplication are FPUs/execution units, shared memories/L1 caches, L2 caches and TLBs. In the GPUs, components like shared memories, instruction caches or register files can consume up to 70% of the power, and in some cases the register file alone contributes more than 30%. By eliminating instructions, associated cache power is removed from the LAP. Similarly, register files are very small and shared memories are replaced by sequentially accessed, partitioned SRAM with a maximum of 2 read/write ports. For the Penryn, we mainly relied on the power breakdown presented in [27], where we assumed that GEMM utilizes all of the core. In the graph, the SRAMs and MACs of the LAP are listed under the MMU and execution unit categories. We conservatively added all of the miscellaneous and IO power consumption factors to the LAP, which favors the Penryn in this comparison. We can observe that the Penryn uses 40% of the core power (over 5 W) in the Out of Order and Frontend units that do not exist in LAP

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

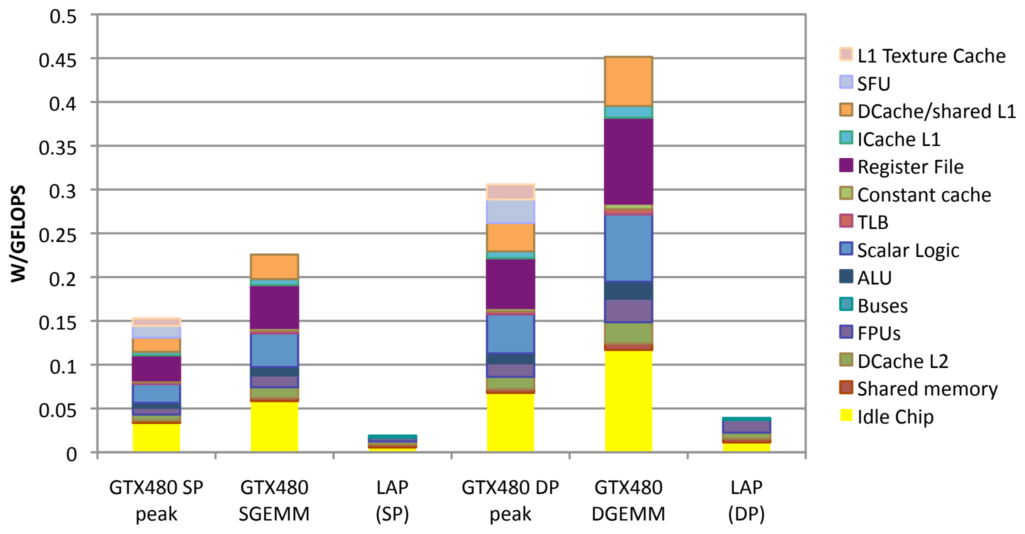

Figure 16: Normalized power breakdown of Nvidia Fermi GTX480 versus LAP at 45nm.

architecture. Furthermore, with around 5 W the execution unit consumes one third of the core power, which may be attributed to the fact that FPUs are fully IEEE-754 compatible and support the full range of exception handling.

Overall, some of the major differences between traditional general-purpose designs and a specialized linear-algebra architecture lie in the memory architecture and the core execution unit datapaths. The LAP has relatively large L1- and L2-equivalent PE and on-chip memories, comparable in size to multi-core architectures but an order of magnitude bigger than in GPUs. This keeps bandwidth between memory layers low. All memories are pure, banked SRAMs with no tagging or cache consistency overhead. Consequently, memories are more power efficient and smaller than in other architectures despite being larger. Shared on-chip memory can be partitioned among groups of cores with each bank being only coupled with its set of cores. Note that we do not include external memory in our analysis. With system architectures increasingly integrating host processors and accelerators on a single die, we can expect similar benefits to extend into other such memory layers. Again, larger on-chip memories in the LAP help to decrease external memory bandwidth and power consumption requirements.

For execution units and data paths, we can observe that unnecessary overheads are removed by performing whole chains of operations in local accumulators without any register file moves that become necessary in traditional SIMD arrangements. This is further confirmed by low GEMM utilizations, which indicates that despite existing architectural features, idiosyncrasies of traditional architectures make it difficult to keep a large number of FPUs busy. Overall, the 2D PE arrangement with local, partitioned memory is scalable with exponential growth in compuation power for a linear growth in interconnect and bus lengths. With relatively low overhead for specialized MAC units and broadcast busses, we can envision such specialized data paths to be integrated into standard processor pipelines for order of magnitude improved efficiency in a linear algebra computation mode. Table 3 summarizes the differences discussed in this section.

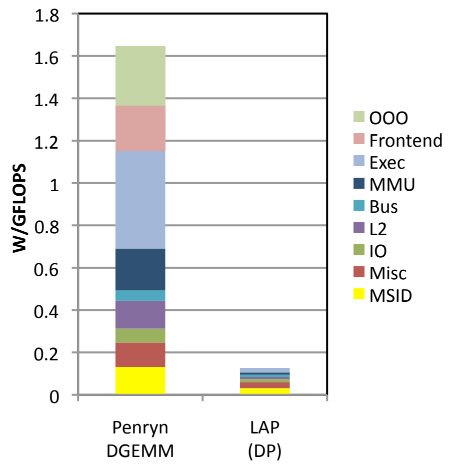

Figure 17: Normalized power breakdown of Intel dual-core Penryn versus LAP at 45nm.

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

Table 3: Comparison between main design choices in the studied platforms.

5.5 Summary Comparison

We compare overall efficiency and inverse energy-delay [28] of single- and double-precision realizations of our design against other systems. Figure 18 shows an analysis of core- and chip-level efficiencies for studied architectures and a LAP in which we vary the number of cores to match the throughput in existing architectures. Our LAP with 30 single- or 15 double-precision cores and 5Mbytes of on-chip memory achieves a GEMM performance of 1200 and 600 GFLOPS at a utilization of 90% in an area of 115 mm^2 or 120 mm^2, respectively. By comparison, the dual-core CPU achieves 22 GFLOPS in 100mm^2 and the GTX480 runs SGEMM/DGEMM with 780/390 GFLOPS and 58% utilization using 15 SMs in total 500mm^2 chip area.

Finally, Table 4 summarizes key metrics for various systems running GEMM as a representative matrix computation. For this table, we extended the analysis presented in [41] by including estimates for our LAP design, the 80-tile network-on-chip architecture from [72], the Power7 processor [75], the Cell processor [52], Intel Penryn [27], Intel Core i7-960 [12], CSX700 [4], Altera Stratix IV [60], and the NVidia Fermi GPU (GTX480) [40] all scaled to 45nm technology and to GEMM utilizations.

We note that for a single-precision LAP at around 1.4GHz clock frequency, the estimated performance/power ratio is an order of magnitude better than GPUs. The double-precision LAP

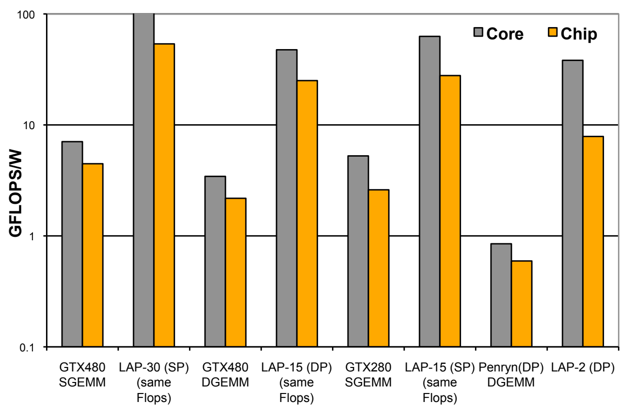

Figure 18: Comparison of efficiencies for single- and double-precision GEMM between NVidia Tesla GTX280, NVidia Fermi GTX480, Intel Penry and a LAP of equivalent throughput.

Architecture	$\frac{\text{GFLOPS}}{\text{s}}$	$\frac{\text{W}}{\text{mm}^2}$	$\frac{\text{GFLOPS}}{\text{mm}^2}$	$\frac{\text{GFLOPS}}{\text{W}}$	$\frac{\text{GFLOPS}^2}{\text{W}}$	Utilization
Cell	200	0.3	1.5	5.0	1000	88%
Nvidia GTX280	410	0.3	0.8	2.6	1066	66%
Rigel	850	0.3	3.2	10.7	9095	40%
80-Tile @0.8V	175	0.2	1.2	6.6	1155	38%
80-Tile @1.07V	380	0.7	2.66	3.8	1444	38%
Nvidia GTX480	780	0.2	0.9	4.5	3510	58%
Core i7-960	96	0.4	0.50	1.14	109.44	95%
Altera Stratix IV	200	0.02	0.1	7	1400	90+%
LAP (SP)	1200	0.2	6-11	30-55	66000	90+%
Intel Quad-Core	40	0.5	0.4	0.8	32	95%
Intel Penryn	20	0.4	0.2	0.6	12	95%
IBM Power7	230	0.5	0.5	1.0	230	95%
Nvidia GTX480	390	0.2	0.4	2.2	858	58%
Core i7-960	48	0.4	0.25	0.57	27.36	95%
Altera Stratix IV	100	0.02	0.05	3.5	350	90+%
ClearSpeed CSX700	75	0.02	0.2	12.5	937.7	78+%
LAP (DP)	600	0.2	3-5	15-25	15000	90+%

Table 4: 45nm scaled performance and area of various systems running GEMM.

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

6 Conclusions and Outlook

This paper provides initial evidence regarding the benefits of customized architectures for linear algebra computations. As had been postulated [32], one to two orders of magnitude improvement in power and performance density can be achieved. We now discuss possible extensions.

For example, Figures 3 and 10 clearly show the tradeoff between the sizes of the local and onchip memories, and their corresponding bandwidth to onchip and offchip 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. Similarly, the choice of the size of the PE array, $n_r = 4$ 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 or by further extending interconnect into a flat on-chip network (NoC) of PEs that can be dynamically configured and partitioned into clusters of cores of variable sizes.

So far, we modeled the power consumption for our design and its competitors. The next step is to further expand our analysis to the area and complexity breakdown for these architectures. This will help designers take into account both area and power budgets as one of the main concerns in the future is the power density. We also plan to extend our cycle accurate simulator into a full LAP system simulator and, if possible, integrate it to other muli-core simulators to study detailed design tradeoffs both at the core and chip level. This integration will allow cycle-accurate modeling of dynamic power consumption of different design choices.

The GEMM operation is in and by itself important. It indirectly enables high performance for the level-3 Basic Linear Algebra Subprograms (BLAS) [13, 39] as well as most important operations in packages like LAPACK [7] and `libflame` [70]. 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 [29], 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 architecture can be generalized to accommodate this class of operations.

References

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

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

[3] Samsung DDR3 SDRAM:High-Performance, Energy-Efficient Memory for Today's Green Computing Platforms. Technical report, SAMSUNG Green Memory, March 2009.

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

[5] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM J. Res. Dev.*, 38:673–681, November 1994.

[6] 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.

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

[8] D. Brooks et al. Wattch: a framework for architectural-level power analysis and optimizations. *ISCA, 2000.*, pages 83 – 94, 2000.

[9] L. E. Cannon. *A cellular computer to implement the kalman filter algorithm.* PhD thesis, Bozeman, MT, USA, 1969. AAI7010025.

[10] 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.

[11] J. Choi, D. W. Walker, and J. J. Dongarra. Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, 6(7):543–570, 1994.

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

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

[14] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

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

[16] V. Eijkhout. *Introduction to High Performance Scientific Computing.* `www.lulu.com`, 2011.

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

[18] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: a vector extension to the alpha architecture. *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 281 – 292, 2002.

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

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

[21] R. Falgout, A. Skjellum, S. Smith, and C. Still. The multicomputer toolbox approach to concurrent blas and lacs. In *Scalable High Performance Computing Conference, 1992. SHPCC-92. Proceedings.*, pages 121 –128, apr 1992.

[22] R. D. Falgout, A. Skjellum, S. G. Smith, and C. H. Still. The multicomputer toolbox approach to concurrent blas. In *Proc. Scalable High Performance Computing Conf. (SHPCC*, pages 121–128. IEEE Press, 1993.

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

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

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

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

[27] V. George, S. Jahagirdar, C. Tong, et al. Penryn: 45-nm next generation intel® core$^{\text{TM}}$ 2 processor. *IEEE Asian Solid-State Circuits Conference*, Jan 2008.

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

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

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

[31] A. Gupta and V. Kumar. Scalability of parallel algorithms for matrix multiplication. *Parallel Processing, 1993. ICPP 1993. International Conference on*, 3:115 – 123, 1993.

[32] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. *ISCA '10*, Jun 2010.

[33] 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.

[34] S. Hong and H. Kim. An integrated GPU power and performance model. *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, Jun 2010.

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

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

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

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

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

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

[41] J. Kelm, D. Johnson, M. Johnson, N. Crago, W. Tuohy, A. Mahesri, S. Lumetta, M. Frank, and S. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. *ISCA '09*, Jun 2009.

[42] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 399 – 409, 2003.

[43] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 52 – 63, 2004.

[44] V. Kumar, S. Joshi, S. Patkar, and H. Narayanan. Fpga based high performance double-precision matrix multiplication. *VLSI Design, 2009 22nd International Conference on*, pages 341 – 346, 2009.

[45] V. Kumar and Y. Tsai. Synthesizing optimal family of linear systolic arrays for matrix computations. *Systolic Arrays, 1988., Proceedings of the International Conference on*, pages 51 – 60, 1988.

[46] V. Kumar and Y. Tsai. On synthesizing optimal family of linear systolic arrays for matrix multiplication. *Computers, IEEE Transactions on*, 40(6):770 – 774, 1991.

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

[48] S. Kung. Vlsi array processors. *ASSP Magazine, IEEE*, 2(3):4 – 22, jul 1985.

[49] H. Kungt. Systolic arrays (for vlsi). *Sparse matrix proceedings*, Jan 1979.

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

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

[52] F. Lauginiger et al. Performance of a multicore matrix multiplication library. *STMCS 2007,*, Jan 2007.

[53] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.

[54] J. Li. A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. *Citeseer*, Jan 1996.

[55] S. Li et al. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. *MICRO-42.*, 2009.

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

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

[58] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Architecting efficient interconnects for large caches with cacti 6.0. *IEEE Micro*, 28:69–79, January 2008.

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

[60] M. Parker. High-performance floating-point implementation using FPGAs. In *MILCOM*, 2009.

[61] M. Parker. Achieving teraflops performance with 28nm fpgas. *EDA Tech Forum*, December 2010.

[62] A. Pedram, A. Gerstlauer, and R. van de Geijn. A high-performance, low-power linear algebra core. In *22nd International Conference on Application-specific Systems, Architectures and Processors*, ASAP '11, pages 35–41. IEEE, 2011.

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

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

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

[66] C. Takahashi, M. Sato, D. Takahashi, T. Boku, A. Ukawa, H. Nakamura, H. Aoki, H. Sawamoto, and N. Sukegawa. Design and power performance evaluation of on-chip memory processor with arithmetic accelerators. *IWIA2008*, pages 51 – 57, 2008.

[67] 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.

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

[69] R. van de Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, April 1997.

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

[71] S. Vangal, Y. Hoskote, N. 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.

[72] 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.

[73] P. Varman and I. Ramakrishnan. Synthesis of an optimal family of matrix multiplication algorithms on linear arrays. *Computers, IEEE Transactions on*, C-35(11):989 –996, nov. 1986.

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

[75] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. C. Rubio, F. Rawson, and J. B. Carter. Architecting for power management: The IBM® POWER7^{TM} approach. *HPCA 2010*, pages 1 – 11, 2010.

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

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

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

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

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