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

Ardavan Pedram, Robert A. van de Geijn *Member, IEEE*, and Andreas Gerstlauer *Senior Member, IEEE*

Abstract—As technology is reaching physical limits, reducing power consumption is a 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 Subprograms. 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 operations. In addition to exposing the sources of inefficiencies in current CPUs and GPUs, our results show our prototype linear algebra processor implementing double-precision GEMM can achieve 600 GFLOPS while consuming less than 25 Watts in standard 45nm technology, which is up to $50\times$ more energy efficient than cutting-edge CPUs.

Index Terms—Low-power design, Energy-aware systems, Performance analysis and design aids, Matrix multiplication, Memory hierarchy, Level-3 BLAS, Special-purpose hardware

1 Introduction

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 for specialization. In particular, it is now likely that a future chip may combine heterogeneous cores while having to cope with "dark silicon" [1]. 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 accelerators can provide orders of magnitude improvements in efficiencies for a wide variety of application domains [2], [3]. However, full custom design is expensive in many aspects. Hence, the question is whether such techniques can be applied to a broader class of 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 this question for the domain of matrix computations, which resides at the core of many applications in scientific, high-performance computing. It is well understood that many linear algebra problems can be efficiently reduced down to a canonical set of Basic Linear Algebra Subprograms (BLAS), such as matrix-matrix and matrix-vector operations [4], [5]. Highly efficient realizations 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 multiplication (GEMM) [6]. This operation is the building block for other matrix-matrix operations (level-3 BLAS) [7]. In [8], 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 level-2 BLAS as well as for operations supported by the Linear Algebra PaCKage (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 next to

- *Ardavan Pedram, and Andreas Gerstlauer are with The Department of Electrical and Computer Engineering, The University of Texas at Austin,Austin, Texas.*
  E-mail: ardavan@utexas.edu, gerstl@ece.utexas.edu
- *Robert van de Geijn is with The Department of Computer Science, The University of Texas at Austin, Austin, Texas*
  E-mail:rvdg@cs.utexas.edu

This research was partially sponsored by NSF grants OCI-0850750 and CCF-1018075, and an endowment from AMD

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, we introduced a custom micro-architecture design for a Linear Algebra Core (LAC) [9]. In this paper, we extend the LAC design with a more general memory hierarchy model to evaluate different trade-offs in system design, including the number of cores, the bandwidth between the 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 core level efficiencies of 45 double- and 110 single-precision GFLOPS/Watt in 11-13 GFLOPS/mm^2 with currently available components and technologies, as published in literature [10], [11]. This represents a 50-fold improvement over current off-the-shelf, desktop- or server-class CPUs and an order of magnitude improvement over current commercial GPUs.

The rest of the paper is organized as follows: In the next section we briefly discuss related work. Section 3 provides a review of the GEMM algorithm and the matrix processor core microarchitecture of our design. In Section 4, we build the memory hierarchy around such cores, show the mapping of matrix multiplication onto a multi-core system, and analyze the existing trade-offs in the design space. 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. Details of formulae derivations in Section 3 and Section 4 are discussed in the Appendix.

2 Related Work

Implementation of GEMM on traditional general-purpose architectures has received a lot of attention. Modern CPUs exploit vector extension units for high performance matrix computations [8], [12], [13]. However, general instruction handling overhead remains. Three main limitations of conventional vector architectures are known to be the complexity of the central register file [14], implementation difficulties of precise exception handling, and expensive on-chip memory [15].

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 [16]. More recently, GPUs have shifted back towards general-purpose architectures. Such GPGPUs replicate a large number of Single Instruction Multiple Data (SIMD) processors on a single shared-memory chip. GPGPUs can be effectively used for matrix computations [17], [18] with throughputs of more than 300 GFLOPS for Single-precision GEMM (SGEMM), utilizing around 30-60% of the theoretical peak performance. In the latest GPGPUs, two single-precision units can be configured as one double-precision unit, achieving more than 700 single-precision and 350 double-precision GFLOPS at around 70% utilization [19] for matrices larger than 512×512.

Over the years, many other parallel architectures for high-performance computing have been proposed and in many cases benchmarked using GEMM as a prototypical application. Systolic arrays were popularized in the 1980s. Different optimizations and algorithms for matrix multiplication and more complicated matrix computations have been compared and implemented on both 1D [20], [21] and 2D systolic arrays [20], [22], [23]. In [24], 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 [25], [26]. 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 [27] utilizes a dedicated network-on-chip interconnect with associated routing flexibility and overhead. This architecture 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 GFLOPS for double-precision GEMM (DGEMM) at 78% of its theoretical peak [28].

As utilization numbers indicate in general-purpose cases, inherent characteristics of data paths and interconnects coupled with associated instruction inefficiencies make it difficult to fully exploit all available parallelism and locality. By contrast, while we will build on the SIMD and GPU concept of massive parallelism, we aim to provide a natural extension that leverages the specifics of matrix operations.

In the domain of custom design, recent Field Programmable Gate Arrays (FPGAs) [29], [30] have moved towards 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 in 40nm technology can achieve at most 100 GFLOPS per-

formance at 7 GFLOPS/Watt of power efficiency [31]. Specialized hardware realizations of GEMM and other BLAS routines on FPGAs have been explored, either as standalone hardware implementations [32], [33] or in combination with a flexible host architecture [34]. Such approaches show promising results (up to 99% utilization), but are limited by the performance and size restrictions in FPGAs [35], [36].

Existing solutions for dedicated realization of matrix operations primarily focus on 1D and 2D arrangements of Processing Elements (PEs) [22]. In early FPGA designs with limited logic blocks on the chip, most of the approaches targeted an array arrangement that pipelines the data in and out of the PEs [37], [33]. More recently, with sufficient area on the chip, the design choice between 1D and 2D arrangement of PEs once again becomes relevant. There are three major benefits in a 2D versus a 1D solution: scalability, addressing, and data movement. The 2D arrangement has proven to be scalable with regard to the ratio of problem size to local store memory size for level-2 and -3 BLAS operations [38]. Furthermore, address computations and data accesses in local stores of PEs become 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.

A taxonomy of matrix multiplication algorithms on 2D grids of PEs and their interconnect requirements is presented in [39]. The flexibility of broadcast-based SUMMA algorithms has made them the most practical solution for distributed memory systems [40] and FPGAs [35]. This class of algorithms is the basis for our design.

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 computational core that is a highly optimized matrix multiplier [9] and then 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 Linear Algebra Core (LAC) Design

In this section, we briefly review our design for a LAC, as published in [9] and illustrated in Figure 1. It consists of a 2D array of $n_r \times n_r$ PEs, with $n_r = 4$ in the figure. Each PE has a Multiply-ACcumulate (MAC) unit with a local accumulator, and local Static Random-Access Memory (SRAM) storage, bus interfaces to communicate data within rows and columns. LAC control is distributed and each PE has a state machine that drives a predetermined, hardcoded sequence of communication, storage, and computation steps for each supported BLAS operation.

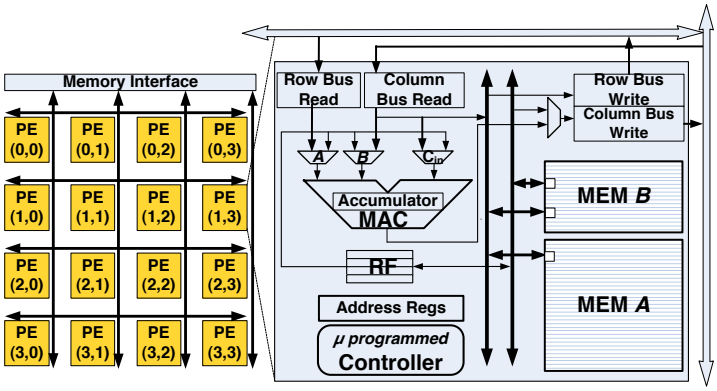

Fig. 1. Core architecture. PEs that own the current column of $4 \times k_c$ matrix A and the current row of $k_c \times 4$ matrix B, write elements of A and B to the buses and the other PEs read them.

In the following, we start by generally describing how a GEMM algorithm is implemented and blocked across different layers of a memory hierarchy, which will culminate in a discussion of how, in our case, the innermost GEMM kernel is mapped onto our LAC architecture.

3.1 GEMM Operation

In designing a complete linear algebra realization, we not only need to optimize the core kernels, 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.

Assume the square $n \times n$ matrices A, B and C are stored in external memory. In our discussions, upper case letters denote (sub)matrices while Greek lower case letters denote scalars. 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):

$$C \mathrel{+}= \begin{pmatrix} A_0 & \cdots & A_{K-1} \end{pmatrix} \begin{pmatrix} B_0 \\ \vdots \\ B_{K-1} \end{pmatrix} = \sum_{p=0}^{K-1} A_p B_p,$$

so that the main operation to be mapped to the Linear Algebra Processor (LAP) becomes $C \mathrel{+}= A_p B_p$, where A_p and B_p are of dimension $n \times k_c$ and $k_c \times n$, respectively. 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 the ith panel of C, C_i, must be updated by $C_i \mathrel{+}= A_{i,p} B_p$ as part of computing $C \mathrel{+}= A_p B_p$.

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 a 2D block cyclic round-robin

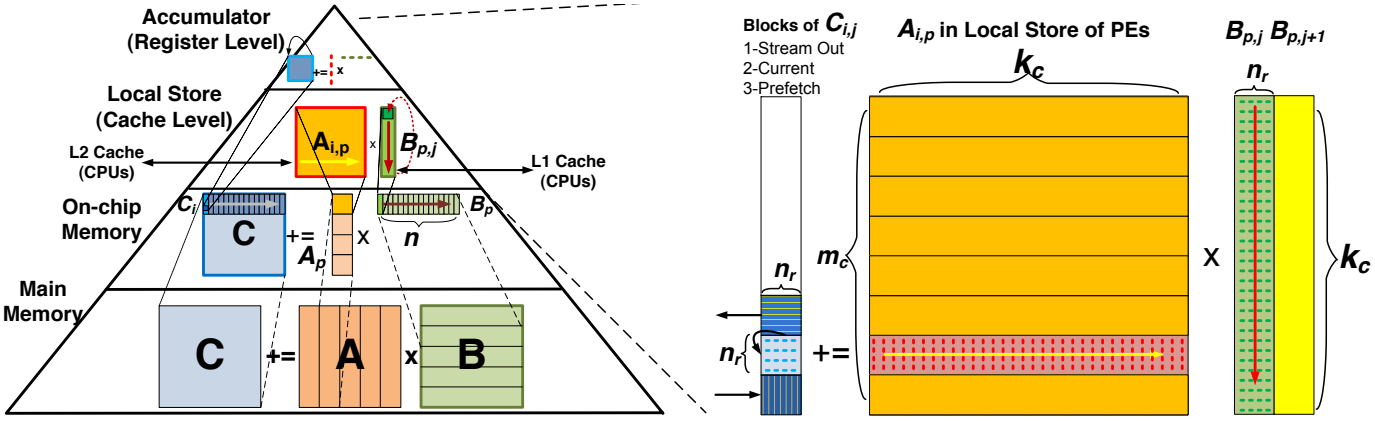

Fig. 2. Memory hierarchy while doing GEMM. In each of the top three layers of the pyramid, the largest matrix is resident, while the other matrices are streamed from the next layer down.

distribution. Then, partition C_i and B_p into panels of n_r 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 $C_i \mathrel{+}= A_{i,p}B_p$ requires the update $C_{i,j} \mathrel{+}= A_{i,p}B_{p,j}$ for all j. For each j, the $k_c \times n_r$ block $B_{p,j}$ is loaded into the local memories of the PEs. 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, we use a case of $n_r = 4$ to illustrate how $A_{i,p}$ 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 4×4 blocks of $C_{i,j}$ are each brought in from main memory.

The multiplication of each row panel of $A_{i,p}$ with $B_{p,j}$ to update the corresponding 4×4 block of $C_{i,j}$ (see Figure 2-right) is accomplished by our core via so-called rank-1 updates as follows. Let $\widehat{C}$, $\widehat{A}$, and $\widehat{B}$ be 4×4, $4 \times k_c$, and $k_c \times 4$ matrices, respectively. Then $\widehat{C} \mathrel{+}= \widehat{A}\widehat{B}$ can be computed as below:

$$\begin{pmatrix} \widehat{\gamma}_{(0,0)} & \cdots & \widehat{\gamma}_{(0,3)} \\ \vdots & \ddots & \vdots \\ \widehat{\gamma}_{(3,0)} & \cdots & \widehat{\gamma}_{(3,3)} \end{pmatrix} \mathrel{+}= \sum_{q=0}^{k_c-1} \begin{pmatrix} \widehat{\alpha}_{(0,q)} \\ \vdots \\ \widehat{\alpha}_{(3,q)} \end{pmatrix} \left(\widehat{\beta}_{(q,0)} \cdots \widehat{\beta}_{(q,3)}\right)$$

Each such update is known as a rank-1 update. Our core micro-architecture introduced previously is designed to natively implement these rank-1 updates. In our design, we distribute the matrices $A_{i,p}$ and $B_{p,j}$ on the array of PEs in Figure 1 in a 2D cyclic round-robin fashion, much as one distributes matrices on distributed memory architectures [41], [42]. Element $\widehat{\gamma}_{(\widehat{x},\widehat{y})}$ resides in an accumulator of PE$(\widehat{x},\widehat{y})$ during the computation. Now, a simple algorithm for computing this special case of GEMM among the PEs is, for $q = 0, \ldots, k_c - 1$, to broadcast the qth column of $\widehat{A}$ within PE rows, the qth row of $\widehat{B}$ within PE columns, after which a local MAC operation on each PE updates the local element of $\widehat{C}$.

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 of $C_{i,j}$ 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 [6]. 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).

3.2 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 $B_{p,j}$.

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 $B_{p,j}$. 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. This partitioning needs only the second SRAM, which is much

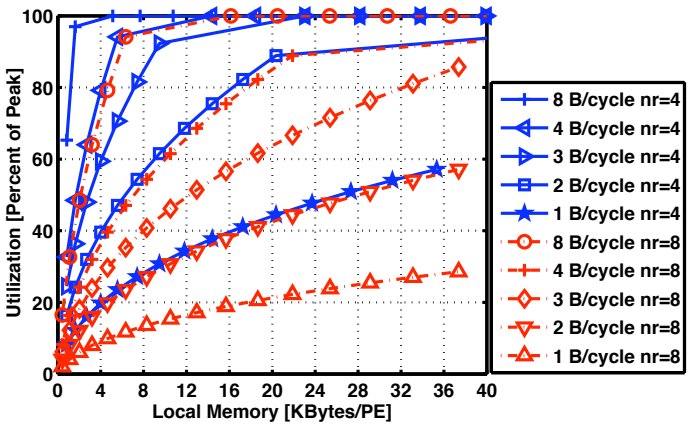

Fig. 3. Estimated core performance as a function of the bandwidth between LAC and on-chip memory, and the size of local memory with $n_r = 4$ and $n_r = 8$, $m_c = k_c$, and $n = 512$.

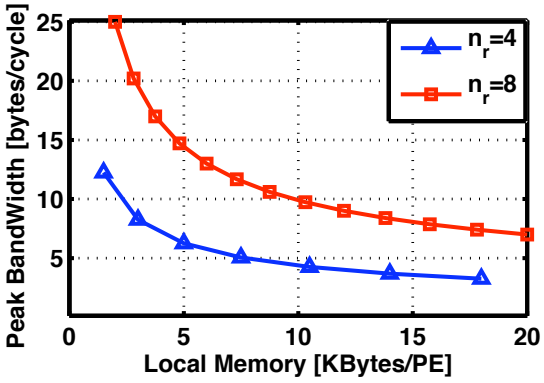

Fig. 4. Core Performance vs. bandwidth between LAC and on-chip memory for peak performance with $n_r = 4$ and $n_r = 8$, $m_c = k_c$, and $n = 512$.

smaller than the first one, to be dual-ported. In each cycle, an element of B is read from this SRAM to feed the local MAC unit in each PE. This strategy increases the aggregate local store size by a negligible amount, but reduces the power consumption significantly.

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. The focus is thereby on one computation $C_i \mathrel{+}= A_{i,p}B_p$. Since the complete computation $C \mathrel{+}= AB$ requires loops around this "inner kernel" for one C_i, 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. This extra memory is effective if there is enough bandwidth to bring data to the cores.

We have developed detailed analytical performance models in relation to storage and bandwidth requirements. Derivation of these models can be found in the Appendix.

3.3 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 tradeoff 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 the conditions under which we can achieve 100% utilization.

The tradeoff between bandwidth per core and local store per PE is shown in Figure 4. The curve illustrates the required bandwidth to maintain peak performance as a function of local store size. 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 requires repeaters, which adds overhead; (2) exploiting task-level parallelism and achieving high utilization is easier with a larger number of smaller cores; and (3) with our choice of $n_r = 4$, the number of MAC units in each core is comparable to modern GPUs, allowing us to more easily provide a fair comparison.

4 Linear Algebra Processor (LAP)

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. For simplicity and clarity of the following explanations, we assume that all original A, B, and C are square $n \times n$ matrices.

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 multiprocessor 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 analytical models (see Appendix) for the size of the shared on-chip memory and the required bandwidth between the LAP and external memory, all

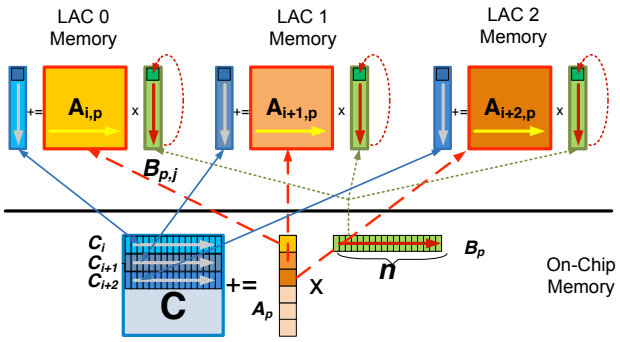

Fig. 5. Memory hierarchy with multiple cores in a LAP system.

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

Fig. 6. Bandwidth and memory requirements of different layers of memory hierarchy.

in relation to the number and size of the LAP cores themselves (see Section 3.2).

Figure 5 shows the use of the memory hierarchy for a larger matrix multiplication distributed across multiple ($S = 3$) cores. Different row blocks and panels of A and C are assigned to different cores. Bigger panels and blocks of 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.

Our analysis for the case of partial overlap shows that when computation time dominates the communication time, the peak performance is independent of the granularity at which C and the A panels are split into row chunks. Thus, this size m_c can be chosen to optimize bandwidth requirements and the local store sizes of the cores.

4.2 Chip-Level Exploration

Designing a complete system 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

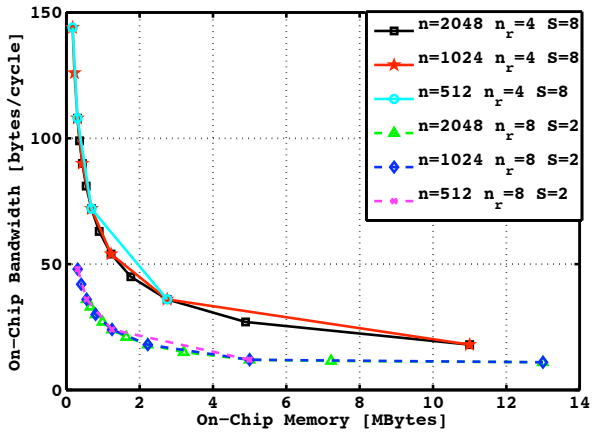

Fig. 7. On-chip bandwidth vs. memory size for different core organizations, and problem sizes for fixed number of total PEs, and $m_c = k_c$. Utilization in all cases is 93%+.

the sizes and organization of the different levels of the memory hierarchy.

Figure 6 summarizes the bandwidth requirements and size 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 sizes 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. In Section 5, we will merge the knowledge gained from these studies with power and area models to explore the design space from a practical perspective.

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 7. 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 larger on-chip memory, leading to a trade-off 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 LAP size vs. on-chip bandwidth and memory

In the following, we analyze the overall performance of the design for different on-chip memory sizes and on-chip memory bandwidths when the number of cores is increased. The curves in Figure 8 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 of 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 super-linearly. However, as the size of memory increases, there are performance gains when using more cores 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 that achieves peak performance is the optimal design point. Note that such a point is on the optimal design curve in Figure 7, too. For example, for the case of $S = 8$ cores, a bandwidth of 4 bytes (or 0.5 words) per cycle with an on-chip memory size of 13 MBytes, or a bandwidth of 8 bytes per cycle with an on-chip memory size of 2.5 MBytes are both optimal design points.

As mentioned above, we observe that an exponential increase in bandwidth is needed to maintain optimal performance with a linear increase in the number of cores. This can be further studied by finding the optimal points that have equal on-chip memory sizes, but a different number of cores. For example, to achieve peak performance with different number of cores $S = 4, 8, 16$ and 2.5 MBytes of on-chip memory, the required bandwidth is 2, 8, 32, bytes per cycle respectively. 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. Next, we shrink the available on-chip memory and compute the external bandwidth demands necessary to keep the performance over 90%. The algorithmic solution to this problem is adding another layer of blocking as shown in Figure 9. The matrix dimension of the original problem size is n and the new block size is n_s. We call this ratio $d = n/n_s$. After shrinking the available on-chip memory,

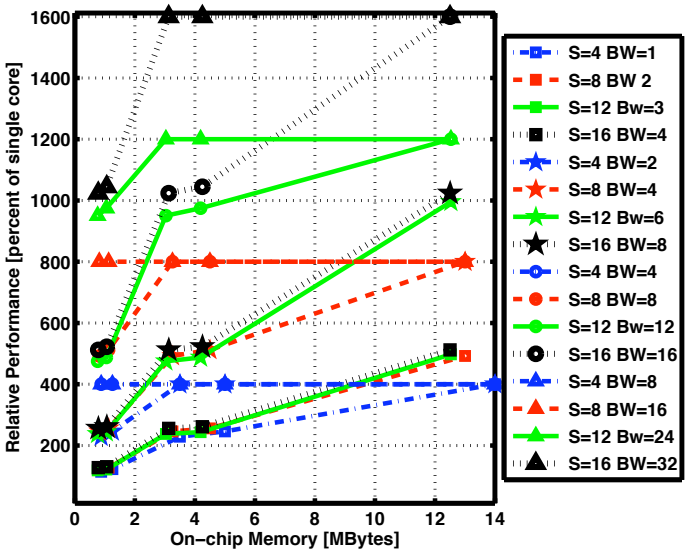

Fig. 8. LAP performance for different on-chip memory sizes, different number of cores, and different total on-chip bandwidths with $n_r = 4$ and $S = 4, 8, 12, 16$.

the solution assumes that a single (Figure 9-(a)) or $\widetilde{k} \leq d$ (Figure 9-(b,c) , with $\widetilde{k} = d$) sub-block of the original matrix C can fit in the new on-chip memory. Then, the algorithm performs all operations and data movements necessary to compute these $\widetilde{k}$ sub-blocks of C. The new off-chip bandwidth for the smaller on-chip memory and a sub-problem size $\widetilde{k} \times (n_s \times n_s)$ can be computed as

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

Figure 10 shows the external bandwidth demands for three different problem sizes and how they increase as the size of on-chip memory decreases. We observe that as the original problem size $n \times n$ increases, the external off-chip bandwidth requirement for the same system configuration decreases slightly. Still, to maintain high system utilization, a similar bandwidth vs. on-chip memory size trade-off exists.

Figure 11 summarizes overall performance of a 1.4 GHz 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 $n_s = 256, 512, 768$ or 1024 as the dimension of matrix C (kernel size, which translates into a corresponding on-chip memory size). As we increase the available core parallelism, the needed off-chip bandwidth increases for the same problem size[1]. Also, with the same off-chip bandwidth, we observe better performance, when the problem size grows. This graph shows that a small L2 memory size (as is the case in GPUs), which determines the possible on-chip problem size, limits the achievable peak utilization ("ex-

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

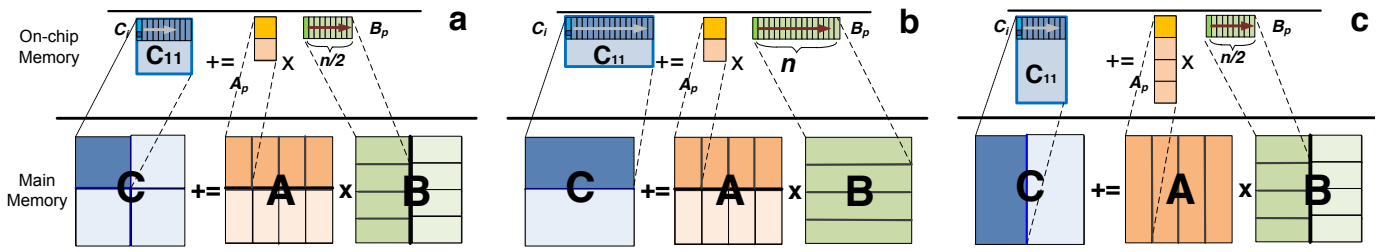

Fig. 9. Blocking algorithm to map a big problem on a small on-chip memory. a) blocking for quarter size b,c)blocking for half size.

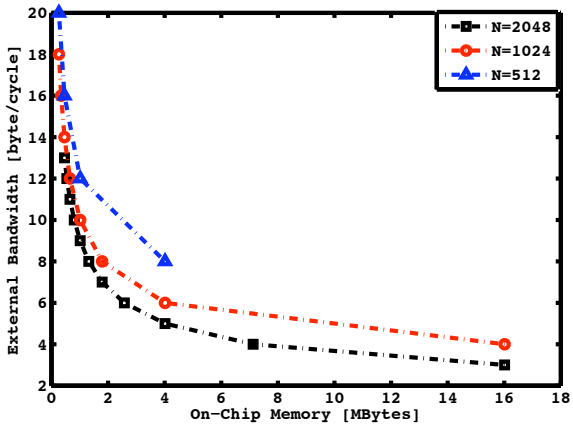

Fig. 10. External Bandwidth vs. Size of on-chip memory tradeoff for different original problem sizes. All utilization numbers are over 92%.

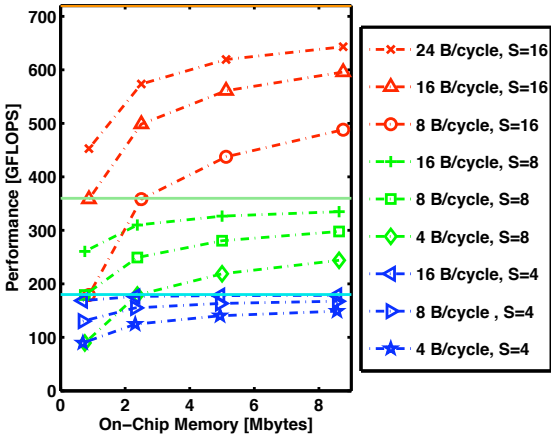

Fig. 11. LAP performance as a function of external off-chip bandwidth and the size of on-chip memory with $n_r = 4$, $m_c = k_c$. Horizontal lines denote peak performance for $S = 4, 8, 16$ respectively.

ploitable 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 Performance Prediction

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

There are two common limitations in parallel architectures that restrict their performance and efficiency. First, the core architectural and micro-architectural features can limit the accessibility of local register files and the number of instructions executed in each cycle. Second, the memory hierarchy organization that includes sizes of layers and bandwidths between them might not be able to sustain data movement from/to the computation cores. In the following, we assume that the cores are perfectly designed. The main metric affected by core-level design issues is the achievable peak efficiency in terms of both energy spent per operation (GFLOPS/W) and achievable utilization. We have shown how to design such an ideal core in 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 on-chip cache is 768 KBytes. The clock frequency is 1.15 GHz. Let us assume that cores are designed to achieve up to peak performance. With 768 KBytes and $S = 14$ cores, the dimension of the largest block of matrix C that is evenly divisible by S and $n_r = 4$ while fitting in the on-chip memory is $n_s = 280$. Including the corresponding panels of A and B, this setup fills 700 KBytes of on-chip L2 cache. Dividing the block C into row panels among the 14 cores results in $m_c = n_s/S = 280/14 = 20$. Hence, the size of each row panel of C is $m_c \times n_s = 20 \times 280$. Thus, the parameters of the design are as follows: $m_c = k_c = 20, S = 14, n_s = 280$. Assuming full overlapping, the maximum required off-chip bandwidth according to Figure 6 is $(\frac{4\times14\times4^2}{280}) \times$ 1.15GHz$\times$8Bytes$=$ 30GBytes/second, which is within the 144 GBytes/second that Fermi offers. The required on-chip bandwidth is $(\frac{2S}{k_c} + \frac{S}{m_c})n_r^2 = (\frac{2\times14}{20} + \frac{14}{20})4^2 \times$ 1.15 GHz $\times$8 Bytes$=$ 310 GBytes/second, which is much more than the 230 GBytes/second that Fermi offers. To calculate theoretically achievable utilization using such

a configuration, we divide the available bandwidth by the demanded bandwidth: $230/310 = 74\%$. In reality, implementations of GEMM on C2050 achieve 70% of peak performance [19]. Hence, our model accurately predicts that the on-chip bandwidth of Fermi does not meet the needs of matrix multiplication. One can overcome this under-utilization by increasing the on-chip bandwidth (see above), or by increasing the on-chip memory size. If the size of on-chip memory is doubled in the previous case, the required on-chip bandwidth can drop to half, or 160 GBytes/second, using the solution in Figure 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 [28]. The CSX architecture has 128 KBytes of on-chip memory. The block of C that fits on this memory is 64×128. Again, we assume that this architecture has six optimal 4×4 cores. Using the algorithm described in Figure 9, with $d = 16, \widetilde{k} = 2$, the minimum off-chip bandwidth demand is 4.7 GBytes/second. With an actual 4 GBytes/second off-chip bandwidth, our predicted upper limit for achievable utilization for this architecture is 83%. We can increase the utilization by increasing the size of on-chip memory. If one doubles the size of memory it can fit 128×128 blocks of C. Using the same algorithm with $d = 8, \widetilde{k} = 1$, the minimum off-chip bandwidth drops to 3.375 GBytes/second, which is less than off-chip bandwidth provided by the CSX architecture.

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.

We validated the performance model and LAP operation in general by developing a cycle-accurate micro-architectural LAP simulator. The simulator is written in C++, and can be plugged into a typical linear algebra implementation. The simulator is configurable in terms of PE pipeline stages, bus latencies, and memory and register file sizes. Furthermore, using power consumption numbers for the components, 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. We were able to successfully realize Cholesky factorization with minimal changes to the LAP control and data paths.

We have studied the feasibility of a LAP implementation in current 45nm bulk CMOS technology using publicly available components and their characteristics as published in literature. Details of the basic PE and core-level implementation are reported in [9], [43]. We used data from [10] for MAC units. For memories, register files and buses we used CACTI [11]. Running at a clock frequency of 1 GHz, which provides the best tradeoff between energy-delay and raw area and power efficiency, a 4×4 LAC is estimated to achieve an efficiency of 110 single-precision or 45 double-precision GFLOPS/W. The corresponding area efficiency and energy-delay are 0.1 mm^2/GFLOPS and 10 $\mathrm{mW/GFLOPS}^2$, respectively.

5.1 Power Modeling of Architectures

We have developed a general analytical power model that builds on existing component models (e.g. for Floating-Point Units (FPUs) and memories) described in the previous section. The model is derived from methods described in [44], [45], 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 GPGPU running matrix multiplication [46], [47]. We used the sizes of different GPU memory levels reported in [47] together with numbers from [46] and [48] 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 GPGPU [49], [50] or the Intel Penryn [51] 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.

5.2 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

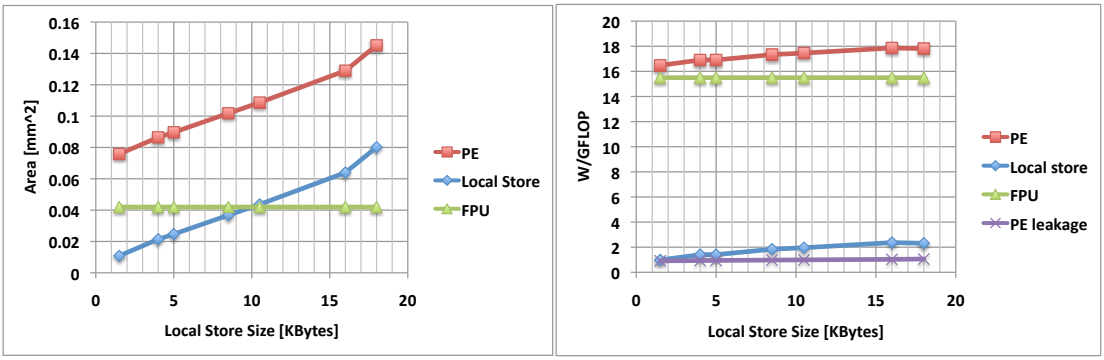

Fig. 12. Area of a single PE in a 4x4 core for different local store sizes (left), and leakage, local store, and total power efficiency of a PE in a 4x4 core at 45nm (right).

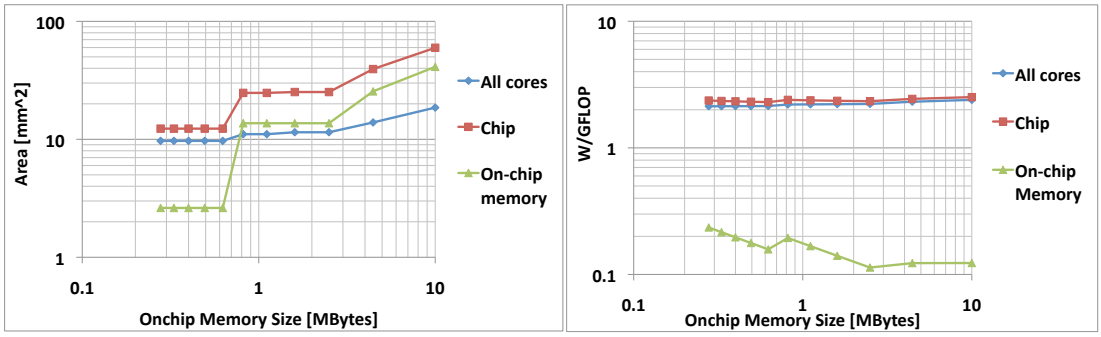

Fig. 13. Area (left) and power efficiency (right) of cores, on-chip memory and a total 128 MAC unit system with S=8 4x4 cores, different on-chip SRAM memory sizes, and n=2048.

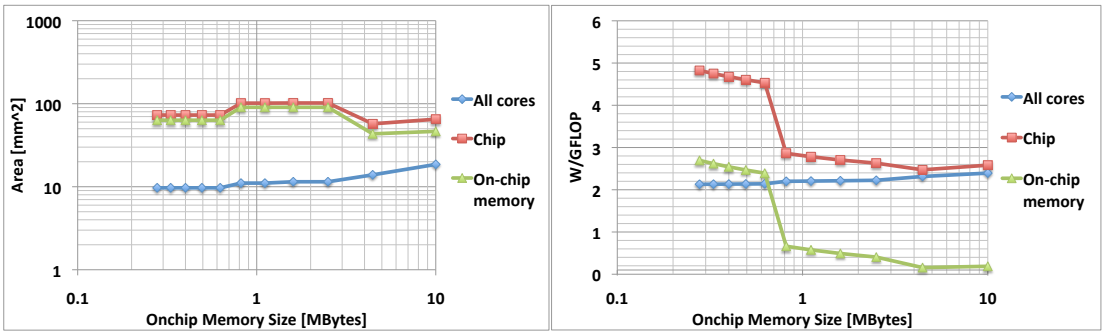

Fig. 14. Area (left) and power efficiency (right) of cores, on-chip memory and a total 128 MAC unit system with S=8 4x4 cores, different on-chip NUCA memory sizes, and n=2048.

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 the PE. With a local store size of 18 KByte, the local store occupies at most 2/3 of the PE. Overall PE area exhibits a linear relation to the local store capacity. 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, 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 requirement will increase exponentially, which decreases the utilization and also results in a significant increase in total power consumption. Finally, for algorithms like Cholesky factorization in which all the data is in-core, a bigger local

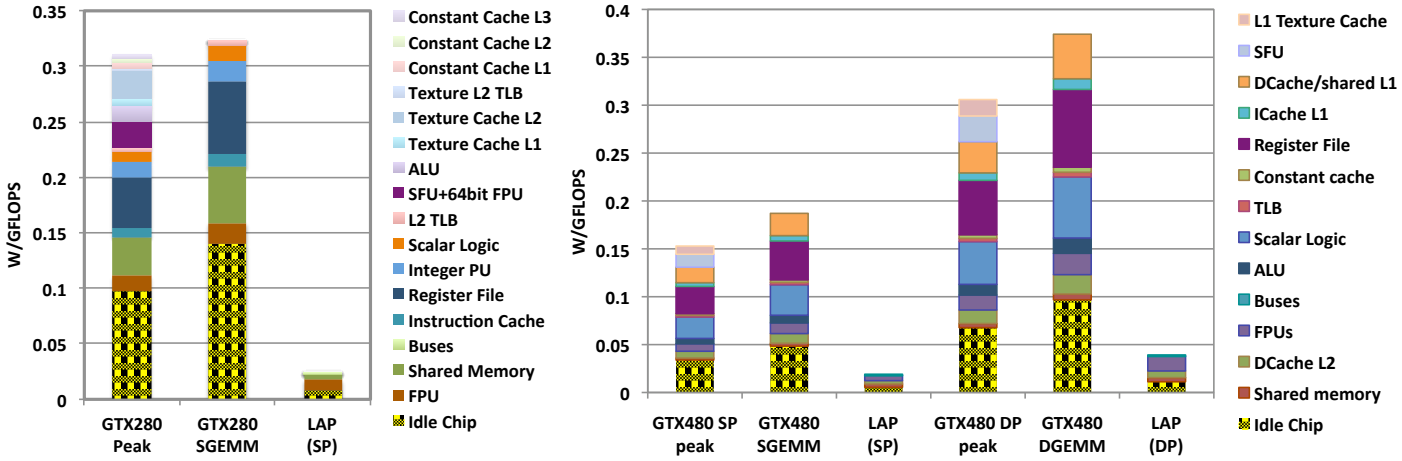

Fig. 15. Normalized power breakdown of Nvidia GPUs: Tesla GTX280 versus LAP at 65nm (left) and Fermi GTX480 versus LAP at 45nm.

store per PE yields the ability to handle bigger kernels and amortize 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 10). 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 [52] memory simulator of CACTI and replacing the SRAM design with 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 but simpler on-chip NUCA cache size is both more power and more area efficient.

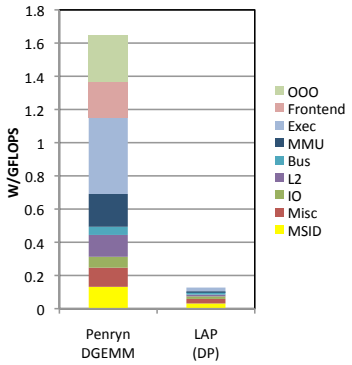

Fig. 16. Normalized power breakdown of Intel dual-core Penryn versus LAP at 45nm.

5.3 Comparative Power and Performance Analysis

Figure 15 and Figure 16 show a breakdown of performance-normalized power consumption for current high-performance GPGPU and multi-core architectures as compared to single- or double-precision versions of a prototypical LAP with an equivalent number of cores (i.e. Shared Multiprocessors, SMs, in GPUs[2]) running at equivalent raw Floating-point MAC (FMAC) performance (1.3GHz or 1.4GHz). In the case of GPUs (Figure 15), we show efficiencies for both peak operation and when running GEMM. Current GPUs run single- or double- precision GEMM (SGEMM or DGEMM) at around 60-70% of their theoretical peak performance [17], [18], [19]. 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 performance of 20 and 90 double-precision GFLOPS, respectively (Figure 16).

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

Breakdowns show that traditional architectures include significant overhead. The only units that are actually 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, and 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 SRAMs with a maximum of two read/write ports. For comparison with the Penryn, we mainly relied on the power breakdown presented in [51], where we assumed that GEMM utilizes the core with peak performance. 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 (OOO) and Frontend units that do not exist in the LAP architecture. Furthermore, with around 5 W, the execution unit consumes one third of the core power, which may be attributed to register file overheads, support for exception handling, and full IEEE-754 compatibility.

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 larger than in GPUs. This keeps bandwidth between memory layers low. All PE- and LAP-internal memories are pure, banked SRAMs with no tagging or internal cache consistency overhead. Consequently, despite being larger, memories are more power efficient and smaller than in other architectures. 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 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 in GPGPU architectures, which indicate that despite existing architectural features, idiosyncrasies of such architectures make it difficult to keep a large number of FPUs busy. Overall, the 2D PE arrangement with local, partitioned memory is scalable with an exponential growth in computation power for a linear growth in interconnect and bus lengths. With relatively low overhead for specialized MAC units and broadcast buses, we can envision such specialized data paths to be integrated into standard processor pipelines for orders of magnitude improved efficiency in a linear algebra computation mode.

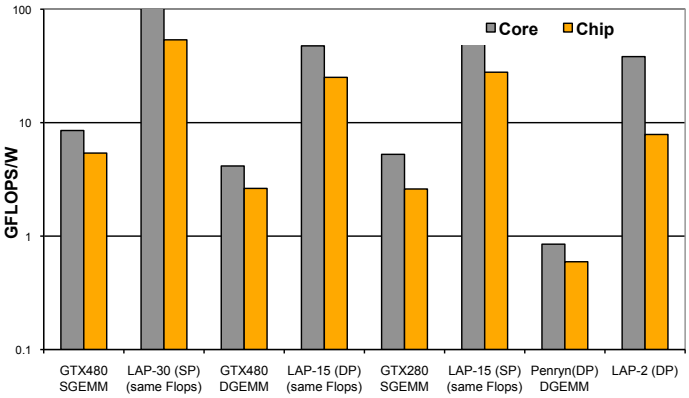

Fig. 17. Comparison of efficiencies for single- and double-precision GEMM between NVidia Tesla GTX280, NVidia Fermi GTX480, Intel Penry and a LAP of equivalent throughput.

5.4 Summary Comparison

We compare overall efficiency and inverse energy-delay [53] of single- and double-precision realizations of our design against other systems. Figure 17 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 5 Mbytes 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 940/470 GFLOPS and 70% utilization using 15 SMs at 1.4 GHz in total 500mm^2 chip area.

Finally, Figure 18 summarizes key metrics for various systems running GEMM as a representative matrix computation. For this table, we extended the analysis presented in [26] by including estimates for our LAP design, the 80-tile network-on-chip architecture from [27], the Power7 processor [54], the Cell processor [55], Intel Penryn [51], Intel Core i7-960 [56], CSX700 [28], Altera Stratix IV [31], and the NVidia Fermi GPU (GTX480) [50] all scaled to 45nm technology and to GEMM utilizations.

We note that for a single-precision LAP at around 1.4 GHz clock frequency, the estimated performance/power ratio is an order of magnitude better than in GPUs. The double-precision LAP design yields around 30 times higher 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 a LAP

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

Fig. 18. 45nm scaled performance and area of various systems running GEMM.

is at least an order of magnitude better than all other designs. All in all, with a double-precision LAP we can achieve up to 32 times higher performance in the same area as a complex conventional core 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 [2], one to two orders of magnitude improvement in power efficiency can be achieved. We now discuss possible extensions.

Figures 3 and 11 clearly show the trade-off between the sizes of the local and on-chip memories, and their corresponding bandwidth to on-chip and off-chip memory. One question that remains is the careful optimization of this trade-off across the multi-dimensional 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 buses that connect the rows and columns of PEs will likely become a limiting factor. This could be overcome by pipelining the communication between PEs or by further extending the 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 into area and complexity breakdowns for these architectures. This will help designers take into account both area and power budgets, with one of the main concerns in the future being the power density. We also plan to extend our cycle-accurate simulator into a full LAP system simulator and, if possible, integrate it with other multi-core simulators to study detailed design tradeoffs both at the core and chip levels. This integration will allow cycle-accurate modeling of dynamic power consumption of different design choices.

The GEMM operation is in and of itself important. It indirectly enables high performance for the level-3 BLAS [5], [7] as well as most important operations in packages like LAPACK [57]. We started our 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-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 [8], 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 entire class of operations.

References

[1] H. Esmaeilzadeh *et al.*, "Dark silicon and the end of multicore scaling," *ISCA '11*, pp. 365–376, 2011.

[2] R. Hameed *et al.*, "Understanding sources of inefficiency in general-purpose chips," *ISCA '10*, Jun 2010.

[3] N. Zhang *et al.*, "The cost of flexibility in systems on a chip design for signal processing applications," University of California, Berkeley, Tech. Rep., 2002.

[4] J. J. Dongarra *et al.*, "An extended set of FORTRAN basic linear algebra subprograms," *ACM Trans. Math. Soft.*, vol. 14, no. 1, pp. 1–17, March 1988.

[5] ——, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Soft.*, vol. 16, no. 1, pp. 1–17, March 1990.

[6] K. Goto *et al.*, "Anatomy of a high-performance matrix multiplication," *ACM Trans. Math. Soft.*, vol. 34, no. 3, May 2008.

[7] B. Kågström *et al.*, "GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark," *ACM Trans. Math. Soft.*, vol. 24, no. 3, pp. 268–302, 1998.

[8] K. Goto *et al.*, "High-performance implementation of the level-3 BLAS," *ACM Trans. Math. Softw.*, vol. 35, no. 1, pp. 1–14, 2008.

[9] A. Pedram *et al.*, "A high-performance, low-power linear algebra core," *ASAP '11*, pp. 35–41, 2011.

[10] S. Galal *et al.*, "Energy-efficient floating point unit design," *IEEE Transactions on Computers*, vol. PP, no. 99, pp. 1 – 1, 2010.

[11] N. Muralimanohar *et al.*, "CACTI 6.0: a tool to model large caches," HP Laboratories Palo Alto, Technical Report HPL-2009-85, 2009.

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

[13] R. C. Whaley *et al.*, "Automatically tuned linear algebra software," in *Proceedings of SC'98*, 1998.

[14] S. Rixner *et al.*, "Register organization for media processing," *HPCA-6*, pp. 375 – 386, 2000.

[15] C. Kozyrakis *et al.*, "Overcoming the limitations of conventional vector processors," *ISCA'03*, pp. 399 – 409, 2003.

[16] K. Fatahalian *et al.*, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," *HWWS '04:ACM SIGGRAPH/EUROGRAPHICS*, Aug 2004.

[17] V. Allada *et al.*, "Performance analysis of memory transfers and GEMM subroutines on NVIDIA Tesla GPU cluster," *CLUSTER '09*, pp. 1 – 9, 2009.

[18] V. Volkov *et al.*, "Benchmarking GPUs to tune dense linear algebra," *SC 2008*, pp. 1 – 11, 2008.

[19] G. Tan *et al.*, "Fast implementation of DGEMM on fermi GPU," *SC '11*.
[20] R. Urquhart *et al.*, "Systolic matrix and vector multiplication methods for signal processing," *IEEE Communications, Radar and Signal Processing,*, vol. 131, no. 6, pp. 623 – 631, 1984.
[21] V. Kumar *et al.*, "Synthesizing optimal family of linear systolic arrays for matrix computations," *ICSA '88*, pp. 51 – 60, 1988.
[22] H. Jagadish *et al.*, "A family of new efficient arrays for matrix multiplication," *Computers, IEEE Transactions on*, vol. 38, no. 1, pp. 149 – 155, 1989.
[23] T. Lippert *et al.*, "Hyper-systolic matrix multiplication," *Parallel Computing*, Jan 2001.
[24] K. Johnson *et al.*, "General-purpose systolic arrays," *Computer*, vol. 26, no. 11, pp. 20 – 31, 1993.
[25] C. Takahashi *et al.*, "Design and power performance evaluation of on-chip memory processor with arithmetic accelerators," *IWIA2008*, pp. 51 – 57, 2008.
[26] J. Kelm *et al.*, "Rigel: an architecture and scalable programming interface for a 1000-core accelerator," *ISCA '09*, Jun 2009.
[27] S. Vangal *et al.*, "An 80-tile sub-100-w teraFLOPS processor in 65-nm cmos," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 29 – 41, 2008.
[28] "CSX700 Floating Point Processor," ClearSpeed Technology Ltd, Datasheet 06-PD-1425 Rev 1, 2011.
[29] M. Parker, "Achieving teraFLOPS performance with 28nm FPGAs," *EDA Tech Forum*, December 2010.
[30] O. Garreau *et al.*, "Scaling up to teraFLOPS performance with the Virtex-7 family and high-level synthesis," *Xilinx White Paper: Virtex-7 FPGA*, February 2011.
[31] M. Parker, "High-performance floating-point implementation using FPGAs," in *MILCOM*, 2009.
[32] P. Zicari et al., "A matrix product accelerator for field programmable systems on chip," *Microprocessors and Microsystems 32*, 2008.
[33] 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*, vol. 18, no. 4, pp. 433 – 448, 2007.
[34] G. Kuzmanov *et al.*, "Floating-point matrix multiplication in a polymorphic processor," *ICFPT 2007*, pp. 249 – 252, 2007.
[35] Y. Dou *et al.*, "64-bit floating-point FPGA matrix multiplication," *FPGA '05*.
[36] J.-W. Jang *et al.*, "Energy- and time-efficient matrix multiplication on FPGAs," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 13, no. 11, pp. 1305 – 1319, 2005.
[37] V. Kumar and Y. Tsai, "On synthesizing optimal family of linear systolic arrays for matrix multiplication," *Computers, IEEE Transactions on*, vol. 40, no. 6, pp. 770 – 774, 1991.
[38] V. Eijkhout, *Introduction to High Performance Scientific Computing*. `www.lulu.com`, 2011.
[39] J. Li, "A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies," *Citeseer*, Jan 1996.
[40] R. van de Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, April 1997.
[41] B. A. Hendrickson and D. E. Womble, "The Torus-Wrap mapping for dense matrix calculations on massively parallel computers," *SIAM J. Sci. Stat. Comput.*, vol. 15, no. 5, pp. 1201–1226, 1994.
[42] J. Choi *et al.*, "ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers," *FMPC '92*.
[43] A. Pedram *et al.*, "Co-design tradeoffs for high-performance, low-power linear algebra architectures," UT Austin, CERC, Tech. Rep. UT-CERC-12-02, 2011.
[44] S. Li *et al.*, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," *MICRO-42.*, 2009.
[45] D. Brooks et al., "Wattch: a framework for architectural-level power analysis and optimizations," *ISCA, 2000.*, pp. 83 – 94, 2000.
[46] S. Hong *et al.*, "An integrated GPU power and performance model," *ISCA '10*, Jun 2010.
[47] H. Wong *et al.*, "Demystifying gpu microarchitecture through microbenchmarking," *ISPASS'2010*, pp. 235 – 246, 2010.
[48] "Samsung DDR3 SDRAM:High-Performance, Energy-Efficient Memory for Today's Green Computing Platforms," SAMSUNG Green Memory, Tech. Rep., March 2009.
[49] "Fermi computer architecture white paper," NVIDIA, Technical Report, 2009.
[50] D. Kanter, "Inside Fermi: Nvidia's HPC push," Real World Technologies, Tech. Rep., September 2009.
[51] V. George *et al.*, "Penryn: 45-nm next generation intel® core™ 2 processor," *IEEE Asian Solid-State Circuits Conference*, Jan 2008.
[52] N. Muralimanohar *et al.*, "Architecting efficient interconnects for large caches with cacti 6.0," *IEEE Micro*, vol. 28, 2008.
[53] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 9, pp. 1277 –1284, sep 1996.
[54] M. Ware *et al.*, "Architecting for power management: The IBM® POWER7™ approach," *HPCA 2010*, pp. 1 – 11, 2010.
[55] F. Lauginiger et al., "Performance of a multicore matrix multiplication library," *STMCS 2007,*, Jan 2007.
[56] E. S. Chung *et al.*, "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?" *MICRO '43*, pp. 225–236, 2010.
[57] E. Anderson *et al.*, *LAPACK Users' guide (third ed.)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999.

Ardavan Pedram is a PhD candidate at the University of Texas at Austin. He received the masters degree in computer engineering from the University of Tehran in 2006. He enrolled in the doctoral program in computer engineering at the University of Texas at Austin in 2007. His research interests include high performance computing and computer architecture. He specifically works on hardware-software co-design (algorithm for architecture) of special purposed accelerators for high-performance energy-efficient linear algebra and signal processing applications.

Robert A. van de Geijn is a Professor of Computer Science and member of the Institute for Computational Engineering and Sciences at the University of Texas at Austin. He received his PhD in Applied Mathematics from the University of Maryland College Park, in 1987. His interests are in linear algebra, high-performance computing, parallel computing, and formal derivation of algorithms. He heads the FLAME project, a collaboration between UT-Austin, Universidad Jaume I (Spain), and RWTH Aachen University (Germany). This project pursues foundational research in the field of linear algebra libraries and has led to the development of the libflame library, a modern, high-performance dense linear algebra library that targets both sequential and parallel architectures. One of the benefits of this library lies with its impact on the teaching of numerical linear algebra, for which Prof. van de Geijn received the UT Presidents Associates Teaching Excellence Award. He has published several books and more than a hundred refereed publications.

Andreas Gerstlauer received the Dipl.-Ing. degree in electrical engineering from the University of Stuttgart, Germany, in 1997, and the M.S. and Ph.D. degrees in information and computer science from the University of California, Irvine (UCI), in 1998 and 2004, respectively.

Since 2008, he has been with the University of Texas at Austin, where he is currently an Assistant Professor in electrical and computer engineering. Prior to joining the University of Texas, he was an Assistant Researcher with the Center for Embedded Computer Systems, UCI, leading a research group to develop electronic system-level design tools.

Dr. Gerstlauer serves on the program committee of major conferences such as DAC, DATE and CODES+ISSS. His research interests include system-level design automation, system modeling, design languages and methodologies, and embedded hardware and software synthesis.

APPENDIX
ANALYTICAL FORMULAE

In this Appendix, we present derivations of our analytical formulae, which are summarized in Figure 6. We first derive details of the core layer and then details of the shared memory layer.

Core Architecture

The size of the local store, aggregated over all PEs, is given by $m_c \times k_c$ elements for $A_{i,p}$, plus $2 \times k_c \times n_r \times n_r$ elements for the current and next $B_{p,j}$ and $B_{p+1,j}$[1]. In total, the local memory must be able to hold $m_c k_c + 2k_c n_r^2 = (m_c + 2n_r^2)k_c$ single or double precision floating point numbers. 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 element of $B_{p,j}$, $\beta_{p,j}$, every PE requires a register file of size 4 (a size of 3, rounded up to the next power of two).

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

To achieve peak performance, the prefetching of the next block of A, $A_{i,p+1}$, is also 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.

LAP Architecture

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 $A_{i,p}$, a n_r^2 subblock of $C_{i,j}$ and a $k_c \times n_r$ panel $B_{p,j}$ (replicated across PEs). The shared on-chip memory stores a complete $n \times n$ block of matrix C and the current $k_c \times n$ row panel of B, which is shared 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 at the chip level.

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

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

Note that when computation dominates (the second term in the "max" dominates) the peak performance is independent of m_c.

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 $n^2 k_c/Sn_r^2$ 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.

1. Note that the elements of $B_{p,j}$ and $B_{p+1,j}$ are replicated and the storage is n_r times more.