High Performance Dense Linear Algebra on a Spatially Distributed Processor

Jeff Diamond* Behnam Robatmili*
Stephen W. Keckler Robert van de Geijn Kazushige Goto Doug Burger

Department of Computer Science
Texas Advanced Computing Center
The University of Texas at Austin
{jdiamond,beroy,skeckler,rvdg}@cs.utexas.edu
kgoto@tacc.utexas.edu, dburger@cs.utexas.edu

Abstract

As technology trends have limited the performance scaling of conventional processors, industry and academic research has turned to parallel architectures on a single chip, including distributed uniprocessors and multicore chips. This paper examines how to extend the archtypical operation of dense linear algebra, matrix multiply, to an emerging class of uniprocessor architectures characterized by a large number of independent functional units, register banks, and cache banks connected by a 2-D on-chip network. We extend the well known algorithm for matrix multiplication by Goto to this spatially distributed class of uniprocessor and describe the optimizations of the innermost kernel, a systolic-like algorithm running on a general purpose uniprocessor. The resulting implementation yields the first demonstration of high-performance in an application executing on the TRIPS processor hardware, a next-generation distributed processor core. We show that such processors are indeed capable of substantial improvements in single threaded performance provided their spatial topography is taken into account.

Categories and Subject Descriptors C.1.1 [*Computer Systems Organization*]: Single Data Stream Architectures; C.4 [*Computer Systems Organization*]: Performance of Systems; C.1.3 [*Computer Systems Organization*]: Other Architecture Styles; C.5.3 [*Computer Systems Organization*]: Microcomputers; D.1.3 [*Concurrent Programming*]: Parallel Programming

General Terms Algorithms, Design, Performance

Keywords Instruction Level Parallelism, dense linear algebra, matrix multiply, hybrid dataflow, on-chip networks, tile based architecture, grid processors, GotoBLAS

1. Introduction

Previous techniques of increasing single thread performance through ever deepening pipelines have finally run up against power limits and diminishing returns. This has led to changes in computer architecture, including emerging massively multicore processors [8], so dramatic that entire programming paradigms must shift to take advantage of them. Existing software will no longer see performance increases, so many applications will need to be rewritten. That conventional processors will no longer be able to accelerate legacy code ironically frees architects to pursue novel architectures. Processors like Raw [5, 10], WaveScalar [4], and TRIPS [7] tend to depart from the classic scalar/Von Neumann architecture of program execution and begin to resemble two-dimensional distributed programming models. In our experience, the characteristic of spatially distributed processors most relevant to the design of optimized code is the communication substrate, especially when the topology is exposed to the programmer.

To determine how conventional algorithms might need to be modified and what performance gains might be possible for these new architectures, we chose to evaluate the implementation of matrix multiply on such a system. A heavily optimized matrix-matrix multiply is at the core of high performance dense linear algebra libraries like the BLAS [17, 18], while at the same time representing a simple, regular operation that is easy to reason about. In this paper, we demonstrate that for our target application, obtaining extremely high performance on such architectures is possible. We further show that this performance can be achieved by applying well known existing parallel algorithms in novel contexts, but that careful attention to detail is required to prevent the interconnecting network from becoming a bottleneck. The result is a systolic-like algorthm running on a general purpose processor.

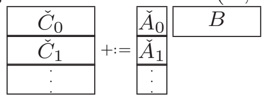
The part of our algorithm most important to achieving high performance is the design of our innermost kernel. We focus on the optimizations that are broadly applicable to spatially distributed processors, omitting many details specific only to our chosen implementation platform.

Section 2 discusses the basic mathematical underpinnings of general high performance matrix multiply, reviews the basic approach of [1, 3], and presents an overview of distributed architectures such as the TRIPS processor, which we use for our evaluation. Section 3 discusses extensions to the conventional algorithm for spatially distributed processors and how to apply these principles to the TRIPS prototype hardware. Section 4 presents an empirical study of matrix multiply on the TRIPS processor. Section 5 discusses optimization principles we discovered when mapping matrix multiply to TRIPS and Section 6 concludes the paper.

* First authors Robatmili and Diamond have made equal contributions to this work and are listed alphabetically.

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.
Copyright © 2008 ACM 978-1-59593-960-9/08/0002...$5.00

```
Algorithm: $C := \text{DGEPP}(A, B, C)$
```



```
Pack $B$ into $\tilde{B}$
for $i = 0, \dots, M-1$
    Pack $\check{A}_i$ into $\tilde{A}$
    $\check{C}_i := \tilde{A}\tilde{B} + \check{C}_i$
endfor
```

Figure 1. Outline of optimized implementation of DGEPP.

```
Algorithm: $C := \text{DGEBP}(\tilde{A}, \tilde{B}, C)$

for $j = 0, n-1$
    for $i = 0, m_b - 1$                          ⎫
        for $p = 0, k_b - 1$                      ⎪
            $\gamma_{ij} += \alpha_{ip}\beta_{pj}$      ⎬ $c_j := \tilde{A}\tilde{b}_j + c_j$
        endfor                                    ⎪
    endfor                                        ⎭
endfor
```

Figure 2. The DGEBP algorithm for conventional processors

2. Background

This section summarizes a state-of-the-art algorithm for high performance matrix multiply on conventional uniprocessors. It then highlights some emerging trends in uniprocessor design which have led to modern spatially distributed architectures.

2.1 Conventional DGEMM Algorithm

Goto's streaming matrix multiply algorithms are commonly found at the core of state of the art linear algebra libraries in conventional processors [1]. There have been other high profile approaches to matrix multiply algorithms in the past, e.g. Atlas [19, 20], but Goto's algorithms have demonstrated the highest performance [1, 13]. Consider the computation $C := AB + C$, where C, A, and B are $m \times n$, $m \times k$, and $k \times n$ matrices, respectively. Assume for simplicity that $m = b_m M$, $n = b_n N$, and $k = b_k K$, where M, N, K, b_m, b_n, and b_k are all integers. Partition original matrices as follows:

$$A \rightarrow \left(\begin{array}{c|c|c|c} A_0 & A_1 & \cdots & A_{K-1} \end{array} \right) \text{ and } B \rightarrow \left(\begin{array}{c} \check{B}_0 \\ \hline \check{B}_1 \\ \hline \vdots \\ \hline \check{B}_{K-1} \end{array} \right),$$

where A_p and $\check{B}_p$ contain b_k columns and rows, respectively and the symbol $\check{}$ is used to indicate a partitioning by rows. Then

$$C := A_0\check{B}_0 + A_1\check{B}_1 + \cdots + A_{K-1}\check{B}_{K-1} + C.$$

A typical high-performance implementation of DGEMM (Double precision General Matrix Multiply) will focus on making each update $C := A_p\check{B}_p + C$ (a *panel-panel multiplication*, DGEPP) as fast as possible. The overall performance of DGEMM is driven by that of each individual DGEPP (Double precision General Panel Panel multiply) with panel width equal to an optimal size b_k.

Figure 1 gives a high-performance algorithm for the DGEPP operation, $C := AB + C$, where the "k" dimension is b_k. The algorithm requires three highly optimized components:

- **Pack B:** A routine for packing B into a contiguous buffer. On some architectures this routine may also reorganize the data for specialized instructions used by the DGEBP (Double General Block Panel multiply) kernel routine described below.
- **Pack and transpose $\check{A}_i$:** A routine for packing $\check{A}_i$ into a contiguous buffer ($\tilde{A}$). Often this routine also transposes the matrix to improve the order in which it is accessed by the DGEBP kernel routine. In general, the purpose of packing the $\check{A}_i$ and B panels, where the symbol $\sim$ indicates a packed block, is to rearrange the data in a way that the lower level kernel can access it more coherently.
- DGEBP **kernel routine:** This routine computes $\check{C}_i := \tilde{A}\tilde{B} + \check{C}_i$ using the packed buffers.

On current conventional architectures, the size of $\check{A}_i$ is chosen to fill about half of the L2 cache (or the memory addressable by the TLB), as explained in [1]. Considerable effort is required to tune each of these components, especially the DGEBP kernel routine.

We now give a high level description of how DGEBP itself is (naively) implemented. Assume that $C = \tilde{A}\tilde{B} + C$ is to be computed, where $\tilde{A}$ is $m_b \times k_b$ and C and B have n columns. The key is to orchestrate the computation so that $\tilde{A}$ stays in the L2 cache and $\tilde{B}$ and C are streamed from memory. To accomplish this, the jth column of C, denoted by c_j, is computed as $\check{c}_j := \tilde{A}\tilde{b}_j + \check{c}_j$, where $\tilde{b}_j$ equals the jth column of $\tilde{B}$. Thus as a first approximation, the implementation of DGEBP boils down to the implementation of a matrix-vector multiplication, where the matrix is in the L2 cache and the vectors are assumed to be in memory.

A specific entry of C, γ_{ij}, can be computed as the inner product of the ith row of $\tilde{A}$ and $\tilde{b}_j$. Thus, we should orchestrate the computation so that $\tilde{b}_j$ remains in the L1 cache (since it is reused many times), and to load elements of $\check{c}_{ij}$ into registers from main memory before they are updated by the inner product, after which they are stored back to main memory. In practice, a number of elements of $\check{c}_{ij}$ at a time are kept in registers and updated, allowing the cost of loading an element of $\tilde{b}_j$ from the L1 cache to registers to be amortized over more computation. Furthermore, in order to amortize the cost of bringing elements of $\tilde{A}$ into registers, computation with several columns $\tilde{c}_{ij}$ and $\tilde{b}_j$ may be occurring. With careful prefetching, the cost of loading elements of $\tilde{A}$ from the L2 cache into registers can be hidden by the computation with a previous element of that matrix, allowing DGEBP to achieve close to peak performance. Figure 2 shows the resulting algorithm that implements this simple DGEBP.

2.2 Linear Algebra Algorithms for Distributed Memory and Multithreaded Architectures

Many studies have been focused on parallel dense linear algebra algorithms for distributed memory and SMP/multicore architectures. The most practical distributed memory matrix multiplication algorithms are summarized in [15] while a recent algorithm that targets SMP-like architectures is discussed in [14] . Further references can be found in those papers. On those architectures, the primary challenge lies with the partitioning of the matrices and course-grained computation among processors. Extending the conventional algorithm to Chip Multi Processors (CMPs) is largely an extension of this paradigm in which each core is in essence a uniprocessor, although the nature of the higher level algorithm must be altered to accommodate shared on chip caches and off chip bandwidth [16].

Future chips are clearly going to allow for more concurrency, but it is not yet clear what architectures will be the best. The core of a spatially partitioned uniprocessor represents the finest granularity of parallelism, and the purpose of this paper is to examine how current approaches for matrix multiply can be adapted in this context. Light weight CMPs such as Cyclops, Niagara 2, or modern GPUs represent a different point in the spectrum [22]. Our results and other published data [1] show that our algorithm is generally applicable to a wide range of architectures, and so we expect it would do well on a varety of CMPs. In particular, our observations about network contention and load balancing will likely play an important role across this spectrum, while specific optimizations of the innermost kernel will likely be different on each platform.

2.3 Characteristics of an Emerging Class of Uniprocessor Architectures

The low level kernel of a matrix multiply algorithm is highly dependent on the structure of the uniprocessor. Distributed uniprocessor architectures are now emerging to allow higher frequency operation at lower power, while exposing greater concurrency and data bandwidth. For example, the Raw processor [9, 10] integrates a low latency on-chip network into the pipelines of the processors in a single-chip multiprocessor and allows a programmer to program the network routers for static routing. This feature allows a programmer to treat the Raw tiles as elements of a distributed serial processor. A 2-D spatially distributed uniprocessor may superficially look like a CMP, but the key differences are in the granularity of computation, with only a few computations done and a few data values transferred per operation, and the granularity of synchronization, with the ability to efficiently synchronize at a fine grain such as a single datum. Often computation is triggered by the arrival of data at each element. Computational elements are often little more than a simple FPU with little local storage, and are typically referred to as Data Processing Units (DPUs).

The most important programming characteristic of a spatially distributed architecture is that the actual topology of instruction placement is exposed in the ISA (instruction set architecture), and performance is greatly dependent on the exact placement of instructions and the methods of routing data between them. For matrix multiply algorithms, 2-D spatial arrays are a natural match for 2-D matrix blocks.

Systolic arrays were specifically designed for regular, computationally intensive tasks like matrix multiply [11, 12]. A typical systolic array for matrix multiply has a 2-D array of DPUs, each with a certain number of local registers for accumulating C values. Values are passed among DPUs through nearest neighbor network connections.

In a systolic algorithm for matrix multiplication, a panel of A values is streamed through one side of the array, while a panel of B values is streamed through the orthogonal network. On each cycle of operation, each DPU adds the product of A and B to a local C register, then passes A and B on to their nearest neighbor.

While it would be possible to implement a systolic algorithm on TRIPS, it would not truly be "systolic", since TRIPS has no local storage at DPUs for accumulating values, and both A and B vectors would both need to be streamed from memory, removing the benefits of orthogonal datapaths.

Spatially distributed uniprocessors are designed for general purpose computation, not matrix multiply, and thus differ from a systolic array in ways that require some changes to the algorithm. TRIPS [7] is a prototypical example of a spatially distributed processor, having DPUs, register banks, and L1 cache banks connected via a tightly coupled on-chip network coordinated by the program. TRIPS uses a hybrid dataflow model that places small groups of instructions on the 2-D grid of DPUs and executes them in dataflow fashion using systolic-like communication between the DPUs. Unlike a pure dataflow design, each dataflow group reads and writes to register banks, allowing more conventional algorithms to be used. Wavescalar is another tiled architecture that also employs a dataflow execution model but incorporates greater degree of speculation [4]. Exploiting locality in Wavescalar is challenging as the lack of register storage prevents easy accumulation of C in matrix multiply.

2.4 Overview of the TRIPS Processor

Because TRIPS is a good example of a distributed uniprocessor design and has existing hardware that is readily accessible, we chose this as the basis for our matrix multiply implementation. Substantial detail on the TRIPS architecture and its silicon implementation can be found in [6, 7].

Figure 3 shows a diagram of a TRIPS processor core composed of a 2-D grid of 16 execution tiles (ETs), 4 distributed register file tiles (RTs), 4 distributed data tiles (DTs) as L1 cache, 5 instruction cache tiles, and a global control tile. Each ET has an integer unit, a floating-point unit, and reservation stations for instructions. Each RT includes 32 registers resulting in a total register file capacity of 128 registers. The TRIPS tiles communicate using a lightweight operand network which dynamically routes operands and load/store traffic through intermediate tiles in Y-X order as necessary.

A TRIPS program consists of a series of instruction blocks that are individually mapped onto the array of execution tiles and executed atomically. Instruction block inputs are read from registers and delivered into the ET array. Operands produced and consumed within the array are delivered directly to the target instruction with each instruction encoding its targets. Operand arrival triggers instruction execution, thus implementing a dataflow execution model. A TRIPS block may hold up to 128 computation instructions with up to 8 mapped to any given ET. Up to 8 blocks may execute simultaneously (1 non-speculative block and up to 7 speculative blocks) resulting in a maximum of a 1,024 instruction window. Because each of the tiles operates independently, the processor can execute up to 16 instructions and perform up to 4 L1 cache accesses per cycle.

3. A New Kernel Algorithm for Spatially Distributed Processors

Goto's algorithm has two primary components: a higher level concerned with data packing and large block movement, and an innermost kernel routine that performs the block panel matrix multiply [1]. Our contribution lies primarily with the innermost kernel. We describe a data placement motivated by spatial characteristics and derive a sub-blocking scheme to accommodate this placement. We also discuss the mapping of operations in the inner kernel to the 2-D topology of a partitioned uniprocessor and resulting alterations to the higher level algorithm.

3.1 Extending DGEBP Blocking Hierarchy for Spatial Processors

We extend the high level DGEBP algorithm with another level of sub-blocking to exploit a partitioned processor's large number of partitioned registers. We then show how the topology of a spatial uniprocessor motivates general block placement.

The two characteristics of spatially distributed processors that most affect data placement in the inner kernel are a large number of registers and a 2-D network connecting the DPUs, L1 cache banks and register banks. The traditional method of block panel multiply stores small blocks of C in registers and performs outer products on these register blocks. Distributed register banks allow such architectures to have a large number of registers, which in turn permits larger register blocks and leverages the property that the ratio of computation to bandwidth increases linearly with block size.

We can take full advantage of bandwidth in two dimensions and avoid loading both elements of A and B from L1 cache banks by

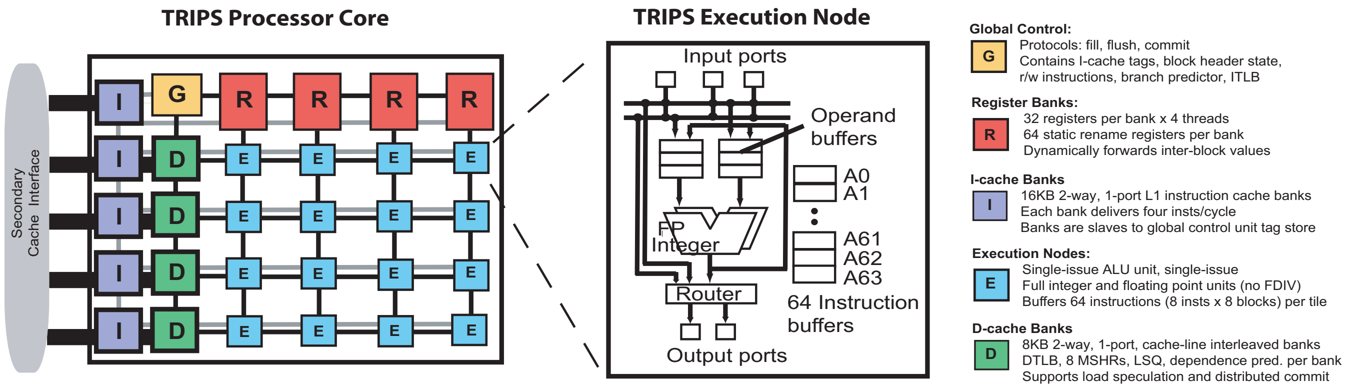

TRIPS Processor Core
Secondary Cache Interface
I
G
R
D
E
TRIPS Execution Node
Input ports
Operand buffers
A0
A1
A61
A62
A63
FP
Integer
Router
64 Instruction buffers
Output ports
Global Control:
Protocols: fill, flush, commit
Contains I-cache tags, block header state, r/w instructions, branch predictor, ITLB
Register Banks:
32 registers per bank x 4 threads
64 static rename registers per bank
Dynamically forwards inter-block values
I-cache Banks
16KB 2-way, 1-port L1 instruction cache banks
Each bank delivers four insts/cycle
Banks are slaves to global control unit tag store
Execution Nodes:
Single-issue ALU unit, single-issue
Full integer and floating point units (no FDIV)
Buffers 64 instructions (8 insts x 8 blocks) per tile
D-cache Banks
8KB 2-way, 1-port, cache-line interleaved banks
DTLB, 8 MSHRs, LSQ, dependence pred. per bank
Supports load speculation and distributed commit

Figure 3. Block Diagram of a TRIPS Processor Core.

additionally storing a block of A in registers. However, the register banks can only hold a finite amount of data which must be loaded from memory into registers before use and later returned. Therefore, we should maximize the reuse of elements of A as well as C in register blocks. Since C determines the amount of local computation, a register block of C must be as large as possible. To additionally reuse a smaller block of A in registers necessitates small block-panel multiplies between A and B. Use of panels implies rectangular shaped register blocks for B and C, which can potentially lower the ratio of computation to memory bandwidth. However, this advantage is offset by decreasing the memory bandwidth that would be required if both A and B blocks were loaded from cache. Streaming data from two locations also provides a more natural computational fit to the existing physical grid. To better leverage this topology, the innermost kernel recasts this small block panel multiply as a series of matrix-vector multiplies between register block of A and individual columns of B read from memory. By only updating a single column of C each multiply instead of doing a traditional outer product, we reduce the required register bandwidth to achievable levels.

Block Decomposition: We now formally extend the high level algorithm with this extra level of decomposition by further partitioning the C panel, $\tilde{A}$ blocks, and $\tilde{B}$ panel in the GEPB algorithm. We denote a partition of C, $\tilde{A}$, and $\tilde{B}$ into $m_r \times n_r$, $n_r \times k_r$, and $k_r \times n_r$ sized blocks as γ, α, and β, respectively.

In the new algorithm we keep γ and α in registers distributed across register banks and β in the L1 cache. This new register blocking amortizes the cost of bringing γ to and from registers over the multiplication of an $n_r \times k_b$ panel of $\tilde{A}$ by a $k_b \times n_r$ slice of $\tilde{B}$. It keeps α in registers and streams β from memory, multiplying α by each column of β and updating the corresponding column in γ. In a spatially partitioned processor, we can multiply α by each column of β very efficiently by exploiting the spatial layout of the grid of DPUs and network links in a systolic fashion. However, the kernel must be designed for minimum contention in the network. For example, α and β values should arrive in each DPU with minimum delay, and after local computation, the results sent to a neighbor DPU through uncontested links.

Relationship to higher level blocking: Because this algorithm repeatedly uses one slice of $\tilde{B}$, we try to keep it in the L1 cache while γ blocks of the current slice of C are being updated. One optimization that achieves this goal is limiting the area of a slice of $\tilde{B}$ to half of the L1 cache. Figure 4 displays a high level representation of the algorithm and shows the way that the algorithm traverses a panel of $\tilde{A}$ and a slice of $\tilde{B}$ to update a γ block.

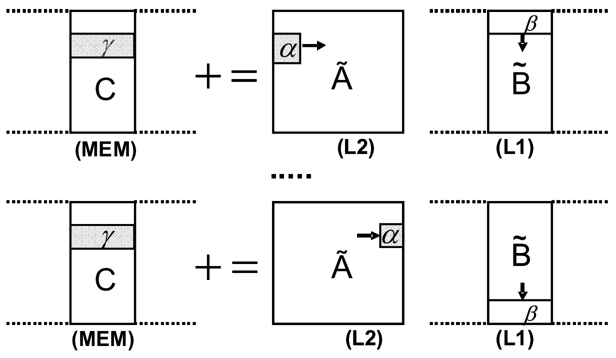


Figure 4. High level representation of the new DGEBP designed for spatial processors. Shaded areas represent values kept in register blocks. The figure illustrates a block of γ being computed as a dot product between a row of α blocks and a slice of $\tilde{B}$.

The width of β and n_r determines the dimensions of $\tilde{A}$ and slices of $\tilde{B}$ in L1 cache. The choice of n_r has the following performance implications:

- Increasing the width of β and n_r can amortize α over more columns of β, but n_r is also the width of γ and is limited by the total number of registers: $m_r n_r (\mathit{sizeof}\,\gamma) + m_r k_r (\mathit{sizeof}\,\alpha) < \mathit{TotalNumberofRegisters}$ Also, limiting each slice of $\tilde{B}$ to no more than half of the L1 cache capacity bounds the values of k_b and n_r: $n_r k_b = L1_{size}/2$
- Increasing the height of $\tilde{B}$ (m_b) can amortize loading and storing of γ over a larger amount of computation, but m_b is also the height of $\tilde{A}$ which cannot be increased beyond where $\tilde{A}$ fills half of L2 cache: $m_b k_b = L2_{size}/2$

3.2 Customizing the Algorithm for TRIPS

To customize our general DGEBP algorithm to match the specific topology of the TRIPS processor, we use the guiding principles above to choose the best possible sizes of α, β, and γ.

3.2.1 Design of the Innermost Kernel

3.2.2 Effect on Higher Level Blocking

Selecting register block sizes: The first step is to choose the optimal size of the register blocks. Mathematically, we desire to maximize the ratio of computation, which is $2m_r n_r k_r$, divided by the required bandwidth, which depends on how fast blocks are being

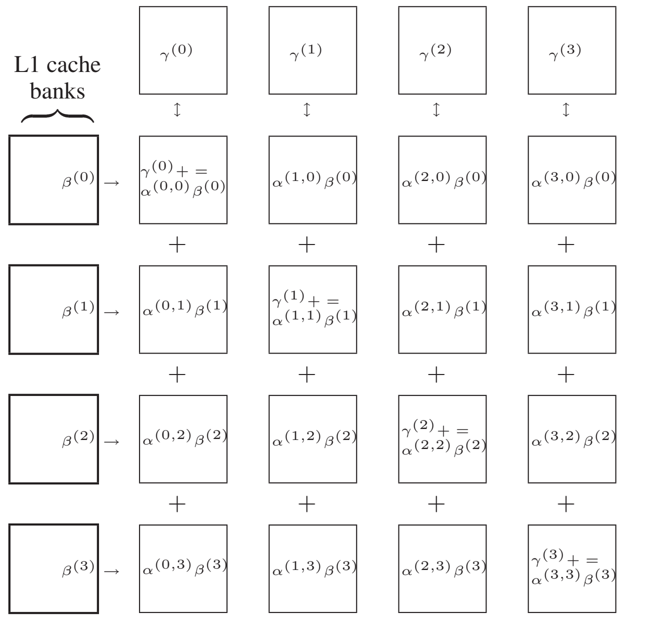

L1 cache banks
$\gamma^{(0)}$
$\gamma^{(1)}$
$\gamma^{(2)}$
$\gamma^{(3)}$
$\beta^{(0)}$
$\gamma^{(0)}+=\alpha^{(0,0)}\beta^{(0)}$
$\alpha^{(1,0)}\beta^{(0)}$
$\alpha^{(2,0)}\beta^{(0)}$
$\alpha^{(3,0)}\beta^{(0)}$
$\beta^{(1)}$
$\alpha^{(0,1)}\beta^{(1)}$
$\gamma^{(1)}+=\alpha^{(1,1)}\beta^{(1)}$
$\alpha^{(2,1)}\beta^{(1)}$
$\alpha^{(3,1)}\beta^{(1)}$
$\beta^{(2)}$
$\alpha^{(0,2)}\beta^{(2)}$
$\alpha^{(1,2)}\beta^{(2)}$
$\gamma^{(2)}+=\alpha^{(2,2)}\beta^{(2)}$
$\alpha^{(3,2)}\beta^{(2)}$
$\beta^{(3)}$
$\alpha^{(0,3)}\beta^{(3)}$
$\alpha^{(1,3)}\beta^{(3)}$
$\alpha^{(2,3)}\beta^{(3)}$
$\gamma^{(3)}+=\alpha^{(3,3)}\beta^{(3)}$

Figure 5. Mapping of operations into the TRIPS FPUs for updating a column of γ by multiplying a column of β with a block of α in registers, where superscripts denote elements.

loaded, constrained such that we can fit needed blocks in registers. For example, if γ, α, and β must be loaded every time (ignoring writes), then the bandwidth is $m_r n_r + m_r k_r + k_r n_r$, and the ratio is maximum when $m_r = n_r = k_r$, that is, square blocks are optimum. However, if only one block is reloaded, then rectangular dimensions which minimize the size of that block create a more favorable ratio. In our algorithm, we stream β the fastest and α at a lower rate, while γ can be viewed as stationary, leading to a bandwidth cost of $m_r k_r + k_r n_r$ and an optimal area for α that is significantly smaller than β . However, Section 5.5 describes in detail that this optimization depends not only on relative bandwidth but also on relative register usage (register foot print). Maximizing all these ratios while constraining size to available register storage yields a mathematically ideal size of 7×14 for γ.

However, we learned quickly that the cost of not mapping to the natural 4×4 topology of TRIPS more than offsets any performance gains, so the next closest match was a size of 4×16 for γ, which implied a size of 4x4 for a block of α, and a width of 16 for each panel of β. Note that this has the optimum computation/bandwidth ratio of any possible multiple-of-four sizes, and its ratio is still almost 70% that of the mathematically ideal size, while using only 2/3 of the registers.

Mapping to the grid: In the case of TRIPS, the data banks and register banks are connected orthogonally to the computation grid as shown Figure 3, providing a data flow of A and B elements similar to systolic arrays. Arranging the data for systolic computation is obvious: each register bank is assigned one successive column of the transpose of α and γ. Similarly, each column of β is stored on successive TRIPS L1 cache banks. As we send values of α from registers to corresponding DPUs, we load each column of β from the data banks, broadcasting elements of it across each row of DPUs. Next, we multiply β by α at each DPU, and add the results up each column, updating each column's γ.

Figure 5 shows the mathematical notation of this method and identifies which values of α and β participate in computing a value of γ in each column. In this figure the four boxes on the top are register banks holding the values of γ. The 4×4 grid in

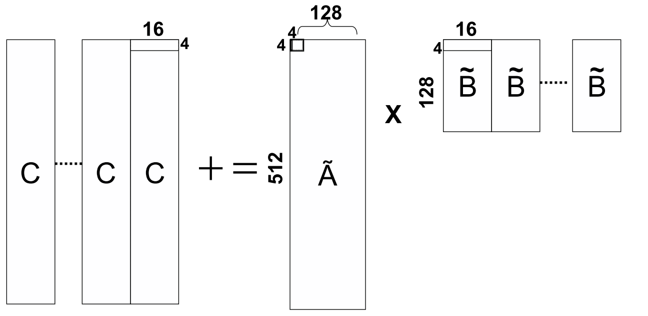


Figure 6. Typical blocking sizes for the TRIPS processor

the middle represents DPUs. For each DPU, the figure shows the corresponding α value, read from the register bank in that column, and β value, loaded from L1 data bank in the same row. This pattern makes excellent usage of the network links because fetching values of α and β into DPUs uses south bound and east bound links while sending load addresses for β and reducing sums of γ uses only west bound and north bound links. Sixteen of the remaining registers are used for double buffering α, loading the next α block while multiplying the current α by β.

On a partitioned core communicating over a network, the proper routing of values to minimize network contention and latency is the most important aspect of optimized kernel design – a factor that is completely absent from conventional kernels, but which will likely be increasingly important to kernel design in future on chip architectures. In our case, we relied heavily on the concept of *data path routing*. This approach first lays out uncontested data paths between registers and data banks. Then, all the computation that would operate on this data stream are attached to the path. Finally, TRIPS has eight instruction blocks in flight and competing for the shared resources such as registers, L1 caches, DPUs and the operand network. To minimize inter-block contention, we have all instructions in one data path operate on the data like an assembly line.

The design of the innermost kernel and register block sizes affects both the sizes of the higher level blocks in DGEBP and the way data must be packed into those blocks in DGEMM.

High level block size: Having determined the optimum size of register blocks α and panels of γ and β, we can determine the size of the buffers in Goto's high level algorithm [1], namely the block of $\tilde{A}$ stored in the L2 cache and the slice of $\tilde{B}$ stored in the L1 cache. As described in Section 3.1, filling half of the L1 cache with a slice of $\tilde{B}$ and half of the L2 cache with $\tilde{A}$ produces the best reuse. Figure 4 shows that for the innermost kernel to reuse the register panel, γ, it must use all of the panels of β in a vertical column. Given the TRIPS L1 cache size of 32KB and the width β being 16 values (each eight bytes), the height $\tilde{B}$ is set to 128 values, which must also be the width of $\tilde{A}$. Since the TRIPS L2 cache size is 1MB and we want to fill half of it with $\tilde{A}$, the height of the $\tilde{A}$ and the height of $\tilde{C}$ is 512. Figure 6 summarizes different block sizes calculated for the improved DGEBP algorithm optimized for TRIPS processor.

Effect on packing: As mentioned before, Goto's algorithm [1] requires packing for $\tilde{A}$ blocks and $\tilde{B}$ panels. Simultaneously leveraging all cache banks in our innermost kernel can quadruple memory bandwidth, reduce operand network contention, and help load balance the entire computation grid. The price is a slightly more complicated packing algorithm and the need to pack $\tilde{C}$ in addition to $\tilde{A}$ and $\tilde{B}$. Figure 7 illustrates a sample cache interleave packing of two consecutive α blocks and β columns. The γ columns are packed similarly to β.

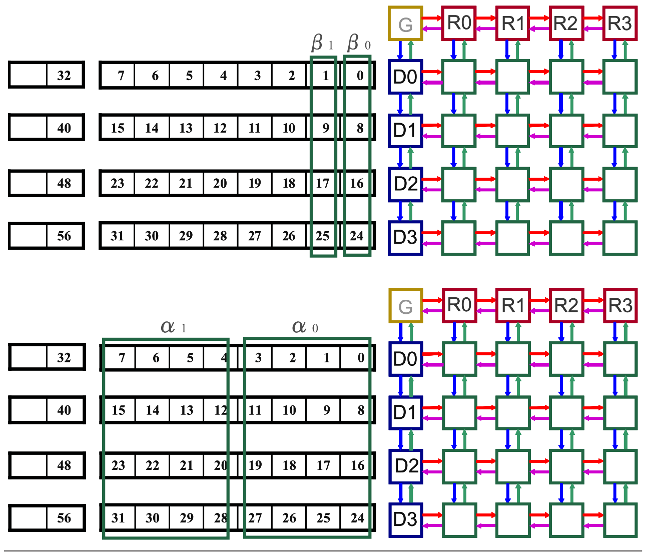


Figure 7. Data packing model for α and β blocks using in DGEBP algorithm customized for TRIPS.

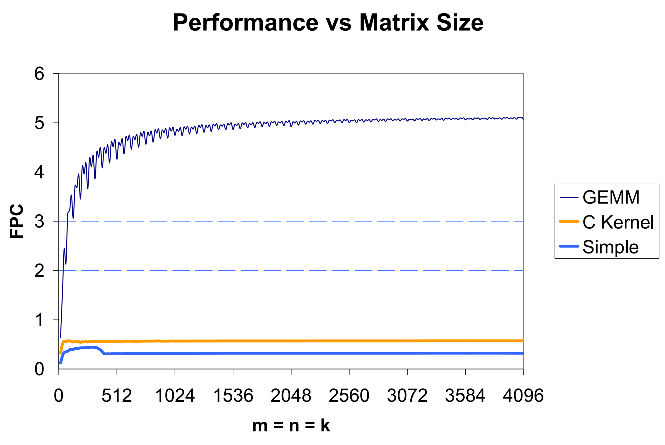


Figure 8. Different implementations of the Matrix Multiply with or without GEBP layer, inner kernel and the packing code

4. Experimental Results

To evaluate the algorithmic modifications for distributed processor cores, we used actual TRIPS hardware with basic characteristics shown in Table 1 [7]. The TRIPS prototype processor chip is a custom 170 million transistor ASIC implemented in a 130nm technology. We collected cycle counts from the hardware performance counters using customized libraries and a runtime environment developed by the TRIPS team. The algorithm is implemented in C and assembly language and compiled using the TRIPS custom compiler [21]. Section 4.1 includes a standard performance analysis and Section 4.2 compares our results to implementations of Goto's algorithm on other processors. All the results in the section are shown in FLOPS per cycles (FPC).

Table 1. TRIPS Chip Parameters.

L1 cache size	L2 cache size	SDRAM size	Processor speed
32KB	1MB	2GB	366MHz

4.1 Performance Results and Analysis

Performance vs Matrix Size: Figure 8 shows how performance (measured in FPC) varies in response to changes in the size of A, B and C matrices, for three different kernel implementations:

- *DGEMM*: The full matrix multiply application including packing, a DGEBP layer, and our inner kernel customized for TRIPS based on the algorithm described in previous sections.
- *C Kernel*: Identical to above *DGEMM* implementation, including packing and a DGEBP layer, but with the innermost kernel implemented in C with no explicit spatial positioning of instructions.
- *Simple*: A classic *triple loop* implementation of the matrix multiply.

The performance of *Simple* grows gradually and drops at the point where matrix sizes outgrow the L2 cache near 512×512 matrix size. However, *C Kernel* does not drop for any matrix sizes beyond that point due to the explicit data management of the DGEBP. *DGEMM* starts ramping up soon and performs at peak at almost the same point where the *Simple* code drops. The fluctuations seen by the complete *DGEMM* timings are due to the unoptimized nature of the packing code. Another interesting observation is that while *C Kernel* and *DGEMM* share identical code except for the innermost kernel, the 10X performance difference demonstrates the impact of data placement and spatial layout of instructions on performance.

Packing Overhead: Figure 9 illustrates the performance lost due to the packing overhead by showing timings for:

- *No-Packing*: *DGEMM* code without re-arranging the data during packing, i.e., we pre-arrange the data so that no packing is required.
- *DGEMM* : The full matrix multiply application.
- *Remainder*: The packing overhead computed by taking the difference between No-Packing and *DGEMM*.

The small variation in the *No-Packing* performance is due to fringe overhead where matrix sizes not evenly divisible by the buffer size decompose into small remainder blocks. Packing overhead is roughly 5% once the size reaches 1024×1024, falling to less than 1% at 4096×4096 matrix sizes. As previously mentioned, our high packing overhead at smaller matrix sizes is due to our unoptimized implementation of packing which cannot take advantage of all four available data banks. Additionally, packing is more complex for distributed caches than for single-banked caches due to the interleaving of cache lines across the four banks.

The Effect of the Shape of $\tilde{A}$: To examine our choice for the dimensions of the $\tilde{A}$ buffer (512×128, which is narrow and tall), we varied the aspect ratio of $\tilde{A}$ while keeping its size at half of the L2 cache. As shown in Figure 10, both square and wide $\tilde{A}$ buffers have lower performance, with the wide buffer performing the worst. Our observations confirm Goto's assertion that a tall and narrow $\tilde{A}$ typically offers better performance.

Performance on the Thin Matrices: Dense linear algebra libraries like LAPACK [2] can cast algorithms most typically in terms of DGEMM where the k dimension is relatively small (also known as a rank-k update). Figure 11 shows the performance for different values of k when m and n are held constant at 4,096. Performance ramps up with k as low as 100, which is important since this allows algorithms that cast computation in terms of rank-k updates to achieve high performance for smaller problem sizes.

Table 2. Comparing our performance with select performance numbers taken from [1], Figure 12

Processor	Kernel FPC	DGEMM FPC[1]	# of Reg[2]	γ dim	$\tilde{A}$ dim	L2 cache[3]
Opteron-EM64T	1.88	1.79	16x2	4x4	384x256	1024
P4-Prescott-EM64T	1.92	1.87	16x2	4x4	696x192	2048
Core2 Duo	3.68	3.58	16x2	4x4	512x256	4096
POWER5	3.84	3.78	32	4x4	256x256	1920
Itanium2	3.96	3.92	128	8x8	128x1924	256
TRIPS	5.80	5.10	128	4x16	128x512	1024

[1]Flops Per Cycle measured at m=n=k=2000, [2]"x2" denotes 2-way SIMD register storage, [3]L2 cache size in kilobytes

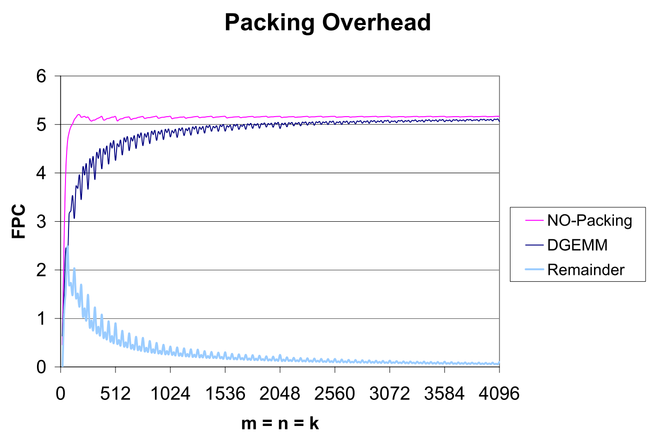


Figure 9. The packing overhead and the performance ideal kernel with no packing overhead.

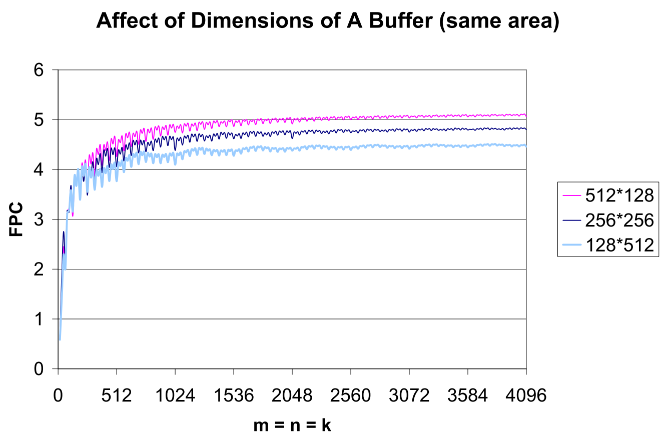


Figure 10. The effect of different $\tilde{A}$ buffer dimensions.

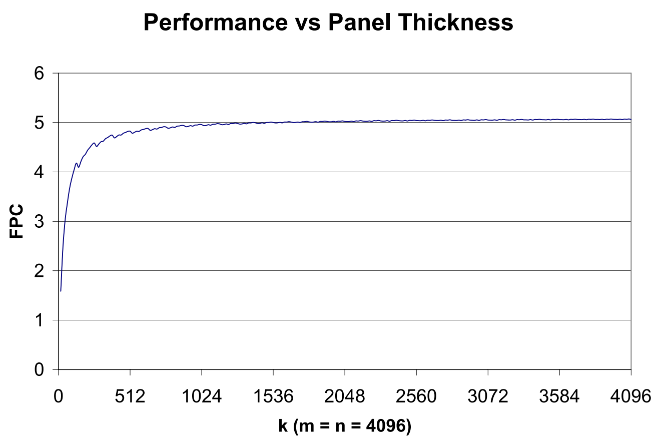


Figure 11. Varying k while keeping m and n are constant.

4.2 Comparative Analysis

In this section, we compare the performance characteristics of our algorithm for spatial processors to that of Goto's algorithm on conventional processors [1].

The simplest metric for evaluating the performance of DGEMM is absolute performance in gigaflops/second during computation of a large matrix multiply. However, DGEMM is also used to implement the Basic Linear Algebra Subroutines, and these routines break up matrices into smaller blocks of different shapes. For this reason, other metrics are important in evaluating DGEMM performance:

- ramp-up speed: how large must a matrix be to reach peak performance.
- aspect ratio: how thin can a vertical panel become before causing a performance penalty.
- smoothness: how much does performance vary for incremental changes in matrix size.

However, this paper focuses on the implementation of the high speed inner kernel on spatially distributed uniprocessors. As seen in Figure 9, when removing the influence of unoptimized packing code, our kernel exhibits sufficient ramp-up speed and smoothness.

An important metric that accompanies any matrix multiply paper is the theoretical peak floating point performance of the processor, but TRIPS does not have a clear peak Flops Per Cycle (FPC) because unlike conventional processors, a DPU must explicitly trade off floating point computations with data replication and movement across the network as well as loads, stores, prefetching and integer operations.

Table 2 compares the performance of our full DGEMM implementation to other implementations of Goto's algorithm on conventional processors. Because the TRIPS prototype is an ASIC implemented on 130nm, we compare FPC instead of absolute performance in gigaflops/second [1]. We measure "peak FPC" at a matrix size of 2000 to compare directly with [1]. Results indicate that TRIPS performance ranges from 1.30x to 2.85x times the peak FPC

[1] On the current prototype, our sustained performance is 1.9 gigaflops. However, spatially partitioned processors are specifically designed for extremely high megahertz/low power performance. For a conservative full-custom 90nm VLSI implementation at 4 GHz with 8 cores, our algorithm would sustain over 20 gigaflops per core.

of the five most significant conventional processors in the study. This improvement is even more significant considering that it was accomplished solely with general purpose instructions, whereas most other implementations leverage specialized SIMD units in addition to general purpose resources. Also note that the chosen optimal register blocking was generally different, reflecting major differences between the architectures.

5. Discussion

During the course of our development, we made several interesting observations on the algorithms and architectures that are worthy of extra discussion.

5.1 Register Resources on Distributed Uniprocessors

On conventional processors, register bandwidth is essentially infinite (being twice the instruction bandwidth) and so is considered free. But on a distributed processor of the topology we have described, a 2-D grid of DPUs borders a 1-D array of register banks, and so all the DPUs in a column of the grid must share serial access to a single register bank. The result is an imbalance of register-file to computation bandwidth which increases with the size of the grid, although partially mitigated via local communication between DPUs. For this reason, highly parallel algorithms like matrix multiply must be designed to conserve register bandwidth as well as cache bandwidth.

5.2 Designing an Optimized Algorithm for a Grid Topology

Section 3.2.1 discussed a set of optimization principles for designing an grid-topology aware algorithm that can be summarized as:

- Minimize network contention: excess contention for even a single link could cut code performance in half.
- Balance use of grid resources: A single resource bottleneck can easily dominate performance, and on a spatially distributed uniprocessor, the opportunity cost is high – when one resource is used more than the rest, every network link and DPU execution slot in the 2-D grid goes idle.
- Utilize all register banks and L1 cache banks: use appropriately strided data access to simultaneously access all cache banks. Mirror common register values across all register banks.

Because any violation of these principles can adversely affect performance, there is a degree of fragility to optimizations akin to what parallel programmers experience. We did find, however, that creating optimized code for a distributed processor is surprisingly simple. Straightforward instruction layout techniques like datapath routing simultaneously satisfy all of the optimization requirements.

5.3 Implementation Details

During the implementation of the kernel, we used the optimization principles introduced in the previous subsection to overcome architectural bottlenecks.

Instruction Block Formation: As mentioned before, a TRIPS program is a series of instruction blocks which are executed atomically. A TRIPS processor can execute up to eight instruction blocks simultaneously. In this pipeline, the maximum instruction block completion rate (BCR) is clamped to one instruction block per 8 cycles. Therefore, each instruction block should contain as as much actual computation as possible. Instruction block formation (deciding what assembly instructions to place in an execution block) is the single most important optimization step, responsible for up to 75% of the performance of the kernel. The goals of instruction block formation are mapping the algorithm described in Section 3 to the TRIPS topology, minimizing instruction overhead (code quality), facilitating interleaving of instruction blocks within the pipeline.

At this stage in the optimization, only conventional style assembly was used, relying on a high performance scheduler to position all of the instructions along the 2-D grid automatically. Getting further performance requires manual 2-D placement of the instructions.

Avoiding Network Contention: The next fundamental step after instruction block formation is to avoid network contention between ALUs. A simple method we use to layout a 2-D schedule that both minimizes network contention and facilitates instruction block interleaving is called data path routing. In this optimization, we think of an instruction block's operations as parallel data paths in each column of the grid. Each data path performs the dot-product of one of the B vectors and a column of the A sub-matrix and adds the results back to the corresponding C element in that register bank. Each of these parallel data paths works as an assembly line operating on the data stream. This pattern requires the B vectors read from L1 banks to be broadcast to all four columns of the DPU grid. Because TRIPS has no general broadcast mechanism, move trees are used to duplicate values. By utilizing a *mov4* instruction that replicates its source to four restricted targets, we reduce move fanout and achieve 12% improvement. Remaining fanout can be eliminated by a technique called register mirroring, in which the same value is put in multiple registers across all banks. As an example, the address values for loading elements of C are fetched from registers. If all loads fetch their addresses from the same register, it causes fanout and network contention. This optimization increased performance by 30%.

Load Balancing and Block Latency: The optimization methods explained so far aim to improve useful instructions per block and end-to-end block execution. However, another important factor affecting the performance is the maximum use of a single physical resource per block. Since TRIPS maximum block completion rate (BCR) is one block every 8 cycles and 8 blocks can be in flight, exceeding 64 cycles block latency (about half being block overhead) reduces BCR rate by (64/average latency). Another major factor affecting BCR relates to load balancing between the 8 blocks in flight. If a block (on average) uses any single physical resource (network link or execution slot) more than 8 active cycles then the BCR is reduced by a factor of 8/(the max number of uses in a block). Once the basic data path was laid out, load balancing and minimizing block latency to the extent possible was trivial. Considering this rule, we realized that our schedule utilized the south bound links under the register tiles 12 active cycles instead of 8. For each of those links, 4 of these cycles are spent on reading the B vector addresses and C vectors' elements and the remaining 8 cycles are needed to read a row of A sub-matrix twice, once for each of the two data path mapped to the DPU column below that register file. We had to read each A element twice because there is no local storage (register or memory) in the DPUs. Because of this imbalance in the use of Southbound links, the code ran at exactly 2/3rds the expected speed. We fixed this problem by replacing the extra 4 register reads with one local replication in each DPU, which led to a 30% boost in performance. Table 3 shows different levels of optimization and the performance associated with them.

Although these optimizations used in the inner kernel are implemented in assembly language, and the current TRIPS compiler does not store arrays in registers, such transformations are within the realm of standard blocking/tiling compilers, and could be easily added to the compiler. Data path routing can be automated in the compiler by detecting parallel paths with minimal communication between them in instruction blocks and mapping each path onto a section of the grid. Finally, achieving a low contention schedule by the compiler is simple by allocating separate data paths in the Instruction Block, interleaving (overlapping) data movement and computation (*mov* instructions).

Optimization	FPC
Block Formation and Data Placement	3.9
Data-path Routing and Load Balancing	5.7
Extending to L2 and SDRAM (C compiler)	5.2

Table 3. Performance at different levels of optimization.

5.4 Hardware Recommendations

The purpose of this paper was to examine techniques required to obtain high performance on a distributed uniprocessor architecture, rather than examine the design space of the architecture. However, during the course of developing the inner kernel we came across some challenges which could be greatly ameliorated by some additional support in the TRIPS hardware, and result in a significantly higher performance:

- **Local Storage in ALUs:** Each instruction block must intially read a 4x4 sub-matrix of A from registers and use replication instructions for further reuse within the same instruction block. Unrestricted access to local "constant" registers in each ALU would allow the read-only A matrix to be accessed in place, greatly reducing register bandwidth and overall latency. Fewer copy and read instructions would allow more computations per instruction block, increasing performance by 50% or more. Unrestricted writes to local registers would additionally allow in place accumulation of an entire 4x4 C matrix.
- **Base Address Registers:** Reading any values from memory requires using an address from the register bank for each load command. This overhead consumes already low register bandwidth, increases the average latency of load instructions, and generates disruptive operand network traffic. By extending the natural dataflow model to include loads as well as register reads, static load commands and base address registers could be stored in peripheral data banks. This would not only reduce register pressure and network congestion but would decouple memory bandwidth from register bendwidth.
- **Efficient Broadcast Mechanism:** TRIPS currently "broadcasts" a value by having the DPUs generate multiple packets for multiple destinations. This replication completely occupies consecutive DPU execution cycles, making interleaving instruction blocks difficult and displacing useful instructions. It also generates more network traffic than necessary and creates local wire hot spots. Replicating packets in routers would solve these issues as well as reduce block latency.
- **Dual Operand Networks:** Most instructions require two inputs. In addition to limited register bandwidth, the TRIPS prototype can only admit one external value into a DPU per cycle, which in the worst case can cut the execution rate in half. Doubling the operand network and the number of reads per instruction block would better support the existing number of computational elements.

5.5 The Optimum Shape of Register Blocks

The *mathematically ideal size* for register blocks is defined as one that maximizes the ratio of computation to bandwidth. During development, we were able to extend the mathematical analysis of ideal block size beyond previous discussions such as [1] and demonstrate results that would have previously been considered counter-intuitive.

Summarizing the derivation from [1], if blocks α, β, and γ are each loaded for every block multiply, then the computation is proportional to $2m_r n_r k_r$, and the bandwidth is proportional to $m_r n_r + n_r k_r + m_r k_r$. Maximizing this ratio (and ignoring the scaling factor of 2) can be visualized as a parallelepiped with dimensions m_r, n_r and k_r, where the volume is the computation and the surface area is the bandwidth. The ratio is maximum when the parallelepiped is a cube and all blocks are square. If γ must be read *and* written for each multiply, its bandwidth carries twice the weight as the others. Reducing the size of γ reduces bandwidth demand, stretching the parallelepiped along the k_r axis, yielding a square γ but rectangular α and β blocks as optimum. Our analysis generalizes this result by accounting for different *bandwidth weights* on all three blocks. Solving this system yields an ideal aspect ratio for the block sizes that can then be scaled to fit in available register space.

We found it necessary to simultaneously extend this optimization analogy along a new axis. Because some blocks in our algorithm must fit into registers, there is pressure to reduce the *register footprint* of those blocks. Since we store two copies of α in registers (for double buffering) and no copies of β, each block must have a different weight attached to its register footprint in the analysis. *Register footprint weight* acts on the parallelpiped in a similar manner to *bandwidth weight* – a large register footprint will exert influence to make the block smaller. This effect was not observed in earlier studies because only one block was stored in registers or all blocks have equal weight.

In our case, each block's register footprint weight is almost directly opposed by its bandwidth weight[2], so there is no single ideal aspect ratio; instead, aspect ratio depends on the absolute size of the block. Thus the ideal shape of a block depends on its size and no single aspect ratio can be considered mathematically ideal. As a result, in an exhaustive search of all possible register block sizes, the sets with the highest ratios of computation to bandwidth included shapes similar to both outer products and inner product computations.

Practical algorithms must also consider other factors, such as the natural topology of the processor, the absolute amount of computation per block multiply, and the total number of registers required.

6. Conclusion

We examined the adaptation of a high-performance matrix multiply algorithm to an emerging class of distributed spatial processors. As with systolic architectures, the two-dimensional nature of matrix multiply is a good match to such planar arrays of processing elements. Our implementation on the TRIPS processor shows a high performance of nearly 6 Flops/cycle, which is 1.30 to 2.85 times the best performance of conventional architectures, and validates the viability of spatially distributed uniprocessors as a general purpose path to future high performance in technology scalable systems. As a result of the optimization techniques we discovered, our implementation of DGEMM is currently the highest performing software on the TRIPS hardware prototype.

Algorithmically, we found that Goto's streaming approach continues to work well for moving data through the cache hierarchy, but algorithms for distributed uniprocessors require some alterations. Our contributions include the incorporation of an extra level of matrix decomposition that makes better algorithmic use of large register files and systolic-like communication, an analysis of optimum block size in terms of general bandwidth and register footprints, and the design and mapping of multiple vector-matrix multiplies across the spatial substrate to maximize resources and minimize network contention in the innermost code. As a result, we

[2] In our algorithm, the bandwidth weights for γ, α, and β are 0, 1, and 4 respectively (γ is considered stationary), while the register footprint weights are 1, 2, and 0 (α is stored twice and β is not stored).

have succesfully implemented a systolic-like algorithm on a general purpose processor.

We found that the costs of network communication versus local computation in on-chip networks makes performance sensitive to small changes in network and execution unit contention. Nonetheless, achieving high performance was relatively straightforward once we determined the relevant resource and algorithmic constraints. Our experience indicates that well-engineered algorithms will continue be relevant to emerging architectures that we anticipate seeing in the future.

Acknowledgments

We would like to thank all the people who gave us valuable feedback including the anonymous reviewers.

This research is supported by the Defense Advanced Research Projects Agency under contract F33615-01-C-4106 and NSF grant CCF-0540926. *Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).*

References

[1] K. Goto and R. A. van de Geijn, Anatomy of High-Performance Matrix Multiplication, ACM Transactions on Mathematical Software, 2008.

[2] E. Anderson and Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, D. Sorensen, LAPACK's user's guide, Society for Industrial and Applied Mathematics, 1995.

[3] K. Goto and R. van de Geijn, On Reducing TLB Misses in Matrix Multiplication, FLAME Working Note #9, The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2002-55, November 2002.

[4] S. Swanson, K. Michelson, A. Schwerin and M. Oskin, WaveScalar, International Symposium on Microarchitecture, 36:291, December 2003.

[5] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee and J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe and A. Agarwal, Baring It All to Software: RAW Machines, IEEE Computer, volume 30(9):86-93, September 1997.

[6] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder and the TRIPS Team, Scaling to the End of Silicon with EDGE Architectures, IEEE Computer, 37(7):44-55, July 2004.

[7] K. Sankaralingam, R. Nagarajan, P. Gratz, R. Desikan, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, W. Yoder, R. McDonald, S.W. Keckler and D.C. Burger, The Distributed Microarchitectural Protocols of the TRIPS Prototype Processor, International Symposium on Microarchitecture, 39:480-491, November 2006.

[8] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote and N. Borkar, An 80-Tile 1.28 TFLOPS Network-on-Chip in 65nm CMOS, IEEE International Solid-State Circuits Conference, February 2007.

[9] M. B. Taylor and A. Agarwal, Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams, ACM SIGARCH Computer Architecture News, 32(2):2, March 2004.

[10] M. B. Taylor and A. Agarwal, The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs, IEEE Micro, 22(2), March 2002.

[11] H. T. Kung and C. E. Leiserson, Algorithms for VLSI processor arrays, Introduction to VLSI Systems; Addison-Wesley, 1979.

[12] M. Kunde, H.W. Lang and H. Schmeck and H. Schrder, The Instruction Systolic Array and its Relation to Other Models of Parallel Computers, Parallel Computing, 7:25-39, April 1988.

[13] John A. Gunnels, Greg M. Henry and Robert A. van de Geijn, A Family of High-Performance Matrix Algorithms, Computational Science Part I, Lecture Notes in Computer Science, 2073:51-60, 2001.

[14] B. Marker, F. Van Zee, K. Goto, G. Quintana-Orti and Robert van de Geijn, Toward Scalable Matrix Multiply on Multithreaded Architectures, Euroropean Conference on Parallel Processing, 13:748-757, August 2007.

[15] J. Gunnels, C. Lin, G. Morrow and R. van de Geijn, A Flexible Class of Parallel Matrix Multiplication Algorithms, First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing, 12:110-116, March 1998.

[16] S. Chen, P. B Gibbons, M.l Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry and C. Wilkerson, Scheduling Threads for Constructive Cache Sharing on CMPs, ACM Symposium on Parallelism in Algorithms and Architecture, 19:105-115, June 2007.

[17] J. J. Dongarra, J. Du Croz, S. Hammarling and I. Duff, A set of level 3 basic linear algebra subprograms, ACM Transactions On Mathematical Software, volume 16(1):1-17, March 1990.

[18] K. Goto and R. van de Geijn, High-performance implementation of the level-3 BLAS, FLAME Working Note #20 TR-2006-23, The University of Texas at Austin, Department of Computer Sciences, 2006.

[19] R. C. Whaley and J. J. Dongarra, Automatically tuned linear algebra software, ACM/IEEE conference on Supercomputing, pp 1-27, November 1998 .

[20] R. C. Whaley, A. Petitet and J. J. Dongarra, Automated empirical optimization of software and the ATLAS project, Parallel Computing 27(1-2):3-35, 2001.

[21] A. Smith, J. Gibson, B. Maher, D. Burger, K. S. McKinley and J. Burrill, Compiling for EDGE Architectures, International Symposium on Code Generation and Optimization, pp 185-195, March 2006.

[22] J. Laudon and L. Spracklen, The Coming Wave of Multithreaded Chip Multiprocessors, International Journal of Parallel Programming, volume 35(3):299-330, June 2007.