Floating-Point Sparse Matrix-Vector Multiply for FPGAs

Thesis by

Michael deLorimier

In Partial Fulfillment of the Requirements
for the Degree of
Master of Science

California Institute of Technology
Pasadena, California

2005
(Submitted May 13, 2005)

Acknowledgements

Thanks to my adviser André DeHon for valuable guidance on the construction and presentation of this project.

Abstract

Large, high density FPGAs with high local distributed memory bandwidth surpass the peak floating-point performance of high-end, general-purpose processors. Microprocessors do not deliver near their peak floating-point performance on efficient algorithms that use the Sparse Matrix-Vector Multiply (SMVM) kernel. In fact, microprocessors rarely achieve 33% of their peak floating-point performance when computing SMVM. We develop and analyze a scalable SMVM implementation on modern FPGAs and show that it can sustain high throughput, near peak, floating-point performance. Our implementation consists of logic design as well as scheduling and data placement techniques. For benchmark matrices from the Matrix Market Suite we project 1.5 double precision Gflops/FPGA for a single VirtexII-6000-4 and 12 double precision Gflops for 16 Virtex IIs (750Mflops/FPGA). We also analyze the asymptotic efficiency of our architecture as parallelism scales using a constant rent-parameter matrix model. This demonstrates that our data placement techniques provide an asymptotic scaling benefit.

While FPGA performance is attractive, higher performance is possible if we re-balance the hardware resources in FPGAs with embedded memories. We show that sacrificing half the logic area for memory area rarely degrades performance and improves performance for large matrices, by up to 5 times. We also 0 the performance effect of adding custom floating-point using a simple area model to preserve total chip area. Sacrificing logic for memory and custom floating-point units increases single FPGA performance to 5 double precision Gflops.

Contents

Chapter 1

Introduction

Peak floating-point performance achievable on FPGAs has surpassed that available on microprocessors [12]. Further, memory bandwidth limitations prevent microprocessors from approaching their peak floating-point performance on numerical computing tasks such as Dense Matrix-Vector Multiply (DMVM) due to large memory bandwidth requirements. Consequently, modern microprocessors deliver only 10–33% of their peak floating-point performance to DMVM applications [13]. Delivered performance per microprocessor is even lower in multiprocessor systems. Sixteen microprocessors in parallel rarely achieve 5% peak. In contrast, high, deployable, on-chip memory bandwidth, high chip-to-chip bandwidth, and low communications processing overhead combine to allow FPGAs to deliver higher floating-point performance than microprocessors in highly parallel systems.

We investigate SMVM on the VirtexII-6000-4. On a single microprocessor, SMVM performs somewhat worse than DMVM due to data structure interpretation overhead. In our FPGA implementation (Chapter 2), data structure interpretation is performed by spatial logic, incurring less overhead than on a microprocessor. Loads and stores are streamed, so the computation does not stall between load issue and data arrival. We use local on-chip BlockRAMs exclusively which gives us a further performance advantage from high memory bandwidth. Our design on one FPGA has somewhat higher performance than the 900MHz Itanium II, which is the fastest of microprocessors released in the same period. The performance gap increases when scaled to multiple processors: for 16 processors our design runs at 1/3 peak (750 Mflops/FPGA out of 2240 Mflops/FPGA (Chapter 3)). This is a factor of three higher than 16 processor, microprocessor-based parallel machines. Our design scales to 48 FPGAs before Mflops/FPGA drops below half of single FPGA Mflops.

Novel contributions of this work include:

- Architecture designed for SMVM for large matrices on multi-FPGA systems
- Parameterized mapping strategy that allows deep pipelines
- Analysis and characterization of scalability
- Demonstration of feasibility of sparse matrix routines on modern FPGAs
- Exploration of FPGA architectures balanced for this application

	sequential in dot product	parallel in dot product
sequential over dot products	single microprocessor **1**	Zhuo, Prasanna [17] **100**
parallel over dot products	this work typical parallel microprocessor implementation **10,000**	**1,000,000**

Table 1.1: Types of parallelism. Numbers indicate potential parallel operations for a typical 1,000,000 entry matrix.

1.1 Sparse Applications

Many real life numerical problems in applications such as engineering simulations, scientific computing, information retrieval, and economics use matrices where there are few interactions between elements and hence most of the matrix entries are zero. For these common problems, dense matrix representations and algorithms are inefficient. There is no reason to store the zero entries in memory or to perform computations on them. Consequently, it is important to use sparse matrix representations for these applications. The sparse matrix representations only explicitly represent non-zero matrix entries and only perform operations on the non-zero matrix elements. Further, sparse parallel algorithms often take advantage of matrix locality to perform much less communication on parallel machines than their dense counter parts.

Sparse Matrix-Vector Multiply (SMVM) is one of the most important sparse matrix problems. SMVM is primarily used in iterative numerical routines where it is the computationally dominant kernel. These routines iteratively multiply vectors by a fixed matrix. Examples that solve $Ax = b$ are GMRES, Conjugate Gradient (CG), and Gauss Jacobi (GJ) [10]. Examples that solve $Ax = \lambda x$ are Arnoldi and Lanczos [10].

The specific problem we solve is iterative SMVM, which finds $A^i b$ by performing SMVM repeatedly with a square matrix. We take it as a representative of both the implementation and performance of iterative numerical routines. The extra computations besides SMVM in the routines CG, GJ and Lanczos are a few vector-parallel operations, which require little work and communication compared to the matrix multiply (See Section 3.7).

1.2 Parallelization

The advantage of FPGAs is that many floating point operations can be performed in parallel. The parallelization strategy we choose must allow use of many Processing Elements (PE). We parallelize over the set of dot products in the matrix multiply, assigning a minimum of one dot product to each PE. So the maximum usable number of PEs is the dimension of the matrix. Further scaling would require breaking dot products between processing elements. Parallel scaling can also be limited by the large amount of communication work required when there are many PEs. Our results show that parallel performance becomes inefficient before breaking dot products is necessary for further scaling (Chapter 3.4).

A different strategy is used by Zhuo and Prasanna's FPGA solution [17]. It sequentially iterates over dot products, parallelizing each one. The number of usable PEs is limited by the number of non-zero entries per row, which is typically between 8 to 200. Non-zeros per row is independent of the size of the matrix, so this approach sometimes cannot parallelize large problems by more than 8 PEs. Table 1.1 shows how various approaches choose to parallelize. Zhuo and Prasanna's design stores the matrix in off-chip memory which has the advantage that only one FPGA is required for large matrices. They do not require much parallelism, since they use one FPGA with throughput limited by off-chip bandwidth.

Problems with regular or periodic two- or three-dimensional structure often generate banded matrices. For banded matrices, a common parallelizing strategy is to assign each band to a processor. Then communication, which consists of entries of b and partial accumulations of entries of Ab, is systolic on a linear array of PEs. [18] adapts this strategy to general sparse matrices by assigning matrix entries to jagged bands. For this approach the time complexity is lower bounded by the dimension of the matrix since both b and Ab are streamed through the PE array. This means that, like parallelizing each dot product, potential parallelism is limited to the number of entries per row.

1.3 Communication

For many approaches, including ours, large communication work between processing elements is the main scaling limiter. We perform offline data placement to minimize communication by exploiting locality in the sparse matrix. The interconnect for our design is a bidirectional ring which allows inexpensive local communication between PEs. We also evaluate the time overhead of mesh interconnect which provides lower message latency and higher throughput than the bidirectional ring.

Section 3.1 determines the value of data placement on the bidirectional ring in terms of asymptotic scaling. Random data placement limits the number of efficiently utilizable FPGAs to a constant. Clustering data then assigning one cluster to one PE allows the number of efficiently utilizable FPGAs to increase with matrix size. Intelligently placing clusters on PEs further increases the asymptotic number of efficiently utilizable FPGAs. The asymptotic analysis models connection locality in sparse matrices with a constant Rent parameter.

Single FPGA performance is halved at 16 FPGAs, primarily due to communication overhead on the bidirectional ring. Although data placement can decrease the total work required to communicate, it usually cannot decrease the maximum message latency. Since maximum message latency is proportional to the number of PEs, it dominates communication time over interconnect throughput for more FPGAs than about 30, which is where communication time becomes significant compared to computation time. Section 3.5 evaluates the communication overhead when two-dimensional mesh interconnect is used instead of the bidirectional ring. The mesh decreases the maximum message latency and increases interconnect throughput. When using the mesh, scaling is most often limited by unbalanced computation load. After 96 FPGAs median performance drops below half single FPGA performance. When mesh overhead does limit scaling, which may begin at 20 FPGAs, it is due to limited interconnect throughput rather than message latency.

1.4 FPGA Architecture Exploration

Modern FPGAs are typically used to solve computations for large problems using off-chip memory. Our approach is to utilize high memory throughput by using on-chip memory. For large matrices we use many parallel VirtexII-6000s with enough combined memory capacity to store the matrix. This has the disadvantage that increased parallelism decreases Mflops/FPGA. Also, matrices with more than about 2,000,000 non-zeros cannot fit on any number of FPGAs due to large communication memories. However, as the gap between off chip memory bandwidth and on-chip compute grows, the performance advantage of using on-chip memories will continue to grow. In order to utilize on-chip compute, we consider increasing on-chip memory capacity by devoting a significant fraction of chip area to memory. The types of memory we consider are increased capacity BlockRAMs and embedded DRAMs. Since our application uses double precision floating point arithmetic, we also evaluate the effect of devoting area to custom FPUs. Chapter 4 evaluates these modifications to make hardware more suitable to our application, which is representative of numerical applications with large working memory.

Increasing BlockRAM area while sacrificing reconfigurable logic area allows significant performance improvements for large matrices by up to 5 times, while decreasing performance slightly for small matrices (Section 4.2). Use of embedded DRAM blocks rather than increased capacity BlockRAMs allows only marginal performance increase for benchmark matrices, while the large DRAM blocks sacrifice application flexibility (Section 4.4). Section 4.5 shows that when custom FPUs are used with DRAM memory, single FPGA performance for large matrices is about 5 Gflops. This is about 3 times the performance of the performance of a single VirtexII-6000, and many times the performance for large matrices that require many FPGAs. For large matrices, SRAM memories with custom FPUs forces the number of PEs to be too large for efficient performance. For our benchmark matrices, greater than half of the 14 Gflops peak performance is maintained for up to 35 PEs/FPGA (Subsection 4.7).

1.5 Outline of Paper

Chapter 2 describes the transform from the basic SMVM sequential algorithm to our parallel VirtexII-6000 implementation. It describes the off-line matrix mapping that is performed in software. Chapter 3 reports the single FPGA and parallel FPGA performance of our design which it compares to microprocessor performance. It analyzes the effects of design decisions such as data placement for locality and bidirectional interconnect. Chapter 4 explores FPGAs with reconfigurable logic area sacrificed for hardware resources that are suitable to our design.

Chapter 2

Design

This chapter describes our implementation of Iterative SMVM for many VirtexII-6000s connected in a linear topology. We start with the simple CSR sparse matrix representation and the sequential CSR algorithm. We then parallelize across dot products to run on a parameterized number of processing elements. In a processing element, pipelined arithmetic units stream data between memories. Linear interconnect then communicates data between processing elements, six of which are on each FPGA.

2.1 Sparse Matrix Representation and Sequential Algorithm

One of the simplest and most efficient sparse matrix representations is Compressed Sparse Row (CSR), as shown in Figure 2.2. Using CSR, the matrix A is represented as three arrays: `row_start`, `matrix_value`, `column_index`. A is square with dimension $n \times n$ and has m non-zero entries.

- `matrix_value` of length m stores the non-zero values in row major order (non-zeros in row 0 ordered by their column index, then non-zeros in row 1 ordered by their column index, ...).
- `column_index` of length m stores the column indices of non-zeros also in row major order.
- `row_start` of length $n+1$ stores each row's starting index into `matrix_value` and `column_index`.

If `j<(row_start[i+1]-row_start[i])` then

```
A[i][column_index[row_start[i]+j]]=matrix_value[row_start[i]+j]
```

```
CSR(row_start, matrix_value, column_index, b, x)
  for (int row=0;row<n;row++)
   accum=0
   for (int i=row_start[row]; i<row_start[row+1]; i++)
     accum=accum+matrix_value[i]*b[column_index[i]]
   x[row]=accum
```

Figure 2.1: Compressed Sparse Row SMVM Algorithm

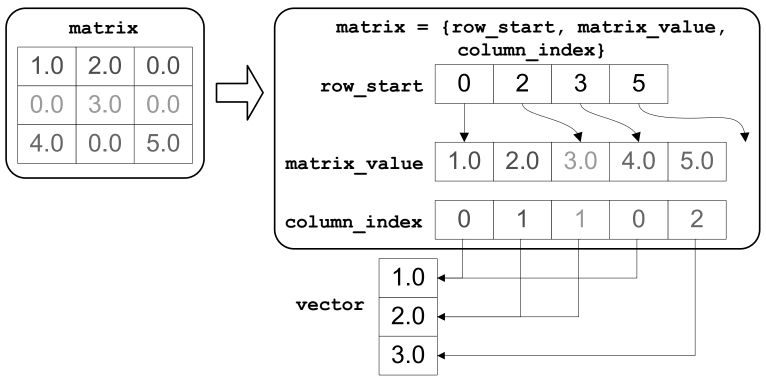


Figure 2.2: Compressed Sparse Row Representation of Example Matrix

The sequential CSR algorithm computes $x = Ab$ by performing a dot product on each row. If A_i is the ith row, then $x_i = A_i b$, where the dot product between vectors a and b is defined as $a^t b = \sum_i a_i b_i$. It performs dot products from top to bottom (See Figure 2.1).

2.2 Architecture

Our implementation parallelizes CSR SMVM by partitioning the set of n dots products across multiple Processing Elements (PEs). The entire computation is the set of dot products between the vector and the matrix rows. We assign the dot products, $A_i b$, to PEs, so they can compute in parallel (Section 2.4). During the compute stage, each dot product results in a vector entry, x_i. Since we are iterating matrix multiply, we must send the resulting entries to the PEs that will use them for the next iteration, setting $b_i^{(t+1)} := x_i^{(t)}$. This is performed by a communication stage (Section 2.2.2).

2.2.1 Compute

During the compute stage each PE accumulates dot products on its rows of A and the vector b to produce the vector x. The PE that is assigned dot product $A_i b$ stores the row A_i in its `compute_mem` and puts the resulting dot product $x_i = A_i b$ into its `dest_mem`. Each PE stores the entries of b that are used by its dot products in its `source_mem`. Each element of b may be used by multiple PEs so the local `source_mem`s redundantly store entries of b.

The PE datapath (Figure 2.3) performs its accumulation with a floating-point multiply-accumulate (MAC). Values from `source_mem` and `compute_mem` stream through the MAC and into `dest_mem`. `compute_mem` also provides indices into `source_mem` and control to initialize accumulations and store accumulations into `dest_mem`. `compute_mem` increments through addresses to provide the same sequence of instructions on each compute stage execution. `compute_mem` acts as a queue which is full at the beginning of each iteration and is popped on each cycle. For the compute stage, we can think of `dest_mem` as a queue which is initialized to empty and is pushed each time a new entry of x is ready. Each `compute_mem` word is an instruction: {`end_dot`, `matrix_value`, `source_address`}.

- `matrix_value` is the entry value.
- `source_address` is the address into `source_mem` which is multiplied by `matrix_value`. Relating to CSR, `source_address` takes the place of `column_index`. Instead of multiplying:

  `matrix_value[i]*b[column_index[i]]`

  we multiply:

  `matrix_value[i]*source_mem[source_address[i]]`
- `end_dot` instructs an accumulation to end by pushing its output into `dest_mem` and reinitializing the MAC to zero.

Figure 2.4 shows pseudocode for the version of the CSR algorithm performed by each PE.

To exploit the full computational throughput of the FPGAs, we want to pipeline the dot-product accumulation as heavily as possible, maximizing clock frequency. Since one accumulation input depends on the result of previous MAC operations, the latency of the addition stage prevents us from pipelining a **single** dot product at the full throughput which the FPGA can offer. However, we are computing multiple dot products on each PE, and these dot products may be computed in parallel. Consequently, we can interleave the independent dot products in C-slow fashion [9] on a single floating-point MAC pipeline. The adder latency, L_{add}, becomes the interleave factor, C. Consequently, the data streams into the MAC must be interleaved in `compute_mem` consistently with the adder latency as shown in Figures 2.3 and 2.5. The following recursion computes the accumulation of row i; $accum_t$ is computed on cycle t:

$$accum_{(t_i+L_{add}\times j)} = accum_{(t_i+L_{add}\times(j-1))} + (\texttt{matrix\_value[row\_start[i]+j]} \times \texttt{b[column\_index[row\_start[i]+j]]})$$

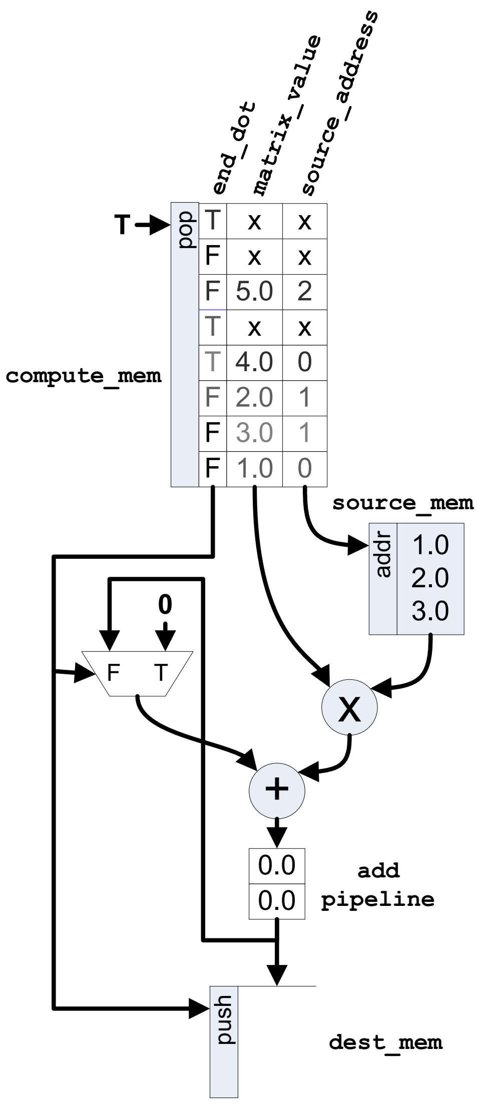
and

$$accum_{t_i} = (\texttt{matrix\_value[row\_start[i]]} \times \texttt{b[column\_index[row\_start[i]]]})$$

That is, the accumulation of row i begins on cycle t_i and is interleaved with L_{add} other accumulations. Table 2.5 shows the succeeding memory states. We can think of the accumulations as occurring on L_{add} processors in parallel; we call each "processor" a MAC slot. We parameterize logic generation and memory configuration (Section 2.3) around L_{mult} and L_{add}.

2.2.2 Communicate

The communication stage sets $b_i^{(t+1)} := x_i^{(t)}$ by copying the contents of each `dest_mem` to `source_mems` on different PEs. The interconnect topology is a bidirectional ring as shown in Figure 2.6. One ring sends messages to the right and the other sends messages left. Matrix locality and good partitioning imply locality in inter-PE communication. Two ring directions allow local communications between PEs to be short.



$L_{add} = 2$ in this example. The memory contents are the initial values to multiply the example matrix and vector in Figure 2.2. X values are don't cares which result when a matrix does not exactly fill a multiple of L_{add} MAC slots and at the end of the computation when we need to flush the adder pipeline. `end_dot` values are placed L_{add} cycles after the accumulates they end.

Figure 2.3: PE Compute Datapath

The communication pattern is fixed for each multiply, so it can be statically scheduled. Switches are controlled by their adjacent PEs (Figure 2.7). After a message is sent on a ring its receiving PEs copy it off. Once the message has been received by all its destination PEs, it may be overwritten by another message. A vector element with destinations both to the right and to the left generates one message to send right and one message to send left. One advantage of static scheduling is that one message may fan out to multiple PEs without a dynamically sized header. The bus data width is the same as the compute datapath, so each message occupies one ring register at a time.

Like the compute stage, the communicate stage has an instruction memory, `communicate_mem`,

```
accum=0
row_idx=0
 for i  in [0,instr_len)
  prod=source_mem[source_address[i]]
      * matrix_value[i]
   if (end_dot[i])
    destination[row_idx]=accum
    accum=prod
    row_idx=row_idx+1
   else
    accum=accum+prod
```

Figure 2.4: PE Compute Code ($L_{add} = 1$)

compute_mem:{end_dot;matrix_value;source_address}

T; x; x	F; x; x	F; 5.0; 2	T; x; x	T; 4.0; 0	F; 2.0; 1	F; 3.0; 1	F; 1.0; 0
	T; x; x	F; x; x	F; 5.0; 2	T; x; x	T; 4.0; 0	F; 2.0; 1	F; 3.0; 1
		T; x; x	F; x; x	F; 5.0; 2	T; x; x	T; 4.0; 0	F; 2.0; 1
			T; x; x	F; x; x	F; 5.0; 2	T; x; x	T; 4.0; 0
				T; x; x	F; x; x	F; 5.0; 2	T; x; x
					T; x; x	F; x; x	F; 5.0; 2
						T; x; x	F; x; x
							T; x; x

add pipeline

0.0	0.0
1.0	0.0
6.0	1.0
5.0	6.0
4.0	5.0
x	4.0
19.0	x
x	19.0
x	x

dest_mem

6.0		
5.0	6.0	
5.0	6.0	
5.0	6.0	
19.0	5.0	6.0

Figure 2.5: Trace of Compute Memory Values Starting at the Initial Values in Figure 2.3

which it cycles through once per communicate stage execution. `communicate_mem` contains instructions of the form {`dest_address`, `left_recv`, `right_recv`, `left_send`, `right_send`}. If the left or right receive flag is valid, a message is received from the left-ring or right-ring respectively. `source_mem` acts as a queue and pushes received messages. If the left or right send flag is valid, a message is sent on the left-ring or right-ring respectively. When sending, `dest_address` addresses the `dest_mem` word to send. Figure 2.7 shows the communicate logic for one PE along with its left-ring and right-ring switches.

The pipeline depth of interconnect between PEs is parameterized so it does not constrain the maximum operating frequency. When message latency rather than ring throughput dominates the number of cycles required for communication, the communication time is proportional to this pipeline depth. On the other hand, when throughput dominates this pipeline depth makes little difference.

2.2.3 Controller Element

Taking the place of one PE is the Controller Element (CE) which is for performing the high level control and memory loading and unloading. The CE has a state machine where each state corresponds to a stage of PE behavior. To start each stage it issues instructions in the on the right-ring direction. The stages are the compute stage, the communicate stage, and memory load and unload

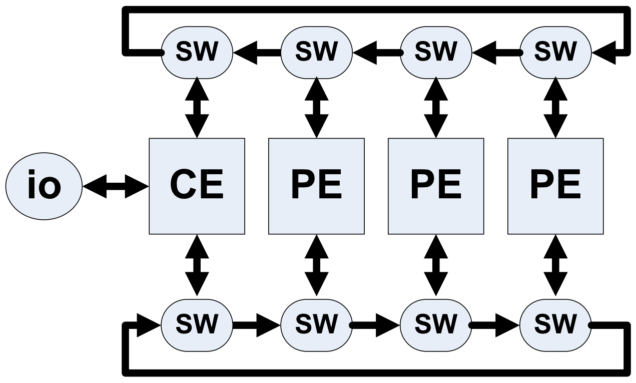


Figure 2.6: Bidirectional Ring

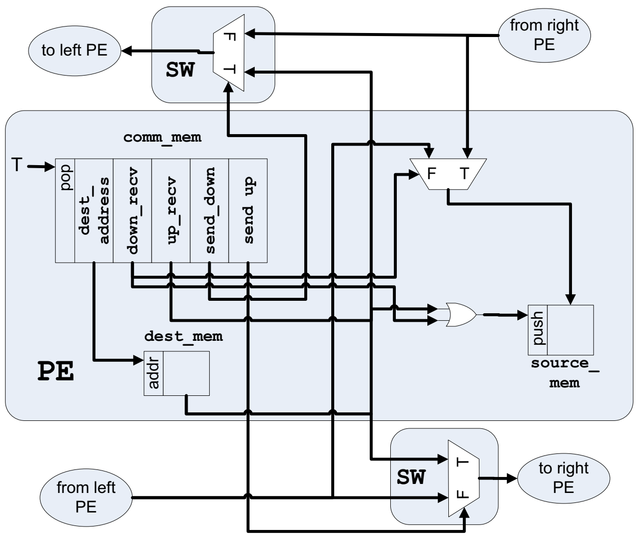


Figure 2.7: PE Datapath used for Communication

stages. There is one memory load stage for each of the four types of PE memories. It loads data from off-chip to the right-ring and unloads from the right-ring. The CE requires little area compared to a PE since it is a simple state machine and has no floating point arithmetic in its datapath.

Stages are divided into two behavior types: sequential for memory load and unload, and parallel for compute and communicate. During a sequential stage one PE is active at a time starting with the PE to the right of the CE. An active PE activates its up neighbor when it finishes. Sequential stages are useful for memory loading and unloading, where the off-chip data stream can only load to or unload from one PE memory at a time. During a parallel stage the CE sends a start signal up the ring which activates PEs as it passes them. Done signals are linearly AND reduced from the PEs to the CE: Once a PE is finished and it has received a done signal from the previous PE it sends done to the next PE. Five extra bits on the right-ring distribute the reset signal and control

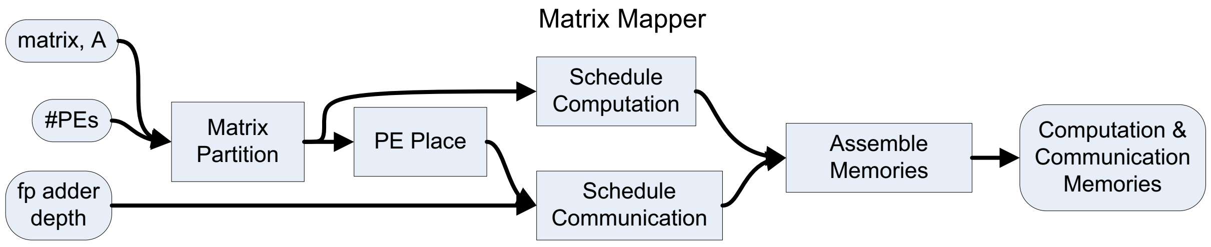


Figure 2.8: Matrix Map Stages

when PEs start and finish. For stage start instructions the other 64 bits of the right-ring specify which stage to start: Compute, communicate or which memory to load or unload is specified which sets the PE data path until the next instruction arrives.

2.3 Design Parameterization

To make this solution general and scalable, we parameterize the logic generation, assembly, and tools. This allows us to quickly assimilate better floating-point cores, new technologies which may have different levels of pipelining, and various FPGA capacities. Key parameters include:

- L_{add} – adder pipeline depth
- L_{mult} – multiplier pipeline depth
- $L_{ringstage}$ – ring stage pipeline depth; this can be tuned so that interconnect latency does not limit the clock cycle and to tolerate pipelining between chips.
- N_{PEs} – number of processing elements.
- N_{FPGAs} – number of FPGAs.
- W – datapath width; this allows support for single-, double-, and custom-precision floating-point units.
- $M_{depth}[mem]$ – memory depth of memory mem per PE; $mem \in \{$`compute_mem`, `communicate_mem`, `source_mem`, `dest_mem`$\}$. This is tuned along with the parallelism. Highly parallel designs have shallow memories, while more sequential designs require deeper memories per PE (See Table 2.1).

Logic is generated using a flexible generator built in JHDL[3].

2.4 Matrix Mapping

To map a matrix to this architecture, we must schedule the communication and computation of the input matrix and produce memory configurations to load onto logic. The scheduling will depend on the logic parameters (L_{add}, L_{mult}, $L_{ringstage}$) and the total number of processors, N_{PEs}. Figure 2.8 shows the operations performed for mapping.

Matrix partitioning assigns dot products, or equivalently, vector entries, to PEs. A good partitioner will load balance to minimize computation latency while minimizing the inter-PE communication. `compute_mem` size will set a limit to the volume of work assigned to any single PE. To minimize communication, dot products should be placed to minimize the number of dot products in other PEs that use their result, effectively minimizing the number of messages that need to be sent and the size of the `source_mems`'. We use UMpack's multi-level partitioner, `UCLA_MLPart4.21.1`, on a Linux platform [4].

Partitions are then placed on PEs to minimize the sum of message distances. Graphs with locality tend to have locality on their partition level as well, so placement of partitions on PEs is important. UMpack's partitioner computes binary partitions, so we apply it recursively to compute an arbitrary number of partitions. The resulting binary tree of partitions is then flattened for a linear placement. Subsection 3.1.4 shows that this placement of partitions asymptotically minimizes the sum of message distances.

After placement, the computation scheduler load balances dot products assigned to a PE across the L_{add} MAC slots. Since `compute_mem` depths are bounded by compute stage latency, a poor quality schedule causes both high computation time and requires large memories. The simple strategy used is to order each accumulate by its length. Accumulates are then greedily scheduled from largest to smallest. The schedules resulting from this heuristic are never longer than the optimal schedule plus the length of the longest dot product [6, 7].

After placement, we also need to schedule communications. The quality of this schedule affects the communicate stage latency which is limited by `communicate_mem` size. `dest_mem` words are sent to `source_mem`s. Each word that is used outside its PE must be sent to a set of sink PEs. Since one message may fanout to multiple PEs and PEs are placed for locality, typically each word is sent by one short left message and one short right message. For a given word, the set of PEs that receive it from the left message and the set that receive it from the right are chosen to minimize the sum of message latencies. Our message scheduling algorithm is described by Figure 2.9. It schedules messages greedily with priority to the longest.

The message scheduler shown in Figure 2.9 uses the following predicates and operations:

- `unscheduled()` is true iff there exist unscheduled messages.
- `unsent(PE)` is the set of unscheduled messages to be sent from `PE`, ordered from longest to shortest.
- `ring_free(m,c)` is true iff message `m`'s ring is unused on cycle `c`.
- `destinations_free(m,c)` is true iff when message `m` is sent on cycle `c` all its destination PEs don't yet input on `m`'s arriving cycle.
- `send(PE,m,c)` schedules message `m` to be sent on cycle `c`.

```
cycle=0
 while unscheduled()
   for each PE  in PEs
     for each message  in unsent(PE)
       if (ring_free(message,cycle)  and
          destinations_free(message,cycle))
        send(PE,message,cycle)
  cycle=cycle+1
```

Figure 2.9: Bidirectional Ring Message Scheduler.

Memory	Width	Depth	BlockRAMs
`compute_mem`	75 (78)	3584 (3584)	15 (15)
`communicate_mem`	14 (14)	5120 (9216)	5 (9)
`dest_mem`	64 (64)	512 (512)	2 (2)
`source_mem`	64 (64)	512 (2560)	2 (10)
Total			24 (36)

Table 2.1: Per PE Memory Shapes for 6 PEs (4 PEs) per FPGA

2.5 Concrete Design

The design is implemented on the VirtexII-6000-4. Since the computation and communication schedules are static given the matrix, we know how many cycles a matrix multiply takes.

Since standard benchmarks are in terms of double precision Mflops, we use double precision arithmetic units. We modified our FPUs from parameterized precision VHDL cores from Northeastern University by increasing pipeline depths [2]. The resulting pipeline depths for the adder and multiplier are 13 and 26 respectively. The overall clock frequency is 140MHz, which is limited by the multiplier frequency.

To operate at 140MHz the ring pipeline depth per PE, $L_{ringstage}$, is 5. So $L_{ring} = 5N_{PEs}$.

The floor-planned adder and multiplier occupy 790 and 3276 slices respectively. The total number of slices for the VirtexII-6000 is 33792, which allows a maximum of 8 PEs/FPGA. The peak performance is then $2 \times 8 \times 140\text{MHz} = 2240$ Mflops/FPGA. Including other logic (*e.g.* control logic, addressing, interconnect) and 1 limits us to 6 PEs/FPGA arranged as two columns of three PEs. This gives us a maximum performance of 3/4 peak. The CE takes the place of one PE on one FPGA. Due to deep pipelining, logic, including the FPUs, uses more registers than LUTs. Since VirtexIIs have one register per LUT, registers are the critical resource.

Memory depths, $M_{depth}[mem]$, and N_{pes} must be large enough to fit the input matrix. Keeping N_{FPGAs} constant, and decreasing N_{pes} while increasing $M_{depth}[mem]$ tends to allow larger matrices for two reasons: Data can be more balanced in larger memories since the BlockRAM depth is finer grain relative to $M_{depth}[mem]$. Smaller N_{pes} decreases $L_{communicate}$ which decreases the upper bound on total `communicate_mem` words: $N_{pes} \times L_{communicate}$. We map to 4 PEs/FPGA as well as

6 PEs/FGPA to allow larger matrices to fit. Table 2.1 shows memory sizes per PE which fit 6 PEs (4 PEs) per FPGA.

All logic except for the FPUs was generated using JHDL 0.3.34. FPUs were synthesized with Synplicity Synplify Pro 7.5. Logic was mapped, placed, and routed with Xilinx ISE 6.1.

Chapter 3

Evaluation

This chapter evaluates the performance of and sources of inefficiency in our implementation on the 35 benchmark matrices in Table 3.2 taken from the Matrix Market Suite [1]. Matrices range from 17,000 non-zeros to 2,000,000 non-zeros, with dimension from 300 to 100,000. Parallel SMVM algorithms cannot maintain constant efficiency or constant Mflops/processor as the number of processors is increased beyond a small number. This chapter reports the efficiency of scaling our design in Section 3.1 and Section 3.4. We evaluate performance in terms of Mflops/FPGA for one FPGA and many FPGAs, which we compare to single and parallel microprocessor performances in Section 3.4.

Section 3.1 asymptotically evaluates our design's maximum efficient scaling. It uses a constant Rent parameter to model matrix locality. The section shows asymptotic improvement in scaling when matrix locality is used for data placement. Section 3.2 defines metrics we use to analyze types of inefficiency in our approach and to find their sources. Section 3.3 discusses memory size impact on efficiency, required number of FPGAs, and maximum matrix size. Section 3.4 reports performances and efficiencies for the benchmark matrices. Section 3.5 shows what happens in terms of $L_{communicate}$ when the interconnect topology is changed to a two-dimensional mesh of PEs. Section 3.7 compares SMVM performance to Conjugate Gradient performance.

3.1 Communication Models

In this section we study how communication bandwidth constraints affect asymptotic scalability. We evaluate how much matrix locality increases scalability by using it to place data on PEs. Our notion of scalability is the number of PEs usable with efficiency greater than a constant. We analyze how the number of PEs, N_{PEs}, scales with matrix dimension, n, for three different partitioning and PE placement types:

- Random assignment of dot products to PEs.
- Partitioning dot products for locality into PEs, then placing PEs in a random order.
- Partitioning for locality, then placing PEs for locality.

For matrices with locality, we show that each refinement improves asymptotic scaling of N_{PEs} in terms of n.

We use $L_{ideal\_compute}$ to model the compute stage latency, and $L_{communicate}$ to model the communicate stage latency. We model of the fraction of total cycles spent in the compute stage as:

$$E_{C_{ideal}} = \frac{L_{ideal\_compute}}{L_{ideal\_compute} + L_{communicate}}$$

Henceforth we use $E_{C_{ideal}} \geq 1/2$ as our target constant, which gives us $L_{ideal\_compute} \geq L_{communicate}$.

Recall the matrix dimension is n, and the number of non-zeros is m. $k = m/n$ is the average non-zeros per row. $L_{ideal\_compute} = kn/N_{PEs}$ so for $E \geq 1/2$ in the following analyses we will use:

$$L_{communicate} \leq kn/N_{PEs} \tag{3.1}$$

Two lower bounds on the latency of the communicate stage are the maximum message latency, L_{ring}, and the cycles required if interconnect is fully utilized, $L_{throughput}$:

$$L_{communicate} \geq \max(L_{ring}, L_{throughput}) \tag{3.2}$$

If the maximum message latency constrains communication, then we say it is latency constrained, otherwise it is throughput constrained. Since maximum message latency is linear in the number of PEs, N_{PEs}, we model:

$$L_{ring} = L_{ringstage} \times N_{PEs} \in \Theta\left(N_{PEs}\right) \tag{3.3}$$

Combining Equations 3.1, 3.2 and 3.3 we get the upper bound on N_{PEs} due to ring latency:

$$N_{PEs} \in O(n^{1/2}) \tag{3.4}$$

Interconnect is fully utilized if each switch routes a message on each cycle:

$$W_{comm} = \sum_{msg \in messages} L_{msg}$$

L_{msg} is the distance each message must travel. W_{comm} is the useful work performed in communication. We say a switch performs one unit of work on each cycle it routes a message. Since there are $2N_{PEs}$ switches:

$$L_{throughput} = W_{comm}/(2N_{PEs})$$

Hence:

$$L_{communicate} \geq W_{comm}/(2N_{PEs}) \tag{3.5}$$

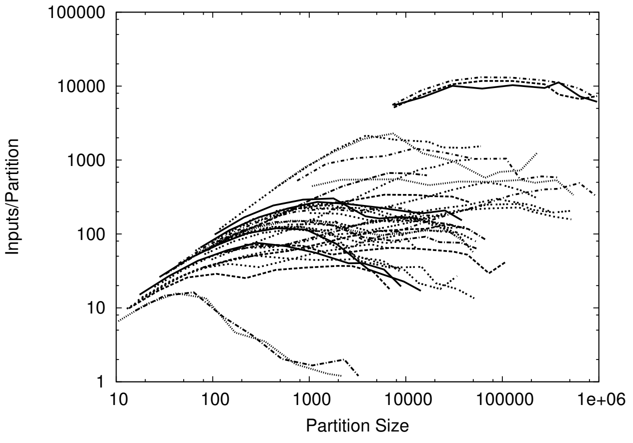


Figure 3.1: I/O Scaling for Benchmark Matrices

In this section's model of perfectly balanced communication, it is throughput constrained if and only if:

$$W_{comm}/(2N_{PEs}) \geq L_{ringstage} \times N_{PEs} \tag{3.6}$$

3.1.1 Random Partitioning

First we find the scaling effect of a partitioning that load balances dot products with no regard to matrix locality. Most vector entries are used by multiple dot products, which are distributed in random PEs. So most entries are sent as either one or two messages which are received by all destination PEs. The length of the ring is $L_{ringstage} \times N_{PEs}$, so the work per vector entry is proportional to N_{PEs}. Since assignment was random, communication is load balanced on switches. Hence $W_{compute} = n \times N_{PEs}$. From Inequality 3.6, communication is throughput constrained:

$$L_{communicate} = W_{comm}/(2N_{PEs}) \propto n$$

Using Eq. 3.1, we find:

$$N_{PEs} \in O(1)$$

This means we can only use N_{PEs} constant in n or communication will dominate. Therefore increasing the matrix dimension while keeping non-zeros per row fixed does not allow us to scale to more processors.

3.1.2 Matrix Model

In order to analyze the effect of good partitioning, we need a model of matrix locality. We first represent the matrix communication structure as a graph. Each dot product is a node. It fans out to each dot product that uses its result. Consequently each node has an input for each non-zero matrix entry. We use the common Rent Parameter model, where the graph is fitted to the two parameters c, p [8]. The Rent Parameter is defined for a graph when there is a power law relating the size of each local cluster with the IO of the cluster. If the number of nodes per cluster is r, then the number of inputs to each cluster is:

$$inputs(r) = c(r)^p \tag{3.7}$$

A graph with $p = 1$ has little locality if any: a constant fraction of nodes in a partition output to another partition. A 3D problem has $p = 2/3$, a 2D problem has $p = 1/2$, and a 1D problem has $p = 0$. Circuit graphs often have $p = 2/3$.

Partitioning well with different size partitions can be used to fit to Equation 3.7. Figure 3.1 shows the average number of inputs per partition for our benchmark matrices. This uses the multilevel partitioner we used for matrix mapping (Section 2.4). Figure 3.1 plots Eq. 3.7 on a log-log scale. Relating x and y gives $y = \log(c) + px$. So p for a matrix is the slope of $inputs(r)$ vs r on the log-log scale plot. Many matrices have a flat slope for two orders of magnitude and hence a well defined p. Others have negative curvature, which means larger partitions have smaller p. The average p for most matrices ranges from 0 to 0.6 (See Table 3.2). In either case, when $p < 1$ there is locality to be exploited by a partitioner and placer.

3.1.3 Partitioning for Locality and Placing Randomly

Next we find the scaling effect of a partitioning that load balances and minimizes communication between partitions. Each partition is then assigned to a random PE. Communication will be load balanced since placement is random. Here, message sends per PE is $\Theta((n/N_{PEs})^p)$. Work per message is still N_{PEs}, so work per PE is $\Theta(N_{PEs} \times (n/N_{PEs})^p)$. Since there are N_{PEs} PEs:

$$W_{comm} \in \Theta(N_{PEs}^2 \times (n/N_{PEs})^p) \tag{3.8}$$

From Inequality 3.6, communication is throughput constrained. From Inequalities 3.1 and 3.5:

$$kn/N_{PEs} \geq W_{comm}/(2N_{PEs}) \tag{3.9}$$

Next use Relations 3.8 and 3.9 to solve for N_{PEs}:

$$N_{PEs} \in O(n^{(1-p)/(2-p)})$$

For example, for $p = 2/3$, $N_{PEs} \in O(n^{1/4})$, and for $p = 1/2$, $N_{PEs} \in O(n^{1/3})$.

Section	partition	place	$N_{PEs}(n,p)$	$p=0$	$p=1/2$	$p=2/3$	$p=1$
3.1.1	random	random	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
3.1.3	good	random	$O(n^{(1-p)/(2-p)})$	$O(n^{1/2})$	$O(n^{1/3})$	$O(n^{1/4})$	$O(1)$
3.1.4	good	good	$O(\min(n^{1/2}, n^{1-p}))$	$O(n^{1/2})$	$O(n^{1/2})$	$O(n^{1/3})$	$O(1)$

Table 3.1: PE Scaling on Ring: Number of PEs which can be supported with bounded efficiency

3.1.4 Partitioning and Placement for Locality

Scaling can be further improved by placing partitions on PEs for locality. We construct a hierarchical, binary tree of partitions, where each pair of siblings partitions its parent. On level k, each partition is of size $n/2^k$. When placing we flatten the tree to a line so each pair of sibling partitions on each level are adjacent. Then all k level siblings can communicate in parallel. This load balances communication. Sends per k-level partition is $\Theta((n/2^k)^p)$. So $L_k \propto (n/2^k)^p$ is the time to communicate between two k-level siblings. $L_{throughput} = W_{comm}/(2N_{PEs})$ is the communication latency due to throughput. Communicating on each level separately, we get

$$\begin{aligned} L_{throughput} &\propto \sum_{k=0}^{\log(N_{PEs})} L_k && (3.10)\\ &\propto \sum_{k=0}^{\log(N_{PEs})} (n/2^k)^p && (3.11)\\ &= n^p \times \sum_{k=0}^{\log(N_{PEs})} (1/2^k)^p && (3.12)\\ &\propto n^p && (3.13)\end{aligned}$$

We get Eq. 3.13 from Eq. 3.12 using:

$$1 \leq \sum_{k=0}^{\log(N_{PEs})} (1/2^k)^p < \sum_{k=0}^{\log(N_{PEs})} 1/2^k < 2$$

Using Eq. 3.1 we get:

$$N_{PEs} \in O(n^{1-p}) \tag{3.14}$$

Considering both throughput and latency constraints, we combine Eq. 3.14 and 3.4 to get:

$$N_{PEs} \in O(\min(n^{1/2}, n^{1-p})) \tag{3.15}$$

3.1.5 Comparison

Table 3.1 compares the three types of partitioning and placement. It shows that scalability is constrained to $O(n^{1/2})$, due to ring latency. Scaling is always limited to $O(1)$ for random partitioning or when $p = 1$.

3.2 Efficiency

This section analyzes the sources of inefficiency which contribute to the actual performance relative to peak performance. Recall that for a given matrix, m is the number of non-zeros. $L_{ideal\_compute} = m/N_{PEs}$ is the ideal latency where all logic is devoted to floating-point units that are fully utilized on each cycle. We decompose the actual latency of one iteration of SMVM into this ideal latency and an efficiency factor, E, for the parallel computation:

$$L = L_{ideal\_compute}/E \tag{3.16}$$

We decompose efficiency into four main components:

$$E = E_A \times E_B \times E_C \times E_L \tag{3.17}$$

Efficiencies are:

- E_A – MAC slot utilization
- E_B – Partition balance efficiency
- E_C – Communication efficiency
- E_L – Logic utilization

Figure 3.6 and Section 3.4 assess the magnitude of E_A, E_B, E_C and E_L.

During the computation stage, it may not be possible to schedule every PE so that it issues a MAC operation on every cycle. MAC slot utilization efficiency, E_A, measures the extent to which dot products assigned to a PE utilize its L_{add} MAC slots. If there are fewer dot products assigned to the PE than MAC slots, then parallelism due to pipelining cannot be fully exploited. Also, due to non-uniform dot product lengths, slots cannot be fed near the end of the computation. E_A is the number of cycles which use a MAC slot on the PE with maximum compute stage latency divided by the compute stage latency. E_A is correlated with m and negatively correlated with L_{add} and k. Section 3.4.3 reports how E_A scales with L_{add}.

The partitioner should try to balance the computation load between PEs. E_B measures how evenly computation work is assigned to PEs. We define E_B as the average number of non-zeros per PE divided by the non-zeros allocated to the PE with maximum latency. The partitioner trades off between minimizing E_B and minimizing communication. L_{max_row} is the size of the largest row. Since we assign rows atomically to PEs, if L_{max_row} is larger than the average non-zeros per PE then work cannot be evenly distributed.

Since computation and communication are separated into two, non-overlapped, stages, all cycles spent communicating contribute to overhead:

$$L = L_{compute} + L_{communicate} \tag{3.18}$$

We then define E_C:

$$E_C = L_{compute}/L \tag{3.19}$$

Section 3.1 modeled $E_A = E_B = E_L = 1$ and E_C as $E_{C_{ideal}}$. Recall from Equation 3.2 that our model of $L_{communicate}$ is based on the both the throughput and latency of the ring:

$$L_{communicate} \geq \max(L_{ring}, L_{throughput})$$

Since we must allocate some control logic, we cannot fill each FPGA with floating-point units. Further, the optimal number of PEs per FPGA may be less than maximum if large N_{PEs} causes communication to dominate computation. E_L measures the impact of these limitations. E_L is the ratio of the area of one double-precision multiply and one double-precision add to the actual PE area used. For our design, we choose between $E_L = 3/4$ and $E_L = 1/2$ (See Section 2.5).

3.3 Memory Sizes

Since our design uses BlockRAM memory only, larger matrices will require more FPGAs. However, as Table 3.1 and Section 3.1 show, there is a limit to the number of PEs, and hence FPGAs, we can effectively use before communication dominates computation (*i.e.* E_C begins to diminish with N_{PEs}). Each matrix has a feasible set of N_{PEs} and equivalently N_{FPGAs}. The feasible set is bounded below by memory capacity requirements. It is strictly bounded above by the point at which mapping to more PEs increases total latency, L. The feasible set is empty when a matrix is too large to be mapped to any number of FPGAs. This is the case because increasing N_{FPGAs} to gain more memory increases N_{PEs}, increasing $L_{communicate}$. Since $L_{communicate}$ is the length of `communicate_mem`, the total memory requirement also increases.

Combining the scaling of memory and communication requirements, we can derive a range of feasible matrix sizes for any constant efficiency, E. For constant E_C, we will spend, at most, a constant fraction of our cycles communicating; this means the communication memory (`communicate_mem`) will be at most linear in the size of the computation memory (`compute_mem`). Each `source` and `dest`

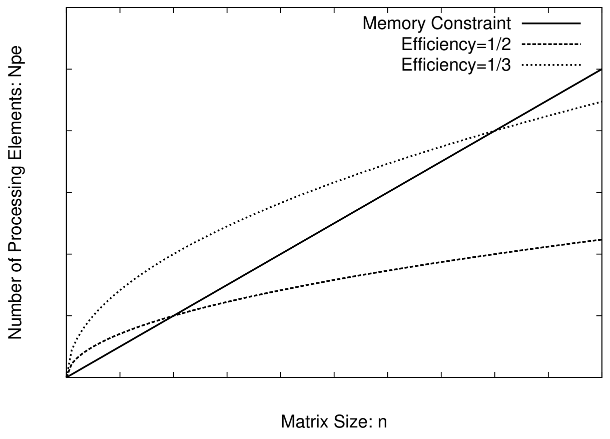


Figure 3.2: Bounded efficiency regions for n and N_{PEs} with constant memory per PE.

entry is used at least once so:

$$\begin{aligned} depth(\texttt{source\_mem}) &\leq depth(\texttt{compute\_mem}) \\ depth(\texttt{dest\_mem}) &\leq depth(\texttt{compute\_mem}) \end{aligned}$$

Together, this means the sum of the depth of all memory components (`compute_mem`, `communicate_mem`, `source_mem`, and `dest_mem`) is proportional to the compute memory depth ($depth(\texttt{compute\_mem})$). Therefore, to bound E_L to a constant, the memory per PE must be constant and hence $depth(\texttt{compute\_mem})$ must be a constant. Assuming E_A and E_B are constant, PE memory will be fully utilized. This means:

$$depth(\texttt{compute\_mem}) \propto m/N_{PEs} \propto n/N_{PEs}$$

This gives us the memory constraint:

$$N_{PEs} \in \Omega(n) \tag{3.20}$$

From Section 3.1 constant efficiency requires:

$$N_{PEs} \in O\left(\min\left(n^{1/2}, n^{1-p}\right)\right) \tag{3.21}$$

Together this means N_{PEs} must be within the region bounded below by the memory constraint (Eq. 3.20) and bounded above by the communication efficiency requirement (Eq. 3.21) as shown in Figure 3.2.

Figure 3.3 relates N_{PEs}, E, and m for benchmark matrices. It plots one point for the minimum possible N_{PEs} in each of the three efficiency regions, partitioned by $E = 1/2$, $1/4$, and $1/8$. The

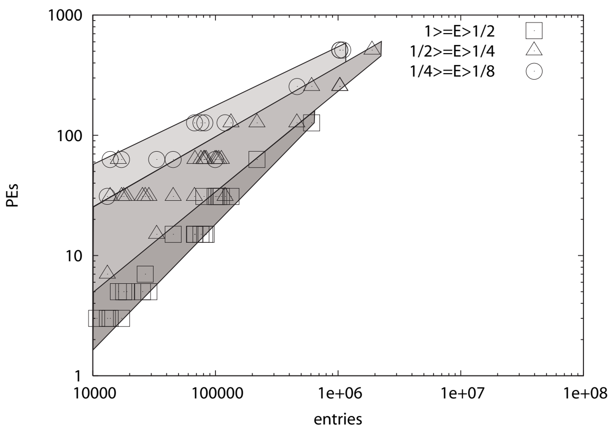


Figure 3.3: Three efficiency regions are shown. For each efficiency region, each matrix is plotted at the point at which it has the greatest efficiency.

graph shows the feasible set of N_{PEs} for each benchmark matrix. Each matrixes points are vertically aligned since the horizontal axis is m. The feasible region narrows as m increases until it ends at $m = 2,000,000$. The region that satisfies $E > 1/2$ narrows until it ends before $m = 1,000,000$. This shows that our use of on-chip memory only means we cannot efficiently support matrices with $m > 1,000,000$.

3.4 Results

SMVM performance is highly dependent on the matrix. For benchmarking, we used 35 matrices from the Matrix Market Suite [1] in Table 3.2. Although median performance tends to be close to maximum performance, performance of different matrices on the same number of processors varies by as much as a factor of four. Table 3.2 lists the matrices and their application areas. Matrix sizes range from 17,000 non-zeros to 2,000,000 non-zeros, with dimension from 300 to 100,000. We also chose matrices to cover a wide range of sizes and applications.

3.4.1 Single Processor Comparison

Table 3.3 compares the performance of our implementation on one VirtexII-6000 to the performance of various microprocessors. The single microprocessor information is performance reported for the SPARSITY sparse matrix pack [13]. Our performance for a single FPGA is the median of our benchmark matrices that fit on a single FPGA.

The Power 4 has the greatest peak performance of the microprocessors summarized here and was released the same year as the VirtexII-6000. The Itanium 2 performs relatively well, delimiting the highest net performance of the microprocessors considered, because it has a large cache and high

Matrix	Application	n	m	p
af23560	Aeronautics	23560	460598	0.2
bcsstk11	Finite Element	1473	17857	0.1
bcsstk18	Finite Element	11948	80519	0.3
bcsstk24	Finite Element	3562	81736	0.2
bcsstk25	Finite Element	15439	133840	0.1
bcsstk28	Finite Element	4410	111717	0.0
bcsstk30	Finite Element	28924	1036208	0.0
bcsstk31	Finite Element	35588	608502	0.1
bcsstk32	Finite Element	44609	1029655	0.1
bcsstm27	Finite Element	1224	28675	0.0
fidapm07	Finite Element	2065	45184	0.0
fidap009	Finite Element	3363	99397	0.0
fidap011	Finite Element	16614	1091362	0.0
fidap020	Finite Element	2203	67429	0.1
fidap035	Finite Element	19716	217972	0.0
fidapm07	Finite Element	2065	53533	0.3
fidapm37	Finite Element	9152	765944	0.0
dwt_2680	Finite Element	2680	25026	0.1
plat1919	Fluid Dynamics	1919	17159	0.2
lnsp3937	Fluid Dynamics	3937	25407	0.2
cavity10	Fluid Dynamics	2597	76171	0.2
conf6.0-0014x4-3000	Quantum Chromodynamics	3072	119808	0.4
gemat11	Power Grid	4929	33108	0.5
add20	Digital Logic	2395	17319	0.3
memplus	Digital Logic	17758	99147	0.6
mhd3200b	Magneto-hydrodynamics	3200	18316	0.0
mhd3200a	Magneto-hydrodynamics	3200	68026	0.0
mhd4800b	Magneto-hydrodynamics	4800	27520	0.0
mhd4800a	Magneto-hydrodynamics	4800	102252	0.0
qc324	Molecular	324	26730	0.1
qc2534	Molecular	2534	463360	0.2
s3dkt3m2	Finite Element	90449	1888336	0.2
s3rmt3m3	Finite Element	5357	106240	0.2
utm5940	Nuclear	5940	83842	0.2
rdb3200l	Chemistry	3200	18880	0.2

Table 3.2: Matrix Market Benchmark Matrices (p denotes average rent parameter)

memory bandwidth [13].

3.4.2 Parallel Processor Comparison

Table 3.4 shows that our implementation scales well to multiple processors. The microprocessor-based implementations may be affected by poor communication and partitioning as discussed in Section 3.1. Further, the multiple processor versions may pay operating systems overhead for communication. Single processor SMVM implementations tend to be more highly tuned to use available memory bandwidth than parallel implementations. We compare our iterative SMVM performance to other parallel machines' Conjugate Gradient(CG) performance; since CG is dominated by its SMVM kernel, and its other operations have higher performance than SMVM (See Section 3.7), the comparison favors the other machines. This shows a three times improvement over the best microprocessor based competitor, 16 ItaniumIIs.

Processor	Year	MHz	Peak Mflops/ Processor	SMVM Mflops/ Processor	fraction of peak	Ref.
Pentium 4	2000	1500	3000	425	1/7	[14]
Power 4	2001	1300	5200	805	1/6	[14]
Sun Ultra 3	2002	900	1800	108	1/16	[14]
Itanium	2001	800	3200	345	1/10	[14]
Itanium 2	2002	900	3600	1200	1/3	[14]
VirtexII-6000-4	2001	140	2240	1500	2/3	

Table 3.3: Single processor performances of SMVM.

Architecture	Processors	Year	MHz	Peak Mflops/ Proc	SMVM Mflops/ Proc	Fraction of Peak	Fraction of Single Proc Perf	Ref.
NEC SX-6	8 x NEC SX-6	2002	500	8000	*131	1/60		[11]
Altix	16 x Itanium II	2002	1500	6000	*263	1/23	1/4	[5]
Cray X1	16 x MSP	2002	800	12800	*170	1/75		[5]
SP4	16 x Power4	2001	1300	5200	*250	1/20	1/3	[5]
This Work	16 x VirtexII-6000s	2001	140	2240	750	1/3	1/2	

* denotes NAS CG performance

Table 3.4: Parallel processor performances of SMVM

We use Mflops/FPGA as our baseline performance metric for scaling. Figure 3.4 shows how performance scales with the number of VirtexIIs. Taking the median performances, 16 FPGAs deliver 1/3 peak. We get 1/7 peak at 128 FPGAs. The best parallel microprocessor architecture in Table 3.4 drops to 1/20 peak by 16 processors.

3.4.3 MAC Slot Scheduling

The MAC slot utilization component of E, E_A, is low if MAC slots are poorly utilized. Figure 3.5 shows how increasing L_{add} decreases E_A. At this point ($L_{add} = 13$), we are able to fill over 80% of our MAC slots, giving $E_A = 0.80$.

3.4.4 Analysis of Inefficiencies

The largest factor contributing to scaling inefficiency is the large ring interconnect latency, L_{ring}. We use Figures 3.6 and 3.7 to analyze sources of inefficiency.

Figure 3.6 shows E_C is the major component of diminishing efficiency as N_{PEs} increases. The two main latencies contributing to $L_{communicate}$ are L_{ring} and $L_{throughput}$. Figure 3.7 shows that at 1024 PEs L_{ring} dominates: $L_{ring} = (2/3)L_{communicate}$ and $L_{throughput} = (1/10)L_{communicate}$.

The second worst scaling efficiency is E_A. E_A decreases when there are too few dot products to be evenly distributed between MAC slots.

E_B is also significant. Large rows sometimes make it impossible to load balance. The heuristic

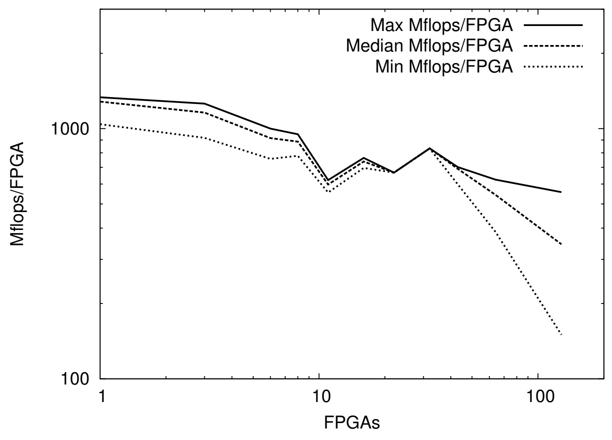


Figure 3.4: Mflops scaling of benchmark matrices in their feasible regions

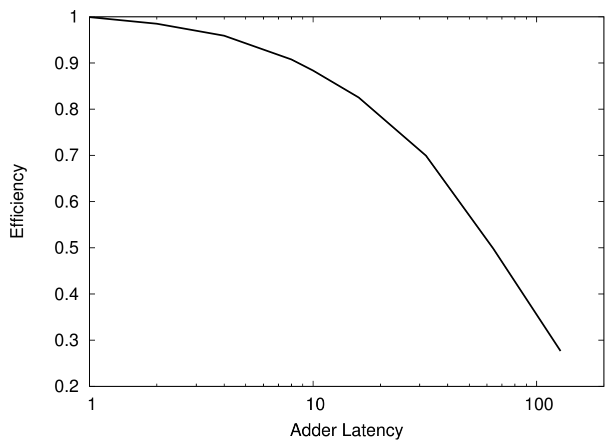


Figure 3.5: Median MAC slot efficiency, E_A, as a function of L_{add} for 16 FPGAs

partitioning algorithm could also contribute to low E_B, as well as the common trade-off between partition cut-size and partition load balance.

Constant E_L shows that we obtain our best performance using 6 PEs per FPGA, rather than 4, up to 95 FPGAs.

Scaling further requires decreasing communication overhead. The primary dominator of communication overhead is message latency rather than interconnect bandwidth. So message latency can be improved cheaply without improving throughput by switching to tree structured interconnect. Section 3.5 shows the latency improvement due to switching to mesh interconnect, which benefits scaling by both decreasing message latency and increasing interconnect bandwidth.

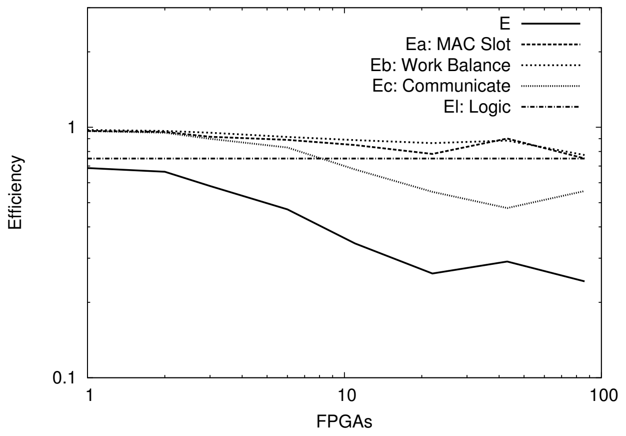


Figure 3.6: Scaling of median efficiencies for benchmark matrices in their feasible regions.

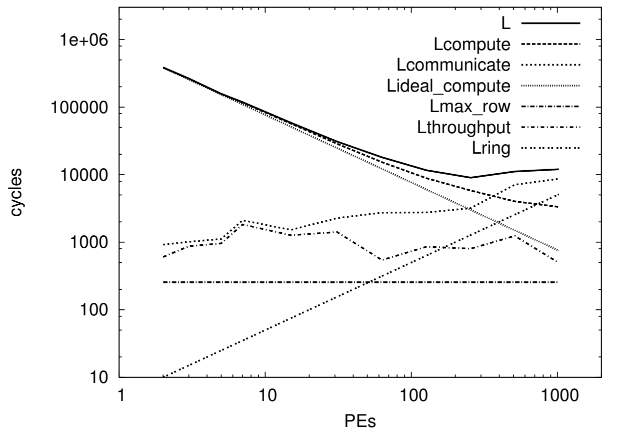


Figure 3.7: Matrix fidapm37 Cycle Breakdown: fidapm37 has $n = 9152$, $m = 765944$.

3.5 Mesh Latency

Communication latency can be decreased using lower latency, higher bandwidth interconnect. This increases scaling efficiency primarily by decreasing communication overhead and also by decreasing communication memory requirements, allowing fewer PEs. Decreased communication memory also allows matrices with larger m. From Subsection 3.4.4, we see the key scaling limitation in this architecture is, not surprisingly, communication latency on the ring. We can easily decrease the worst-case communication latency from the current $5N_{PEs}$ to $O(\sqrt{N_{PEs}})$ or even $O(\sqrt[3]{N_{PEs}})$ by moving to two- or three-dimensional interconnect structures.

This section evaluates how well two dimensional mesh interconnect improves $L_{communicate}$. Maximum message latency for a 2D mesh is $5\sqrt{N_{PEs}}$, compared to $5N_{PEs}$ for the bidirectional ring. Bisection bandwidth is also improved to $2\sqrt{N_{PEs}}$ from 4, where bisection bandwidth is the number

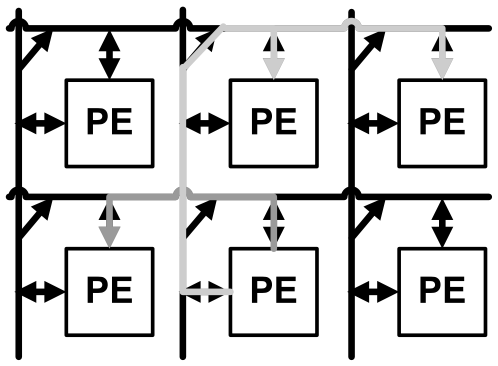


Figure 3.8: Topology of the two dimensional mesh. The two messages shown in gray transmit a vector entry which is used by three PEs other than its source.

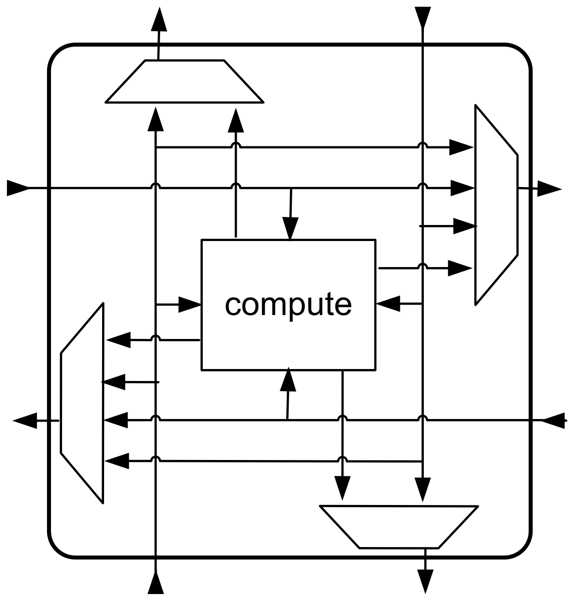


Figure 3.9: Switches for routing. Context memory for switches is not shown.

of lines between two halves of the ring or mesh. Figure 3.8 shows the mesh topology, and Figure 3.9 shows the mesh logic.

Our approach to mesh routing is similar to our approach to bidirectional ring routing:

- Static schedule given matrix
- Each PE has $L_{communicate}$ depth `communicate_mem` for switch instructions.
- Each communicated vector entry is sent as multiple messages, each of which are received by at least one PE.
- Algorithm schedules messages greedily with priority to the longest.

For each vector entry, for each mesh row, one message may be sent in the positive horizontal direction and one in the negative direction, as shown in Figure 3.8. The topology is square, not toroidal like

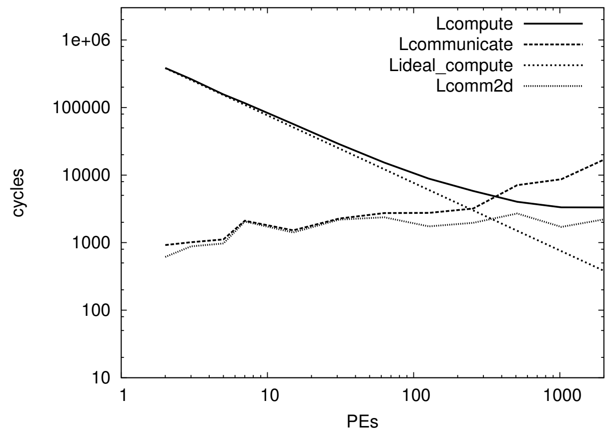


Figure 3.10: 2D mesh vs. bidirectional ring communication latency for the finite element matrix fidapm37. $n = 9152$ and $m = 765944$.

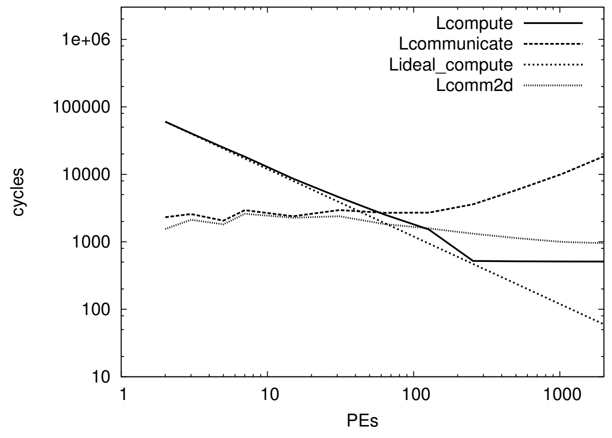


Figure 3.11: 2D mesh vs. bidirectional ring communication latency for the QCD matrix conf6.0-00l4x4-3000. $n = 3072$ and $m = 119808$.

the bidirectional ring.

Figures 3.10 and 3.11 compare mesh scaling with bidirectional ring scaling for two typical example matrices. L_{comm2d} is the communication latency for mesh interconnect. Figure 3.10 shows that $L_{comm2d} < L_{compute}$ for the finite element matrix. So for this case, mesh interconnect is adequate for any number of PEs. On the other hand, for the smaller QCD matrix, Figure 3.11 shows that if $N_{PEs} > 128$ then $L_{comm2d} > L_{compute}$. When $N_{PEs} = 256$ or $N_{FPGAs} = 43$ communication overhead causes a performance decrease by 3/4.

Since mesh latency can still limit scaling we use Figures 3.12 and 3.13 to understand which factors dominate L_{comm2d}. Latencies lower bounding L_{comm2d} are:

- L_{recv} is the maximum number of messages input by any PE.
- $L_{throughput2d}$ is the maximum number of messages routed by any switch.

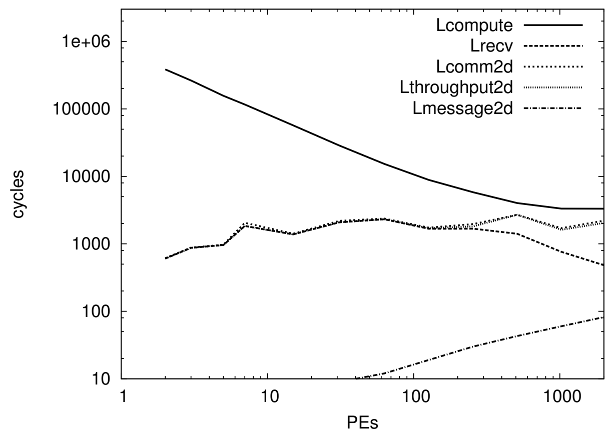


Figure 3.12: Factors contributing to 2D mesh latency for the finite element matrix fidapm37. $n = 9152$ and $m = 765944$.

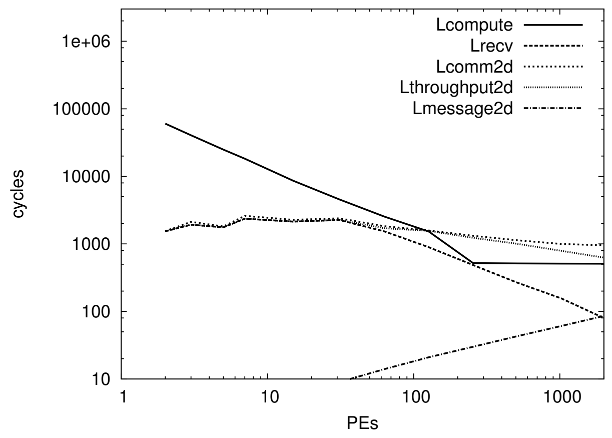


Figure 3.13: Factors contributing to 2D mesh latency for the QCD matrix conf6.0-00l4x4-3000. $n = 3072$ and $m = 119808$.

- $L_{message2d}$ is the maximum message latency.

Contrary to the bidirectional ring, $L_{message2d}$ is not significant, so mesh communication is dominated by throughput constraints rather than message latency constraints. Each PE can only input one message per cycle, which is a bottleneck before N_{PEs} grows large. For larger N_{PEs}, the throughput bottleneck is in the interconnect. The reason for this is that total `source_mem` write throughput for all PEs scales as $\Theta(N_{PEs})$, while total bisection bandwidth scales as $\Theta(\sqrt{N_{PEs}})$. In both cases $L_{recv} < L_{compute}$ means the `source_mem` write bottleneck is never the main scaling limiter. Figure 3.13 shows that interconnect throughput limitations do impede scaling at $N_{PEs} > 128$ for the QCD matrix.

For the finite element matrix, $L_{throughput2d} \approx L_{comm2d}$ shows that the greedy routing algorithm performs very close to an optimal router. For the QCD matrix, the small divergence between

Matrix Map Stage	Seconds
Matrix Partition	70.5
Schedule Communication	30.0
Schedule Computation	8.6
Assemble Memories	1.9
$n = 9152$ SMVM iterations	1.0

Table 3.5: Compute Times for Matrix Map Components for matrix fidapm37 on 16 FPGAs

$L_{throughput2d}$ and L_{comm2d} could be due to message latency or non-optimality of our router.

Since we have not coded mesh interconnect logic, we estimate its impact on area to be at most 30%. Per PE the mesh doubles the number of buffered wires. It increases the number of switches from two to four and doubles inputs for two switches from two to four, which triples the number of LUTs required. The width of `communicate_mem` increases from five bits to eleven bits. So mesh LUT requirements should double or triple bidirectional ring LUT requirements. For the bidirectional ring, control and interconnect logic require 704 slices out of 4065 slices. The rest of the area is occupied by the area dominant FPUs. So LUTs per PE could increase from 4769 by 3×704 to 6177, which is a 30% increase.

3.6 Matrix Mapping Overhead

The software step of mapping a matrix to FPGA memories is on the order of minutes, which is large compared to one SMVM iteration. The number of iterations performed by numerical routines that use SMVM is often less than n. Table 3.5 compares the time taken by Matrix Map stages in software to the time to perform n SMVM iterations. In this example, mapping time requires 10^6 the time of a single iteration. Without improving mapping time, our design is limited to applications which use a fixed matrix structure and run for much longer than a few minutes (e.g. Spice Simulations). Each timestep of a Spice simulation performs a non-linear solve, which is commonly implemented with a sequence of linear solves, each of which can use the Gauss Jacobi solver, which iterates over SMVM. Although nonlinear solvers change matrix entry values, they don't change the structure of the matrix. Since the scheduling algorithm operates on matrix structure, it need only compute the map from matrix entries to `compute_mem` addresses once.

3.7 Scaling and Computational Requirements

Conjugate Gradient (CG) is part of the common NAS benchmark suite and its performance is more often reported than SMVM. As mentioned above, CG computation and communication are dominated by the SMVM kernel and the other operations have little impact on performance. We evaluate

CG performance for an extension of our architecture that supports vector parallel operations: this extension incurs, at most, 20% more cycles for a 16 FPGA design.

Per iteration, CG consists of 1 SMVM, 2 vector dot products, 3 vector-add scalar multiplies, and 2 scalar divisions. Vector-add scalar multiply performs $ax + y$ on vectors x and y, and scalar a. The two scalar divisions each require L_{divide} cycles. Divider latency is typically on the order of tens of cycles, so we conservatively assume $L_{divide} = 100$. A vector-add scalar multiply consists of one scalar broadcast, n adds and n multiplies. A vector dot product consists of an addition reduce and n adds and n multiplies. Adds and multiplies can be pipelined as in SMVM for n/N_{PEs} compute cycles. Each CG operation can immediately follow the previous leaving one L_{ring} latency:

$$\begin{aligned} L_{reduce} &= L_{add} \times n/N_{PEs} + L_{ring} && (3.22) \\ L_{extra} &= L_{ring} + L_{reduce} + 4 \times n/N_{PEs} && (3.23) \\ &\quad + 2 \times L_{divide} \end{aligned}$$

We are interested in comparing the performance of 16 processor machines. Since there are at most 6 PEs per FPGA, the point on Figure 3.7 where $N_{PEs} = 100$ gives us at least 16 FPGAs. Here $L \approx 15000$ and $L_{ring} \approx 700$. For the matrix fidapm37, $n = 9152$. $L_{add} = 13$. Now $L_{reduce} \approx 1900$, so $L_{extra} \approx 3200 \approx (1/5)L$. Without considering performance benefit due to extra floating point operations performed, this shows CG incurs a performance overhead of at most 20%.

Chapter 4

Alternative Hardware Evaluation

Chapter 3 shows the SMVM performance achievable on parallel VirtexII-6000s. Here we evaluate the benefit of choosing a more suitable balance of hardware resources for our application. Restricting our design to exclusively use on-chip memory forces N_{FPGAs} to be too large for many matrices for efficient resource use. We explore the impact of devoting more chip area to memory to allow efficient execution for large matrices. We keep our bidirectional ring architecture the same while changing PE area and memory capacity. Since FPUs are used by many applications and occupy 88% of the reconfigurable logic area in our design, we also evaluate the impact of including custom FPUs in the reconfigurable fabric. Finally we estimate the performance if control logic area is also reduced.

The performance and minimum number of FPGAs for each architectures is shown for the three representative matrices by Figures 4.1, 4.2, and 4.3. The architectures are labeled as:

- 2V6000-4: VirtexII-6000 with no modifications.
- SRAM: Increased SRAM to half logic area. See Section 4.3.
- DRAM: DRAM memory occupies half the logic area. See Section 4.4.
- DRAM_custom_FPU: DRAM memory with reconfigurable FPUs replaced by custom FPUs. See Section 4.5.
- SRAM_custom_only: SRAM memory with custom FPUs and no area for control logic. See Section 4.6.

Section 4.1 explains how we estimate the area cost of the reconfigurable logic we are replacing. Section 4.2 summarizes the effect of the increase in memory capacities for the four alternatives. Succeeding sections use these graphs to show the performance impact and decrease in required N_{FPGAs} for alternative architectures. Section 4.3 shows that sacrificing logic area for memory tends to increase maximum performance for larger matrices, often by a significant amount. Section 4.4 shows that using DRAMs instead of large SRAMs doesn't allow much performance increase for our benchmark matrix sizes. Section 4.5 shows that replacing reconfigurable FPUs with custom FPUs increases maximum performance about 5 times. Section 4.6 shows that setting PEs per FPGA to 105, the maximum allowed by custom FPU area, tends not to increase performance further and

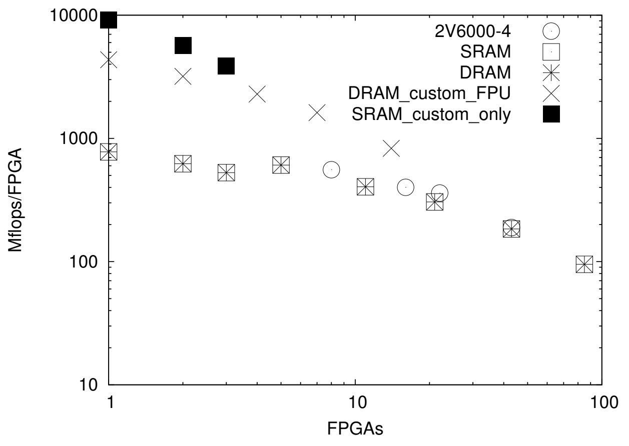


Figure 4.1: Performance scaling for the QCD matrix conf6.0-00l4x4-3000.

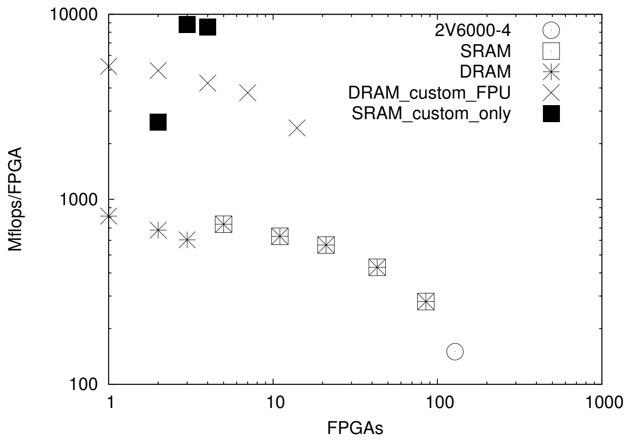


Figure 4.2: Performance scaling for the finite element matrix fidapm37.

to sometimes decrease performance significantly. Section 4.7 shows that for up to 35 PEs/FPGA performance is greater than half the peak performance of 14 Gflops/FPGA.

Table 4.2 shows the peak performances and median performances of each architecture over feasible matrices for one FPGA and for 16 FPGAs. Although this provides a direct comparison when we fix N_{FPGAs} and choose matrices small enough to fit, it does not compare performances when we fix the matrix and choose a feasible N_{FPGAs} to maximize performance.

4.1 Area Estimation

In order to replace reconfigurable logic with extra memory and custom FPUs, we must estimate the area used by the VirtexII-6000's reconfigurable logic. By inspection, the die area of the VirtexII-6000 is approximately $20mm \times 20mm$. This area is used by IO pins, BlockRAMs, 18x18 multipliers, and

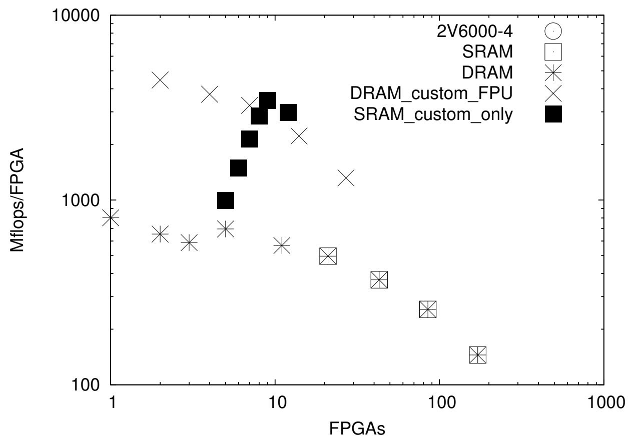


Figure 4.3: Performance scaling for the NAS1 benchmark CG matrix.

CLBs. The reconfigurable logic consists of CLBs, which are LUT clusters, and occupies over half of the $400mm^2$ chip area. Our conservative lower bound for the area used by reconfigurable logic is then $200mm^2$. The feature size for logic is $0.12\mu m$. We define the reconfigurable logic area in terms of λ, half the feature size:

$$A_{reconfig} \geq 5.6 \times 10^{10}\lambda^2 \tag{4.1}$$

Since there are 67584 LUTs, area per LUT is then:

$$A_{LUT} \geq 800K\lambda^2 \tag{4.2}$$

Lower bounding A_{LUT} makes our estimations of relative area sizes conservative in this chapter.

4.2 Large Memory Capacity

Low memory capacity means that many FPGAs are required for medium size matrices. For example the 766,000 non-zero matrix fidapm37 requires more than 23 FPGAs. In the first place, a large number of FPGAs is often not affordable. Further, performance efficiency decreases with the number of FPGAs. Also, as mentioned in Section 3.3, many matrices don't fit on any number of FPGAs because increasing N_{PEs} increases `communicate_mem` depth. Fitting a larger matrix than those found in the Matrix Market Suite is unlikely.

Assuming we avoid all disk access larger problems always require more hardware in the form of more memory. When memory chips are separate from compute chips there is flexibility to choose a suitable compute to logic ratio for the problem. When memory and compute are on the same chip, the ratio may be balanced so not too much of either is wasted in any situation. We replace half of the VirtexII-6000's $5.6 \times 10^{10}\lambda^2$ logic area with memory so the resources used by any application is

Architecture	# memory blocks	Kbits/block	Mbits	Memory Density / DRAM chip density
2V6000-4	144	18	2.5	1/314
SRAM (Section 4.3)	144	144	20	1/39
DRAM (Section 4.4)	6	31,304	183	1/4
DRAM_custom_FPU (Section 4.5)	38	3,143	117	1/7
SRAM_custom_only (Section 4.6)	210	98	20	1/39

Table 4.1: Memory statistics for architectures.

bounded by a factor of two more than the best choice for the application. This leaves us with 3 PEs worth of double precision floating point logic instead of 6. To evaluate each architecture Table 4.1 shows:

- Memory density decrease compared to DRAM chip density with $169\lambda^2$/bit [15]
- Memory capacities (Sections 4.3 and 4.4)

The three example benchmark performance graphs (Figures 4.1, 4.2 and 4.3) show for each architecture:

- Minimum number of FPGAs for the three example benchmark matrices
- Performance scaling

4.3 SRAM Increase

The VirtexII-6000 has 144 dual ported 18Kbit SRAMs called BlockRAMs. Each BlockRAM occupies the same area as a column of four CLBs. Since there are 8448 CLBs, the ratio of CLBs to BlockRAMs is 58:1. Using half the logic area for memory, we multiply the BlockRAM size by 8 to get 144 144Kbit BlockRAMs. This is a conservative estimate since larger memories have lower overhead per bit.

Figure 4.1 shows performance for a relatively small matrix. Increased SRAM capacity increases the feasibility range, allowing one FPGA to be used. Since there are half as many PEs per FPGA, the feasibility upper bound doubles. The maximum performance increases slightly and in the range of 10 to 20 FPGAs it decreases slightly. The difference is much less than the change from 6 PEs per FPGA to 3 PEs should indicate because the original feasibility range is outside of efficient scaling: E is nearly divided by two and L decreases little when N_{PEs} is doubled.

Figure 4.2 shows performance for a larger matrix. The VirtexII-6000 requires too many FPGAs to be efficient, so increasing memory capacity allows maximum Mflops/FPGA over N_{FPGAs} to increase by a factor of four. The matrix in Figure 4.3 is too large to fit on any number of VirtexII-6000s. Increasing SRAM allows it to fit on 20 FPGAs.

4.4 DRAM substitution

Compared to SRAMs, large DRAMs have smaller area per bit and lower power consumption [15]. SRAMs are faster, and have lower area overhead per memory block. We evaluate the area impact of using embedded DRAMs rather than SRAMs. The DRAMs can operate at a higher frequency than the 140MHz imposed by our logic, so the speed difference is not a consideration. We use a linear area model for embedded 256-bit word DRAMs from [15]. The overhead per DRAM block is $1.25mm^2$, and the area per Mbit is $0.6mm^2$. We then normalize in terms of the feature size, $0.13\mu m$.

The large overhead per DRAM makes 144 blocks per processor infeasible. In order to minimize blocks per PE, we show how to convert our design to use two 256-bit DRAM blocks per PE, or 6 DRAM blocks per FPGA. We do this by taking advantage of the regularity of streaming memory access to pack the four PE memories into two wide memories: The compute data path reads `source_mem` as a RAM and it reads `compute_mem` and writes `dest_mem` as streams. The communicate data path reads `dest_mem` as a RAM and reads `communicate_mem` and writes `source_mem` as streams. We pack `source_mem` with `communicate_mem` into `block0` and `dest_mem` with `compute_mem`into `block1`. The compute data path may then read `block0` as a RAM and time multiplex streaming reads and writes to `block1`. Time multiplexing `block1` is feasible since its throughput is 256 bits per cycle whereas `dest_mem` and `compute_mem` require 64 and 78 bits per cycle respectively (Table 2.1). The communicate data path reads `block1` as a RAM and time multiplexes streaming reads and writes to `block0`. For `block0` `source_mem` and `communicate_mem` require 64 and 14 bits per cycle respectively (Table 2.1). We don't consider the extra area overhead to multiplex memory access.

Since PEs/FPGA doesn't change from the increased SRAM case, the significant change is that larger matrices can now fit on one FPGA. For our benchmark matrices, the use of DRAM improves maximum performance over SRAM from 0% to 60%. Of course maximum performance would be significant for larger matrices. One disadvantage is that peak memory throughput is decreased from 5184 bits/cycle to 1536 bits/cycle. The number of independently addressable words also decreased from 144 to 6. So other applications which require high memory throughput or use finer grained memories than 6 per FPGA are no longer efficiently supported.

4.5 Efficient FPUs

Our logic is dominated by floating point multipliers which take substantially more area than custom or ASIC multipliers. In this section we evaluate the increase in terms of peak performance and achieved performance if we use DRAM and custom floating point units, while using FPGA fabric for control logic. This increase is primarily due to an increase in area efficiency rather than clock speed since our referenced DRAMs operate at 200MHz. DRAM still uses 1/2 of the reconfigurable logic area with two DRAM blocks per PE.

The double precision multiply-adder (MADD) in [16] is $0.9mm \times 0.6mm$ with a feature size of $90nm$. This is $2.7 \times 10^8 \lambda^2$/MADD. From Inequality 4.2 area per LUT is at least $800K\lambda^2/LUT$. Then one MADD requires the area of less than 326 LUTs. The non-FPU logic requires 1408 LUTs. With $326 + 1408$ LUTs/PE, 19 PEs fit into half the original logic area. The other half then consists of 38 DRAMs.

Custom FPUs increase peak performance over the original architecture by 4.5 times and over the increased memory architectures by 9 times to 7.6 Gflops. The figures show the maximum achieved performance is between 4 and 5 Gflops. This is at least 5 times that of the increased memory architecture, which is greater than the original architecture. Compared to the original BlockRAMs, throughput increases from 5184 bits/cycle to 9728 bits/cycle, while the number of independent addresses decreases from 144 to 38.

4.6 Arithmetic Logic Only

Since area is dominated by reconfigurable logic after switching to custom FPUs, improving non-FPU logic efficiency may significantly reduce PE area. The control logic is dominated by pipeline registers, so significant savings may result by mapping to reconfigurable logic optimized for pipelined designs or by mapping to custom logic. This section considers the corner case of discounting reconfigurable logic completely, leaving only the custom FPUs. Since half of the reconfigurable area fits 105 MADDs, there are 105 PEs/FPGA. We assume the frequency stays at 200MHz. PEs in this case are too small to support the overhead of DRAM, so we increase the size of BlockRAMs to occupy the other half of the reconfigurable area as in Section 4.3. Two memories per PE are required so the number of BlockRAMs increases from 144 to 210.

Peak performance is 42 Gflops which is 5.5 times the peak when reconfigurable logic is included with custom FPUs. Maximum performance increases by less than two times, however and decreases in some cases. For the small matrix (Figure 4.1) maximum performance is only doubled because there isn't enough parallelism for 105 PEs to be efficient. Since we used SRAM, larger matrices (Figures 4.2 and 4.3) cannot fit onto few enough FPGAs to be efficient. Since fewer than 105 PEs/FPGA may be required to allow the matrix to fit, Mflops/FPGA can drop significantly when N_{FPGAs} is decreased. The number of PEs per FPGA is limited to few enough to use DRAM memory for efficient execution.

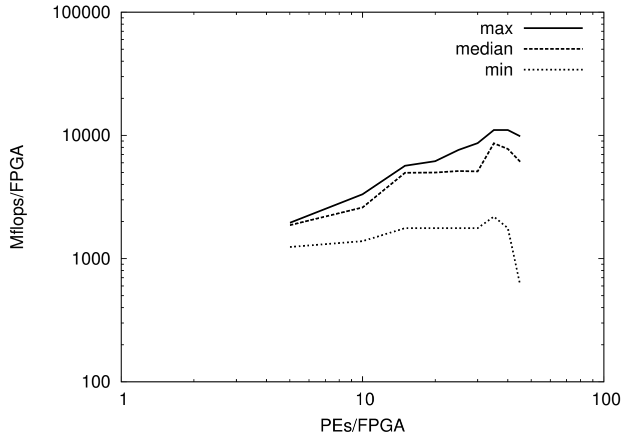


Figure 4.4: Max, Median, Min Mflops/FPGA over benchmark matrices as we increase PEs/FPGA while using DRAM. For a given PEs/FPGA, each matrix contributes its maximum Mflops/FPGA over its feasible N_{FPGAs}.

Architecture	Peak Gflops/FPGA	Delivered Gflops for Single FPGA	Delivered Gflops/FPGA for 16 FPGAs
2V6000-4	2.2	1.5	0.75
SRAM (Section 4.3)	1.1	0.75	0.5
DRAM (Section 4.4)	1.1	0.75	0.5
DRAM_custom_FPU (Section 4.5)	7.6	5	2
SRAM_custom_only (Section 4.6)	42	8	2

Table 4.2: Comparison between architectures of the peak performance and of the median performance over matrices which fit onto one FPGA and onto 16 FPGAs

4.7 Varying PE size

Since SRAM decreases memory density too far for efficient use of custom FPUs (Section 4.6), this section evaluates the performance result of scaling PEs/FPGA from 5 to 45 while using DRAM. This section can be used to evaluate the performance of our architecture after design changes or device changes cause PE area to change.

DRAM overhead forces us to use no more than 45 PEs, or 90 DRAMs per FPGA. Figure 4.4 shows median Mflops/FPGA over matrices for 5 PEs/FPGA to 45 PEs/FPGA. For a given PEs/FPGA, each matrix contributes its maximum Mflops/FPGA over its feasible N_{FPGAs}. At 35 PEs/FPGA, the peak performance is 14 Gflops/FPGA, while the median achieved performance is over half peak at 8 Gflops/FPGA. Performance drops after 35 PEs/FPGA because smaller DRAM sizes prevent matrices from fitting on one FPGA due to the decrease in DRAM density, memory fragmentation and larger $L_{communicate}$.

4.8 Conclusion

Simply increasing SRAM area tends to significantly improve performance for large matrices and slightly decrease performance for smaller matrices. Use of DRAMs instead of increased SRAM is necessary for matrices with $m > 2,000,000$ if using tens or hundreds of FPGAs is not an option. Custom FPUs with DRAM memory increase maximum performance by 5 times to 5 Gflops/FPGA, which is greater than half the peak performance. After moving to custom FPUs, if the size of reconfigurable logic can be reduced, greater than half peak performance can only be maintained up to 35 PEs. This is primarily due to large DRAM block overhead, rather than poor parallel scaling to a single FPGA's PEs. When custom FPUs are used, the 16 FPGA performances shown in Table 4.2 are 1/4 and 1/21 peak. This difference is worse than the VirtexII-6000's performance of 1/3 peak because custom FPUs allow more PEs per FPGA. In order to allow the same degree of hardware scaling, communication overhead must be decreased, perhaps with mesh interconnect as shown in Section 3.5.

Chapter 5

Conclusions and Future Work

5.1 Future Work

This work focuses on design for the SMVM kernel. Our current solver uses sequential software scheduling algorithms to generate the initial memory configuration from a matrix. Memory configuration time should be reduced to allow efficient execution of applications which perform few SMVM iterations (Section 3.6). A complete solution for contemporary FPGAs needs to be general enough to implement a wide range of sparse numerical routines on large matrices (Section 5.1.2).

5.1.1 Reduce Matrix Mapping Time

Section 3.6 reports that the software step of mapping a matrix to FPGA memories is on the order of minutes while one iteration is on the order of 100s of microseconds. Although this is suitable for applications such as long simulations which run on the order of tens of minutes, for the architecture to be useful for a broader set of applications, the Matrix Mapping stages must be streamlined or eliminated. We have not, yet, focused on efficient mapping steps, so all stages could be improved with attention to their runtime. For the largest stages, simple tuning will not be enough. Some promising directions to achieve the large-scale performance improvement required include:

- **FPGA-based clustering**: Exploit the same hardware to rapidly create partitions.
- **Dynamic routing**: Avoid the need to compute a static route.
- **Hardware routing**: Automatically route in parallel via interconnect augmented with route discovery logic [19] [20].

5.1.2 General Applicability

Adapting the architecture to a more complete set of sparse numerical routines requires implementations for vector parallel operations, accumulations, scalar broadcasts, and scalar divides. These operations easily fit into the current ring interconnect and can extend to high-dimension interconnect solutions. The routines CG, GJ, and Lanczos require two extra vector memories per PE of the same size as `dest_mem`. They also require a programmable data path for generalized dataflow between the memories and the adder and multiplier. Like the current design, the algorithm can be divided into stages issued by the CE, with one data path configuration per stage.

5.2 Conclusions

An architecture for performing efficient SMVM on modern FPGAs has been demonstrated. It achieves high scalability and outperforms microprocessors. Our design is mapped to the VirtexII-6000-4 from 2001 and outperforms microprocessors from the same era when comparing both single processors and parallel processors.

We parallelize over dot products for high potential parallelism. Software mapping places data on processing elements to load balance and minimize the communication required. Bidirectional ring interconnect allows inexpensive local communication. These allow our design to scale with a performance penalty less than one half for up to 48 FPGAs and to perform at least three times faster than 16 parallel microprocessors.

We show that placing data for locality asymptotically improves scaling on the bidirectional ring. Random data placement limits the number of efficiently utilizable FPGAs to a constant. Placing data for locality allows the number of efficiently utilizable FPGAs to increase with matrix size. In order to model matrix locality we use a constant Rent parameter.

The bidirectional ring interconnect begins to cause a large communication overhead at greater than 48 FPGAs. This is because its maximum message latency is proportional to the number of processing elements, rather than throughput constraints. We evaluate the communication overhead when two-dimensional mesh interconnect is used instead. When using the mesh, scaling is most often limited by unbalanced computation load. When mesh overhead does limit scaling, it is due to limited interconnect network throughput rather than message latency.

We explore the benefit of FPGA resource distributions balanced for our application. On-chip memory capacities are increased to get memories which are both large and high-throughput. This allows our design to use fewer FPGAs for large matrices, improving their maximum performance per FPGA by up to 5 times. For our benchmark matrices with $m \leq 2,000,000$, the performance benefit of using large DRAMs instead of large SRAMs is marginal. We also estimate the delivered performance increase due to using DRAM and embedding custom FPUs in the reconfigurable fabric to be 3 times, comparing matrices that fit on a single FPGA. For larger matrices the higher memory density gives us an increase of maximum performance per FPGA by up to 15 times.

Bibliography

[1] Matrix Market. <`http://math.nist.gov/MatrixMarket/`>, June 2004. Maintained by: National Institute of Standards and Technology (NIST).

[2] P. Belanović and M. Leeser. A Library of Parameterized Floating Point Modules and Their Use. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pages 657–666, September 2002.

[3] P. Bellows and B. Hutchings. JHDL - An HDL for Reconfigurable Systems. In K. L. Pocek and J. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[4] A. Caldwell, A. Kahng, and I. Markov. Improved Algorithms for Hypergraph Bipartitioning. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 661–666, January 2000.

[5] T. Dunigan. ORNL SGI Altix Evaluation. <`http://www.csm.ornl.gov/~dunigan/sgi/`>, September 2004.

[6] D. S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems.* PWS Publishing Company, 1997.

[7] J. R. Jackson. Scheduling a Production Line to Minimize Maximum Tardiness. Management Science Research Project Research Report 43, UCLA, 1955.

[8] B. S. Landman and R. L. Russo. On Pin Versus Block Relationship for Partitions of Logic Circuits. *IEEE Transactions on Computers*, 20:1469–1479, 1971.

[9] C. Leiserson, F. Rose, and J. Saxe. Optimizing Synchronous Circuitry by Retiming. In *Third Caltech Conference On VLSI*, March 1983.

[10] I. Lloyd N. Trefethen, David Bau. *Numerical Linear Algebra.* SIAM, 3600 University City Science Center, Philadelphia, PA, 1997.

[11] L. Oliker, A. Canning, J. Carter, J. Shalf, D. Skinner, S. Ethier, R. Biswas, J. Djomehri, and R. V. der Wijngaart. Evaluation of Cache-based Superscalar and Cacheless Vector Architectures for Scientific Computations. In *Proceedings of the IEEE/ACM Conference on Supercomputing, 2003*, 2003.

[12] K. Underwood. FPGAs vs. CPUs: Trends in Peak Floating-Point Performance. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 171–180, February 2004.

[13] R. Vudoc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, UC Berkeley, 2003.

[14] R. Vuduc, J. Demmel, K. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply. In *Proceedings of IEEE/ACM Conference on Supercomputing*, November 2002.

[15] Embedded DRAM Comparison Charts. IBM Microelectronics Presentation. http://www-306.ibm.com/chips/techlib/techlib.nsf/products/Embedded_DRAM

[16] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonté, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with Streams. SC'03, November 15-21, 2003, Phoenix, Arizona, USA.

[17] L. Zhuo, V. K. Prasanna Sparse Matrix-Vector Multiplication of FPGAs FPGA'05 February 20-22, 2005, Monterey, California, USA.

[18] R. Melhem A Systolic Accelerator for the Iterative Solution of Sparse Linear Systems IEEE Transactions on Computers, Vol. 38, No. 11, November 1989

[19] R. Huang, J. Wawrzynek and A. DeHon. Stochastic, Spatial Routing for Hypergraphs, Trees, and Meshes. ISFPGA, pages 78–87, February 2003 <`http://www.cs.caltech.edu/research/ic/abstracts/fastroute_fpga2003.html`>

[20] A. DeHon, F. Chong, M. Becker, E. Egozy, H. Minsky, S. Peretz, and T. F. Knight, Jr. A Router Architecture for High-Performance, Short-Haul Routing Networks. ISCA, pages 266–277, May 1994 <`http://www.cs.caltech.edu/~andre/abstracts/metro_isca94.html`>