# Floating-Point Sparse Matrix-Vector Multiply for FPGAs

Michael deLorimier
Dept. of CS, 256-80
California Institute of Technology
Pasadena, CA 91125
mdel@cs.caltech.edu

André DeHon
Dept. of CS, 256-80
California Institute of Technology
Pasadena, CA 91125
andre@acm.org

## 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, it is not uncommon for microprocessors to yield only 10–20% 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. For benchmark matrices from the Matrix Market Suite we project 1.5 double precision Gflops/FPGA for a single Virtex II 6000-4 and 12 double precision Gflops for 16 Virtex IIs (750Mflops/FPGA).

## Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Algorithms implemented in hardware*; B.2.4 [**Arithmetic and Logic Structures**]: High-Speed Arithmetic—*Algorithms*; G.1.3 [**Mathematics of Computing**]: Numerical Linear Algebra—*Sparse, structured, and very large systems (direct and iterative methods)*

## General Terms

Algorithm, Performance, Design, Experimentation

## Keywords

Floating Point, FPGA, Reconfigurable Architecture, Sparse Matrix, Compressed Sparse Row

## 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.

Many real life numerical problems in applications such as engineering simulation, 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 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-zeros 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.

We investigate Sparse Matrix-Vector Multiply (SMVM), the simplest sparse matrix algorithm, on the Virtex II 6000-4. On a single microprocessor, SMVM performs somewhat worse than DMVM due to data structure interpretation overhead. In our FPGA implementation, 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 out 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 (Section 8)). This is a factor of three higher than 16 processor, microprocessor-based parallel machines.

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

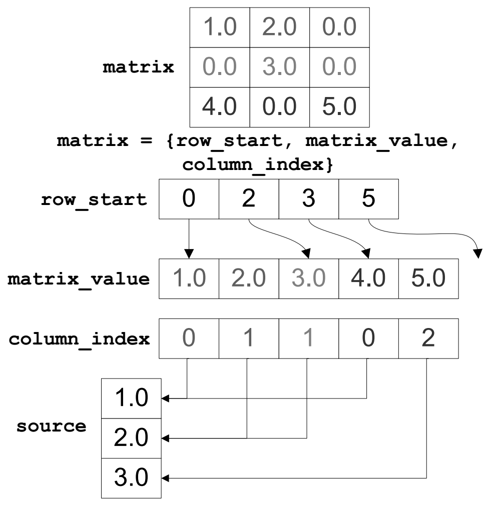


**Figure 1: Compressed Sparse Row Representation of Example Matrix**

## 2. BACKGROUND

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

Iterative SMVM 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 routines such as CG are typically a few vector-parallel operations, which are small compared to the matrix multiply (See Appendix B).

One of the simplest and most efficient sparse matrix representations is Compressed Sparse Row (CSR), as shown in Figure 1. 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].
```

The SMVM algorithm computes $x = Ab$ by performing a dot product on each row. If $A_i$ is the $i$th row, then $x_i = A_i b$, where the dot product is defined as $ab = \sum_i a_i b_i$. It performs dot products from top to bottom (See Figure 2).

## 3. 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. 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 3.2).

```
CSR(row_start, matrix_value, column_index,
    source, dest)
  for (int row=0;row<n;row++)
    accum=0
    for (int i=row_start[row];
        i<row_start[row+1];i++)
      source_value=source[column_index[i]]
      product=matrix_value[i]*source_value
      accum=accum+product
    dest[row]=accum
```

**Figure 2: Compressed Sparse Row SMVM Algorithm**

### 3.1 Compute

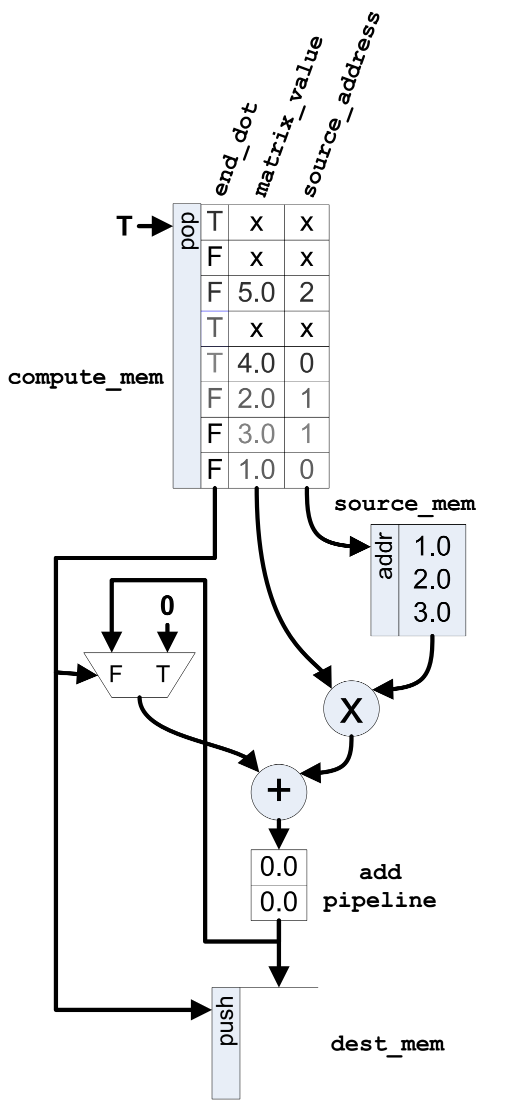
During the compute stage each PE accumulates dot products on the vector `source` to produce the vector `dest`. Dot product partitioning (Section 5) assigns elements of the destination vector, `dest`, into the `dest_mem`s in each of the PEs. Each element of `source` may be used by multiple PEs so the local `source_mem`s redundantly store entries from `source`.

The PE datapath (Figure 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 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 `dest` element 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]*source[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 4 shows datapath pseudocode.

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 FPGA, and these dot products may be computed in parallel. Consequently, we can interleave the independent dot products in C-slow fash-



$L_{add} = 2$ in this example. The memory contents are the initial values to multiply the example matrix and vector in Figure 1. 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 3: PE Compute Datapath**

```
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 4: PE Compute Code ($L_{add} = 1$)**

ion [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 3 and 5. The following recursion computes the accumulation of row $i$; $accum_t$ is computed on cycle $t$:

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

and

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

That is, the accumulation of row $i$ begins on cycle $t_i$ and is interleaved with $L_{add}$ other accumulations. Table 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 4) around $L_{mult}$ and $L_{add}$.

## 3.2 Communicate

The communication stage copies the contents of `dest_mem` to `source_mem`s on different PEs. The interconnect topology is a bidirectional ring as shown in Figure 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.

The communication pattern is fixed for each multiply, so it can be statically scheduled. Switches are controlled by their adjacent PEs. 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`. `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 7 shows the communicate logic for one PE along with its left-ring and right-ring switches.

The pipeline latency of a switch is parameterized so interconnect does not constrain the maximum operating frequency. When ring throughput rather than message latency dominates the number of cycles required for communication, adding ring registers will only result in a small increase in communication cycles.

## 3.3 Controller Element

Also on the rings is a Controller Element (CE) which performs the high level control and IO. The CE first sends the contents of PE memories on the right-ring. `dest_mem`s are loaded rather than `source_mem`s since vector entries map

**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 5: Trace of Compute Memory Values Starting at the Initial Values in Figure 3**

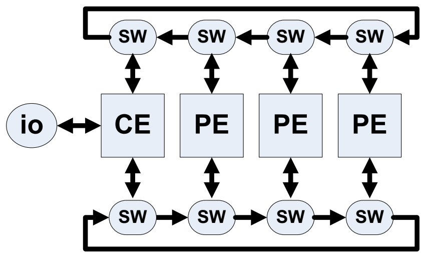


**Figure 6: Bidirectional Ring**

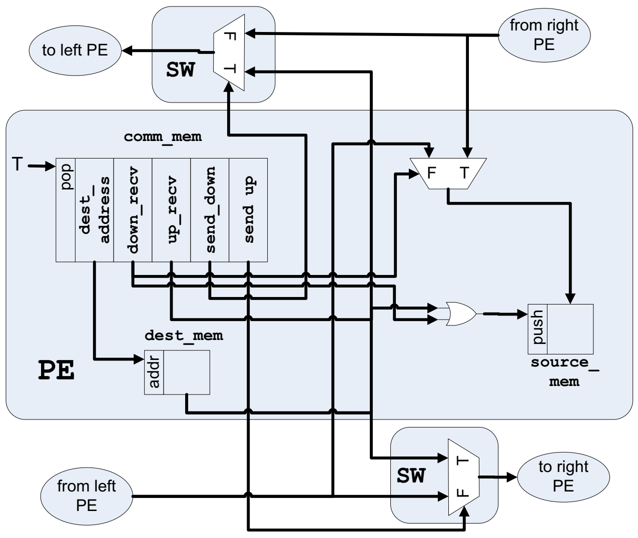


**Figure 7: PE Datapath used for Communication**

onto the `dest_mems`. This requires a communicate stage before iterations start.

## 4. 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.
- $N_{PEs\_per\_FPGA}$ – number of processing elements per FPGA
- $W$ – datapath width; this allows support for single-, double-, and custom-precision floating-point units.
- $M_{depth}[d]$ – memory depth of memory $d$ per PE; $d \in$ {`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).

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

## 5. 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 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 communication 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 placement.

After placement, the computation scheduler load balances dot products assigned to a PE across the $L_{add}$ MAC slots. The quality of this schedule affects the compute stage latency and `compute_mem` size. 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_mems`. Each word that is used outside its PE must be sent to a set of sink PEs. Since

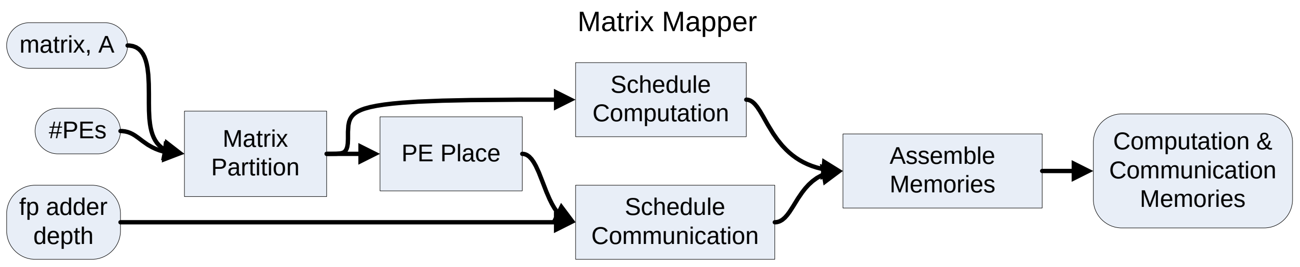


**Figure 8: Matrix Map Stages**

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. Messages are then scheduled greedily with priority to the longest.

## 6. 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:

$$L = L_{ideal\_compute}/E \tag{1}$$

We decompose efficiency into four main components:

$$E = E_A \times E_B \times E_C \times E_L \tag{2}$$

Efficiencies are:

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

Figure 12 and Section 9 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 performs 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 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 affected by $L_{add}$, $m$, $k$, and the partition's load balance. Section 9.3 shows 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. $L_{max\_row}$ 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. When partitioning, there is a trade off between load balancing and minimizing communication.

| partition | place | $p=0$ | $p=1/2$ | $p=2/3$ | $p=1$ |
|---|---|---|---|---|---|
| random | random | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| good | random | $O(n^{1/2})$ | $O(n^{1/3})$ | $O(n^{1/4})$ | $O(1)$ |
| good | good | $O(n^{1/2})$ | $O(n^{1/2})$ | $O(n^{1/3})$ | $O(1)$ |

**Table 1: PE Scaling on Ring: Number of PEs which can be supported with bounded efficiency (See Appendix A)**

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

$$L = L_{compute} + L_{communicate} \tag{3}$$

We then define $E_C$:

$$E_C = L_{compute}/L \tag{4}$$

Appendix A analyzes $E$ modeling $E_A = E_B = E_L = 1$, evaluating several partitioning and placement strategies. Our model of $L_{communicate}$ is based on the both the throughput and latency of the ring. $L_{throughput}$ is the lower bound on $L_{communicate}$ due to data volume; this latency is the maximum number of messages any switch must route. $L_{ring}$ is the latency of one cycle around the ring.

We use the Rent parameter, $p$, [8] to model matrix locality (See Appendix A.2). The average $p$ for most matrices ranges from 0 to 0.6 (See Table 3). Table 1 summarizes the number of processors the ring can support with constant efficiency for each of the models from Appendix A.

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 find $E_L \le 3/4$ (See Section 8).

## 7. MEMORY SIZES

Since our design uses BlockRAM memory only, larger matrices will require more FPGAs. However, as Table 1 and Appendix A 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 $N_{PEs}$).

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,

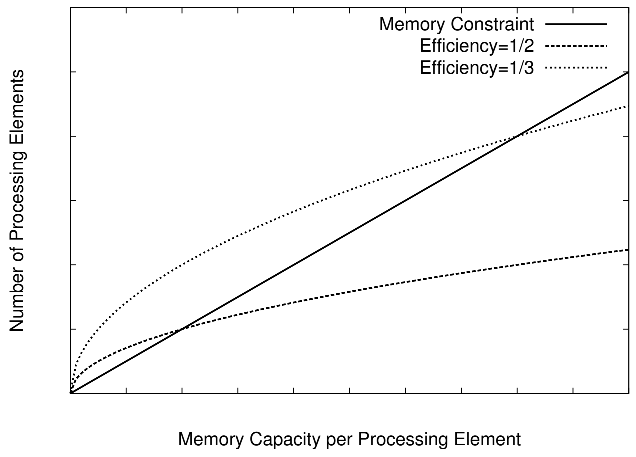


**Figure 9: Feasible regions for $n$ and $N_{PEs}$ with a Constant Memory Size per PE and a Fixed Efficiency Target**

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` 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 ($depth$(`compute_mem`)). Therefore, to bound $E_L$ to a constant, the memory per PE must be constant and hence the $depth$(`compute_mem`) must be a constant. Assuming $E_A$, $E_B$ and $E_S$ are constant, PE memory will be fully utilized. This means:

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

This makes:

$$N_{PEs} \in \Omega(n) \tag{5}$$

From Appendix A constant efficiency requires:

$$N_{PEs} \in O\left(\min\left(n^{1/2}, n^{1-p}\right)\right) \tag{6}$$

Together this means $N_{PEs}$ must be within the feasible region bounded below by the minimum memory requirement (Eq. 5) and bounded above by the communication efficiency requirement (Eq. 6) as shown in Figure 9.

## 8. CONCRETE DESIGN

We implemented the design on a Virtex II 6000-4. Since the computation and communication schedules are static for a partitioning of a given matrix, we know how many cycles a matrix multiply will take.

Since the performance of machines running numerical problems is usually evaluated in terms of double-precision floating-point, we use double-precision arithmetic units. For our FPUs we modified parameterized precision VHDL cores from Northeastern University [2]. We modified both FPUs by pipelining them more deeply. Resulting pipeline depths for the adder and multiplier are 13 and 26 respectively. The overall clock frequency is 140MHz, limited by the multiplier frequency.

| 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 (10) |
| `source_mem` | 64 (64) | 512 (2560) | 2 (2) |
| Total | | | 24 (36) |

**Table 2: Per PE Memory Shapes for 6 PEs (4 PEs) per FPGA**

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

The number of slices taken by the adder and multiplier respectively are 790 and 3276. For both arithmetic units registers are the critical resource. The total number of slices for the Virtex II 6000-4 is 33792, which allows a maximum of 8 PEs/FPGA. The peak is then $2 \times 8 \times 140\text{MHz} = 2240$ Mflops/FPGA. Including other logic (*e.g.* control logic, addressing, interconnect) and floorplanning limits us to 6 PEs/FPGA, which gives us $E_L \leq 3/4$. The CE takes the place of one PE on one FPGA.

Each PE has the four memories that must fit into its available memory. Table 2 shows memory sizes which fit 6 PEs (4 PEs) per FPGA.

All logic except for 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.

## 9. RESULTS

SMVM performance is highly dependent on the matrix. For benchmarking, we used 35 matrices from the Matrix Market Suite [1]. Performance of different matrices on the same number of processors varies by as much as a factor of four. Table 3 lists the matrices and their application areas. Among the matrices we chose, some are the largest matrices from Matrix Market. We also chose matrices to cover a wide range of sizes and applications.

### 9.1 Single Processor Comparison

Table 4 compares the performance of our implementation on one Virtex II 6000-4 to the performance of various microprocessors. The single microprocessor information is a subset of a table in [13], which uses highly tuned SMVM algorithms. 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 Virtex II 6000-4. The Itanium 2 performs relatively well because it has a large cache and high memory bandwidth [13].

### 9.2 Parallel Processor Comparison

Table 5 shows that our implementation scales well to multiple processors. The microprocessor-based implementations may be affected by poor communication and partitioning as discussed in Appendix A. Further, the multiple processor versions may pay operating systems overhead. 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) perfor-

| 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] |
| Virtex II 6000-4 | 2001 | 140 | 2240 | 1500 | 2/3 | |

**Table 4: Performances of SMVM on Single Processors**

| Architecture | Processors | Year | MHz | Peak Mflops/ Processor | SMVM Mflops/ Processor | fraction of peak | 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 | [5] |
| Cray X1 | 16 x MSP | 2002 | 800 | 12800 | *170 | 1/75 | [5] |
| SP4 | 16 x Power4 | 2001 | 1300 | 5200 | *250 | 1/20 | [5] |
| This Work | 16 x Virtex II 6000-4s | 2001 | 140 | 2240 | 750 | 1/3 | |

* denotes NAS CG performance

**Table 5: Performances of SMVM on Parallel Processors**

| 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 | |
| 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 |

($p$ denotes average rent parameter)

**Table 3: Matrix Market Benchmark Matrices**

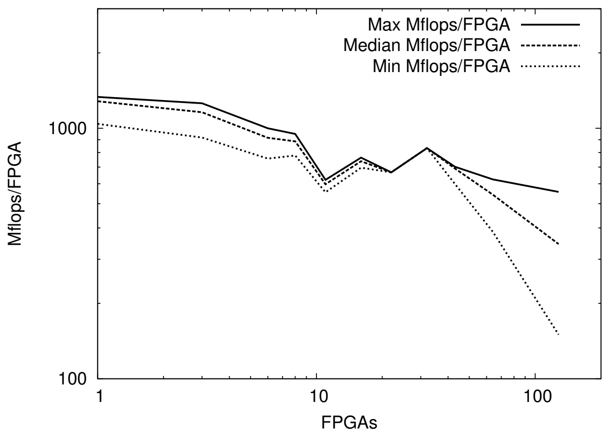


**Figure 10: Mflops Scaling for Benchmark Matrices that Fit into each Number of FPGAs**

mance; since CG is dominated by its SMVM kernel, and its other operations have higher performance than SMVM (See Appendix B), the comparison favors the other machines.

We use Mflops/FPGA as our baseline performance metric for scaling. Figure 10 shows how performance scales with the number of Virtex IIs. Taking the median performances, 16 FPGAs deliver 1/3 peak. We get 1/7 peak at 128 FPGAs. The best parallel architecture in Table 5 drops to 1/20 peak by 16 processors.

## 9.3 MAC Slot Scheduling

The adder slot utilization component of $E_{compute}$, $E_A$, is low if MAC slots are poorly utilized. Figure 11 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 $E_A = 0.80$.

## 9.4 Analysis of Inefficiencies

The largest factor contributing to scaling inefficiency is the large ring interconnect latency, $L_{ring}$. We use Figures 12 and 13 to analyze sources of inefficiency.

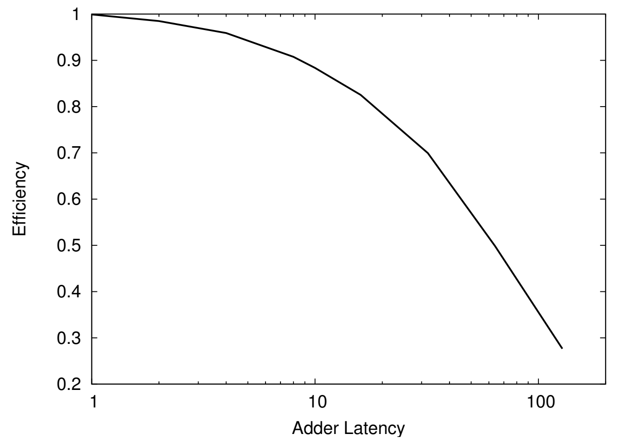


**Figure 11: MAC Slot Efficiency, $E_A$, as a Function of $L_{add}$ for 16 FPGAs**

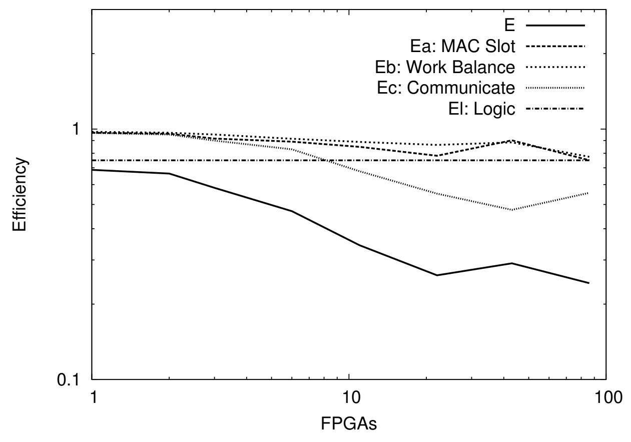


**Figure 12: Scaling of Median Efficiencies for Benchmark Matrices that fit into each Number of FPGAs**

Figure 12 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 13 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 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 up to 95 FPGAs.

## 9.5 Matrix Map Overhead

Currently the overhead to configure memories given a matrix is large compared to Iterative SMVM time. The number of iterations performed by numerical routines that use SMVM is usually much less than $n$. Taking $n$ as an upper bound for the number of iterations, Table 6 compares the time taken by Matrix Map stages in software to the upper bound of logic execution time.

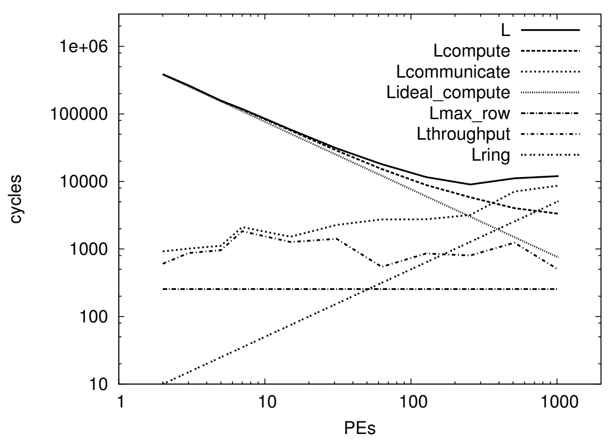


**Figure 13: Matrix fidapm37 Cycle Breakdown: fidapm37 has $n = 9152$, $m = 765944$.**

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

**Table 6: Compute Times for Matrix Map Components for fidapm37 on 16 Virtex IIs**

## 10. FUTURE WORK

A complete solution for contemporary FPGAs needs to implement a wide range of sparse numerical routines on large matrices. Further, memory configuration time must be comparable to or smaller than hardware execution time.

**Communication Latency** From Section 9.4, 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.

**Matrix Mapping** For the architecture to be useful for a broad 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** to exploit the same hardware to rapidly create partitions
- **Dynamic routing** to avoid the need for the scheduling stage

**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.

## 11. CONCLUSIONS

An architecture for performing efficient SMVM on modern FPGAs has been demonstrated. It achieves high scalability by taking advantage of the Virtex II's high BlockRAM memory bandwidth. The compute datapath processes the sparse matrix structure in a streaming fashion so there is no slowdown due to memory latency. We have shown that good matrix partitioning allows us to scale up to 16 Virtex II 6000-4s while maintaining an efficiency of 1/3—much higher than 16-processor, microprocessor-based, multiprocessor systems.

## Acknowledgments

This work was supported by the Microelectronics Advanced Research Consortium (MARCO) and is part of the efforts of the Gigascale Systems Research Center (GSRC). Thanks to Keith Underwood for valuable editorial comments on this writeup.

## 12. REFERENCES

[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.

## APPENDIX

## A. COMMUNICATION MODELS

In this section we study how communication bandwidth constraints affect asymptotic scalability. We evaluate how taking advantage of matrix locality can increase scalability. We find the maximum number of PEs we can use while maintaining a constant efficiency. We analyze how the number of PEs, $N_{PEs}$, scales with matrix dimensions $n$ 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 $n$.

This section models the efficiency $E_C$ from Section 6. We use $L_{ideal\_compute}$ to model the compute stage latency, and $L_{communicate}$ to model the communicate stage latency. Analogously to Equation 4, we model of the fraction of total cycles spent in the compute stage as:

$$E_{C_{ideal}} = \frac{L_{ideal\_compute}}{L_{ideal\_compute} + L_{communicate}} \tag{7}$$

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

Recall the matrix dimension is $n$, and 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:

$$L_{communicate} \leq kn/N_{PEs} \tag{8}$$

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

$$L_{communicate} \geq O\left(N_{PEs}\right) \tag{9}$$

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

$$W_{comm} = \sum_{msg \in messages} L_{msg} \tag{10}$$

$L_{msg}$ 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:

$$L_{communicate} \geq W_{comm}/(2N_{PEs}) \quad (11)$$

Considering both throughput and latency bounds:

$$\begin{aligned} L_{ring} &= L_{ringstage} \times N_{PEs} && (12) \\ L_{communicate} &\geq \max(W_{comm}/(2N_{PEs}), L_{ring}) && (13) \end{aligned}$$

Communication is throughput constrained if and only if:

$$W_{comm}/(2N_{PEs}) \geq L_{ringstage} \times N_{PEs} \quad (14)$$

## A.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 14, communication is throughput constrained:

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

Using Eq. 8, we find:

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

This means we can only use a number of PEs, $N_{PEs}$, 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.

## A.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. 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:

$$inputs(r) = c(r)^p \quad (15)$$

Partitioning well with different size partitions can be used to find the relation. 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 commonly have $p = 2/3$. Figure 14 shows the average number of inputs per partition for our benchmark matrices. The graph has $x = \log(r)$ and from Eq. 15 $y = \log(inputs(r)) = \log(c(r)^p)$. Relating $x$ and $y$ gives $y = \log(c) + px$. So $p$ for a matrix is its plot's slope. 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 $p$. In either case, when $p < 1$ there is locality to be exploited by a partitioner and placer.

## A.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 PE is then assigned to a random partition. 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 $W_{comm} \in \Theta(N_{PEs}^2 \times (n/N_{PEs})^p)$. From Inequality 14, communication is throughput constrained. Using Eqs. 8 and 11:

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

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

## A.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

$$\begin{aligned} L_{throughput} &\propto \sum_{k=0}^{\log(N_{PEs})} L_k && (16) \\ &\propto \sum_{k=0}^{\log(N_{PEs})} (n/2^k)^p && (17) \\ &= n^p \times \sum_{k=0}^{\log(N_{PEs})} (1/2^k)^p && (18) \\ &\propto n^p && (19) \end{aligned}$$

We get Eq. 19 from Eq. 18 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. 8 we get:

$$N_{PEs} \in O(n^{1-p}) \quad (20)$$

Communication may be latency constrained; using Eq. 9 we get:

$$N_{PEs} \in O(n^{1/2}) \quad (21)$$

Considering both throughput and latency constraints, we combine Eq. 20 and 21 in a manner similar to Eq. 13 and get:

$$N_{PEs} \in O(\min(n^{1/2}, n^{1-p})) \quad (22)$$

## A.5 Comparison

Table 1 compares the three types of partitioning and placement with $p = 2/3$ and $p = 1/2$. It shows that scalability is limited to $O(n^{1/2})$ due to ring latency.

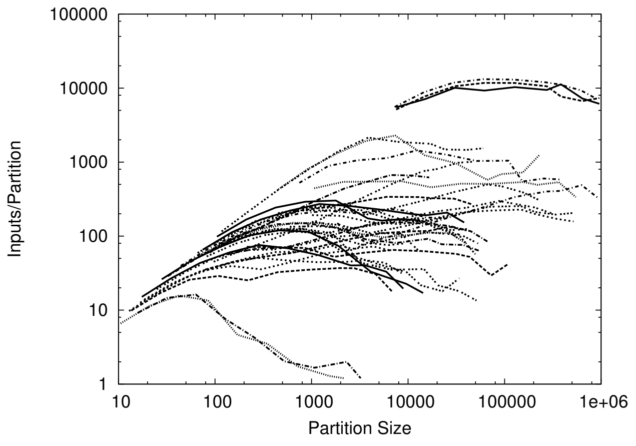


**Figure 14: I/O Scaling for Benchmark Matrices**

## B. 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 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:

$$\begin{aligned} L_{reduce} &= L_{add} \times n/N_{PEs} + L_{ring} && (23) \\ L_{extra} &= L_{ring} + L_{reduce} + 4 \times n/N_{PEs} && (24) \\ &\quad + 2 \times L_{divide} \end{aligned}$$

We are interested in comparing the performance of 16 processor machines. Since there are at most 6 PEs per FPGA, the point on Figure 13 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%.