Evaluation of Architectural Paradigms for Addressing the Processor-Memory Gap

Leonid Oliker, Parry Husbands, Gorden Griem
Lawrence Berkeley National Laboratory
Berkeley, CA
{loliker,pjrhusbands,ggriem@lbl.gov}

Jacqueline Chame
Information Sciences Institute
University of Southern California
jchame@isi.edu

Abstract - Many high performance applications run well below the peak arithmetic performance of the underlying machine, with inefficiencies often attributed to poor memory system behavior. In the context of scientific computing we examine three emerging processors designed to address the well-known gap between processor and memory performance through the exploitation of data parallelism. The VIRAM architecture uses novel PIM technology to combine embedded DRAM with a vector co-processor for exploiting its large bandwidth potential. The DIVA architecture incorporates a collection of PIM chips as smart-memory coprocessors to a conventional microprocessor, and relies on superword-level parallelism to make effective use of the available memory bandwidth. The Imagine architecture provides a stream-aware memory hierarchy to support the tremendous processing potential of SIMD controlled VLIW clusters. First we develop a scalable synthetic probe that allows us to parametize key performance attributes of VIRAM, DIVA and Imagine while capturing the performance crossover points of these architectures. Next we present results for scientific kernels with different sets of computational characteristics and memory access patterns. Our experiments allow us to evaluate the strategies employed to exploit data parallelism, isolate the set of application characteristics best suited to each architecture and show a promising direction towards interfacing leading-edge processor technology with high-end scientific computations.

I. Introduction

The increasing gap between processor and memory speeds is a well-known problem in computer architecture, with peak processor performance improving at a rate of 60% per year, while DRAM latencies and bandwidths improve at only 7% and 20% respectively [15]. To mask memory latencies, current high-end computers now demand up to 25 times the number of overlapped operations required of supercomputers 30 years ago. Further, techniques designed to hide memory latencies, such as increased instruction issue rates, multithreading, and prefetching, may actually increase the memory bandwidth requirements [8]. This so-called "memory wall" is one of the reasons many high performance applications run well below the peak arithmetic performance of the underlying machine. In particular, irregularly structured and data-intensive codes exhibit poor temporal locality and receive little benefit from the automatically managed caches of conventional microarchitectures. In addition, a significant fraction of scientific codes are characterized by predictable data-parallelism that could be exploited at compile time with properly structured program semantics; however, most superscalar general-purpose processors are poor at dynamically exploiting this kind of parallelism. Finally, many scientific programs require a bandwidth-oriented memory system; unlike conventional cache-based memory hierarchies that are entirely organized around reducing average latency time, and generally lack the raw bandwidth required for these applications. This paper presents an evaluation of emerging microprocessor technologies designed to address the processor-memory gap through explicit data-parallelism using three architectural paradigms: vectors, superwords, and streams.

First we examine the VIRAM architecture, which uses a novel processor-in-memory (PIM) design to combine embedded DRAM with a vector-co-processor for exploiting its large bandwidth potential. The PIM technology allows the main RAM to be in close proximity to the processing elements, providing lower memory latency and a significantly wider memory interface than conventional microprocessors. Next we present the DIVA system, which incorporates a collection of PIM chips as smart memory coprocessors and uses wide datapaths to utilize its large memory bandwidth and exploit fine-grained parallelism. Finally we evaluate the Imagine architecture, which provides a stream-aware memory hierarchy to support the tremendous processing potential of its SIMD controlled VLIW clusters.

We develop a scalable synthetic probe called *Sqmat* that allows us to parametize key performance attributes and reveal architectural characteristics of the processors in this study. By varying *Sqmat's* computational requirements, we can explore the main architectural features of the processor, paying attention to the complex interactions among the programming paradigms, ISA, and underlying microarchitecture, while observing the crossover points where different technologies become more suitable. We then present scientific kernels reflecting dense and sparse matrix operations, each requiring a different balance of microarchitectural resources to achieve high performance. The *SPMV* benchmark performs sparse matrix-vector multiplication, and is characterized by irregular data access and low computation per memory access. In contrast, our second scientific kernel *Transitive Closure*, implemented via the Floyd-Warshall algorithm, can be blocked in order to provide a high number of operations per word transferred from memory. Finally we examine the *Neighborhood* benchmark, whose random data access patterns and potential data collisions is particularly challenging for the

data-parallel model. The purpose of this work is not just to compare these processors from a traditional benchmarking perspective. Instead, we use our scientific kernel codes to explore the salient features of these unique architectures, and define the program characteristics best suited for each of these radically different emerging technologies.

II. Architecture, Programming Paradigm, and Kernel Overview

In this section we provide a brief overview of the processors examined in this study, a summary of their programming paradigms, and a description of the scientific kernels used in our experiments.

A. VIRAM

The VIRAM processor [5] is a research architecture being developed at UC Berkeley. A floor plan of the VIRAM-1 prototype chip is presented in *Figure 1*. Its most novel feature is that it is a complete system on a chip, combining processing elements and 13 MB of standard DRAM into a single design. The processor-in-memory (PIM) technology allows the main RAM to be in close proximity to the processing elements, providing lower memory latency and a significantly wider memory interface than conventional microprocessors. The resulting memory bandwidth is an impressive 6.4 GB/s. VIRAM contains a conventional general purpose MIPS scalar processor on-chip, but to exploit its large bandwidth potential, it also has a vector co-processor consisting of 4 64-bit vector lanes. VIRAM has a peak performance of 1.6 GFlop/s for 32 bit data and is a low power chip, designed to consume only 2 Watts of energy.

The hardware resources devoted to functional units and registers may be subdivided to operate on 8, 16, 32, or 64-bit data. When the data width (known as the virtual processor width) is cut in half, the number of elements per register doubles, as does the peak arithmetic rate. The variable data widths in VIRAM are common to other SIMD media extensions such as Intel's SSE, but otherwise the architecture more closely matches a traditional vector supercomputer. In particular, the parallelism expressed in SIMD extensions are tied to the degree of parallelism in the hardware, whereas a floating-point instruction in VIRAM specifies 64-way parallelism while the hardware only executes 8-way. The advantages of specifying longer vectors include a lower instruction bandwidth requirement, a higher degree of parallelism for memory latency masking, and the ability to change hardware resources across chip generations without requiring software changes.

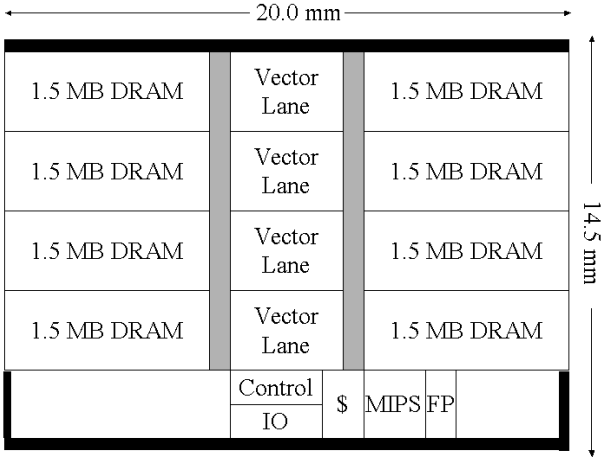

Figure 1: Block diagram of the VIRAM architecture

B. DIVA

The DIVA (Data IntensiVe Architecture) system incorporates a collection of processor-in-memory (PIM) chips as smart-memory coprocessors to a conventional microprocessor. DIVA targets two important classes of bandwidth-limited applications, multimedia and irregular applications, including sparse-matrix and pointer computations. By performing computation directly in memory, streaming multimedia applications obtain high bandwidth to on-chip memories through a 256-bit wide datapath, while irregular applications benefit from very low latency accesses to memory.

DIVA was designed to support a smooth migration path for application software by integrating PIMs into conventional systems as seamlessly as possible. A separate memory-to-memory interconnect enables communication between memories without involving the host processor.

Each DIVA PIM chip is a VLSI memory device augmented with general-purpose computing and communication hardware. Although a PIM may consist of multiple nodes, each of which is primarily comprised of a few megabytes of memory and a node processor, *Figure 2* shows a PIM with a single node, which reflects the focus of the initial research being conducted. Nodes on a PIM chip share a host interface and a single PIM Routing Component for PIM-to-PIM communication. Note that since DIVA was designed for multi-PIM configurations, we expect limited performance from the single PIM system examined in our study. Multi-PIM performance scalability will be addressed in future work.

The PIM node processing logic supports single-issue, in-order execution, with 32-bit instructions and 32-bit addresses. There are two datapaths whose actions are coordinated by a single execution control unit: a 32-bit scalar datapath and a 256-bit wide datapath. The scalar datapath is a standard RISC architecture, augmented with a few DIVA-specific functions for coordinating with the wide datapath. The wide datapath operates on aggregate objects (superwords) of 256 bits, performing SIMD parallel operations on variable-sized fields in the object (8,16, and 32-bit fields). In addition to conventional arithmetic and logic operations, the wide ALU also supports a rich set of operations for manipulating data, including rearrangement of data within a wide operand, transfers between wide and scalar registers and packing and unpacking operations. Furthermore, the wide ALU supports selective execution of instructions on a per-datapath basis, depending on the state of condition codes.

The first DIVA PIM prototype is an SRAM-based, single-node implementation of the DIVA PIM chip architecture. It

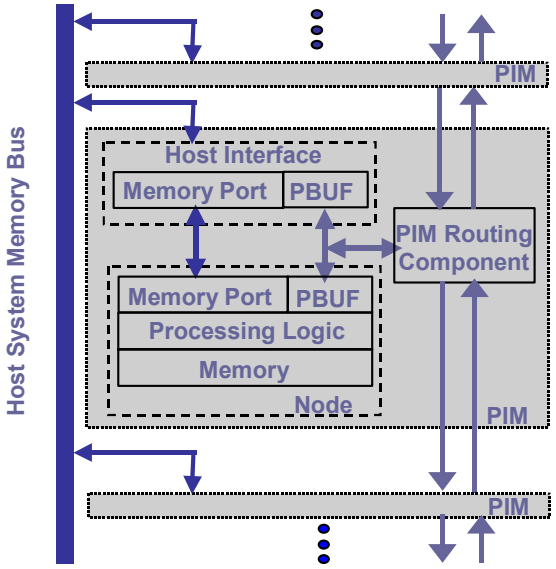

Figure 2: DIVA system and PIM chip organization

includes all architectural features of a DIVA PIM, except address translation and floating-point capabilities, which will be integrated in the second version of the chip. The chip was fabricated through MOSIS in TSMC 0.18m technology, and contains approximately 2 million logic transistors in addition to the 53 million transistors that implement 8 Mbits of SRAM. The current chip, under test, is performing 1.28 GOPS while dissipating only 800mW.

C. *Imagine*

A different approach for addressing the processor-memory gap is through stream processing. Imagine [16] is a programmable streaming microprocessor currently being developed at Stanford University. Stream processors are designed for computationally intensive applications characterized by high data parallelism and producer-consumer locality with little global data reuse. The general layout diagram of Imagine is presented in *Figure 3*. Imagine contains 48 arithmetic units, and a unique three level memory hierarchy designed to keep the functional units saturated during stream processing. The architecture is centered around a 128 KB stream register file (SRF), which reads data from off-chip DRAM through a memory system interface and sequentially feeds the 8 arithmetic clusters. The local storage of the SRF can effectively reuse intermediate results (producer-consumer locality), allowing for the amortization of off-chip memory accesses. In addition, the SRF can be used to overlap computations with memory traffic, by simultaneously reading from main-memory while writing to the arithmetic clusters (double-buffering). The Imagine architecture emphasizes raw processing power much more heavily than the others with a peak performance of 20 GFlop/s for 32 bit data.

Each of Imagine's 8 arithmetic clusters consists of 6 functional units containing 3 adders, 2 multipliers, and a divide/square root. Imagine is a native 32-bit architecture; with support for performing operations on 16- and 8-bit data

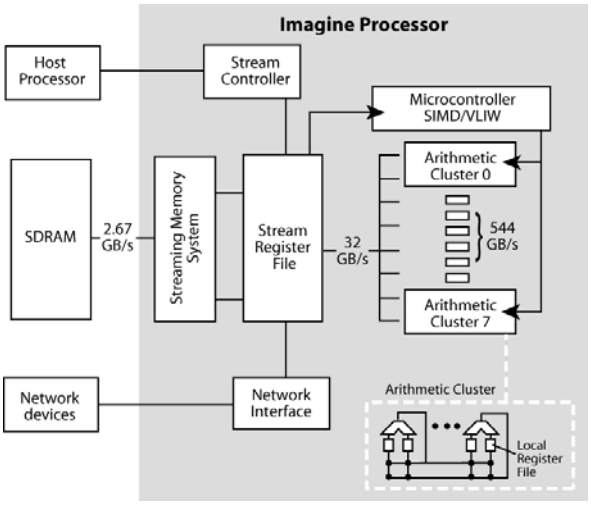

Figure 3: Overview of the Imagine architecture

resulting in two and four times the peak performance respectively. This is analogous to VIRAM's virtual processor widths; however, unlike VIRAM there is no support for 64 bit operations. Thus we restrict our study to 32-bit data elements. A key difference between the architectures is in the way instructions are issued. In Imagine, a single microcontroller broadcasts VLIW instructions in SIMD fashion to all of the arithmetic clusters. In contrast, VIRAM and DIVA use a more traditional single instruction per cycle issue, counting on parallelism within each vector instruction to achieve high performance.

D. *IBM RS6000 Power3*

For comparison purposes, we present actual performance measurements on the IBM RS6000 Power3 [2]. The Power3 is an out-of-order 64-bit PowerPC implementation with a peak (sustained) execution rate of eight (four) instructions per cycle. Like most conventional superscalar architectures, the Power3 relies on cache reuse to reduce memory overhead and it is the programmer's responsibility to write "cache-aware" code. The CPU has a 32 KB instruction cache and a 128 KB 128-way set associative L1 data cache, as well as an 8MB off-chip 4-way set associative L2 cache. The Power3 experiments reported in this paper were conducted on a single CPU of the 6080-processor NERSC system running AIX5.1; currently rated as the fifth most powerful supercomputer [39]. Each 375 MHz processor contains two FPUs that can issue a multiply-add per cycle, for a peak performance of 1.5 Gflop/s. Codes were compiled using the IBM *xlc* compiler. Since purpose of this work is to study emerging microarchitectural paradigms, Power3 results are provided for a baseline comparison, without detailed analysis.

Table 1 summarizes the high level differences between the VIRAM, Imagine, DIVA and Power3 architectures. Notice that Imagine has an order of magnitude higher peak performance, while VIRAM has twice the memory bandwidth and consumes half the power. Also observe that VIRAM and DIVA have enough bandwidth to sustain one operation per memory access, while Imagine requires 30 operations to

amortize one word of off-chip memory (2.5 operations for SRF references). The power consumption reported for DIVA is a projection for the second PIM chip, which will include floating-point capabilities.

	VIRAM	Imagine Memory	DIVA (1 PIM)	Power3
Bandwidth GB/sec	6.4	2.7	1.77	1.6
Peak Flops GFlop/s (32 bit)	1.6	20	1.3	1.5
Peak Flop/Word	1	30	1	3.75
Clock Speed MHz	200	500	166	375
Chip Area	15x18mm (270 mm^2)	12x12mm (144 $mm^{2)}$	9.8x9.8mm (96 mm^2)	270 mm^2
Data widths supported	64/32/16 bit	32/16/8 bit	64/32/16/8	64
Transistors	130 Million	21 Million	55 Million	15 Million
Power consumption	2 Watts	4 Watts	1.6 Watts	33 Watts

Table 1:Highlights of VIRAM, Imagine, DIVA and Power3 architecture

E. Programming Paradigms and Software Environments

The vector programming paradigm [22] of VIRAM is well understood and can leverage years of algorithmic research as well as sophisticated compiler technologies. Logically, a vector instruction specifies the parallel operations to be performed on all elements of the vector register. However, at the hardware level each vector instruction splits into multiple element groups that then perform the operations. For example, when operating on 32-bit data in VIRAM, the logical vector length refers to 64 elements while the physical configuration contains only 8 lanes. Therefore each vector instruction results in the execution of 64/8=8 element groups, where each group uses the actual vector hardware to process 8 elements at a time.

DIVA can be programmed using conventional solutions from parallel computing, rather than requiring a programming paradigm specific to DIVA or to PIMs. As a system-level programming strategy, DIVA has adopted Unified Parallel C (UPC) [40], a relatively new parallel programming language. UPC was developed as a unification of the best ideas among several research C compilers that support a global address space, and allow high-level specification of data distribution in an SPMD abstraction for high-end shared-memory, distributed-shared-memory and even distributed-memory parallel systems. At the PIM level, each node supports SIMD parallel operations on different field widths, 8-bit, 16-bit, 32-bit and 64bit (this type of fine-grain parallelism is referred to as superword-level parallelism, or SLP [23]). The DIVA PIM compiler targets SLP and also exploits superword-level locality, via compiler-controlled caching in the wide register file [34].

Imagine supports the relatively new stream programming paradigm, designed to express the high degree of fine-grained parallelism necessary to effectively utilize the large number of functional units. The stream programming model organizes data as streams and expresses all computations as kernels [19]. A stream is an ordered set of records of arbitrary (but homogeneous) data-objects. For example, in a finite-element scientific simulation the computational stream could contain a set of records, where each record element represents various physical components of the experiment (such as pressure, velocity, position, etc.) Vectors, on the other hand, are restricted to operating on basic data types, and must decompose complex records into vectors of separate elements. Kernels perform computation on entire streams, by applying potentially complex functions to each stream record in order. However, kernels cannot make arbitrary memory references and are limited to only accessing data from the SRF in a sequential fashion. The kernel memory reference restrictions allow the memory subsystem to effectively provide data to the large number of functional units. However, these memory access limitations increase programming complexity, especially for irregularly structured applications. This approach can be viewed as a generalization of vector computing with user defined, coarse-grained kernel operations replacing traditional vector instructions. In addition, chaining is also generalized through the use of the Stream Register File and producer-consumer locality.

Vector, SLP and stream programming paradigms provide methods for expressing the data parallelism of an application. Providing for explicit parallelism in the ISA allows the underlying hardware to directly support vectors, superwords or streams in an energy-efficient manner. The application performance, however, is highly correlated to the fraction of the application amenable to data parallelism. A key distinction between the vector or superword models and the stream model is that the Imagine architecture supports streams of multi-word records directly in the ISA, as opposed to VIRAM whose ISA support is limited to vectors of basic data-types, or DIVA whose ISA supports SIMD operations on superwords for objects of the same data types. Going back to our finite-element example, Imagine is able to access the multi-word data records of the simulation in a unit-stride fashion from main memory. Appropriate reordering is then performed in the on-chip memory subsystem, before passing the correctly structured data to the SRF. However, in vector architectures, strided accesses are required to load each basic data type of the underlying physical component causing potential memory overheads. In architectures with support for SLP, such as DIVA, it is necessary to pack the objects of the same data type into a superword before performing the parallel operation. This permits Imagine to access off-chip main memory in a more efficient manner. Additionally, organizing streams as

multi-word records also increases kernel locality, allowing for efficient VLIW processing by each of the functional units. Other advantages of multi-word parallelism include the potential of reduced programming complexity and low instruction bandwidth.

We end this section with a brief description of the software environment. In VIRAM, applications are coded in C using the vcc [22] vectorizing compiler. However, it is occasionally necessary to hand tune assembly instructions to overcome the deficiencies of the compiler environment. In DIVA the applications are written in C and compiled by a SUIF-based PIM compiler that supports SLP and compiler-controlled caching in superword registers. The output of SUIF-based compiler is an optimized C program, augmented with special superword data types and operations. The optimized C-augmented code is then passed to a GNU backend, modified to support superword data types and operations of the DIVA ISA. As in the VIRAM code, it was necessary to hand tune the assembly code, to overcome the limitations of the preliminary implementation of the DIVA PIM compiler.

In Imagine, two languages are used to express a program: the StreamC language is used to coordinate the streaming of data while KernelC is used to define the computational kernels to be performed on each stream record. Separate stream and kernel compilers then map these two languages to the ISA of the stream controller and micro-controller respectively. The Imagine software environment allows for automatic code optimizations such as loop unrolling and software pipelining, as well as visual tools for isolating performance bottlenecks.

The results reported in this paper were gathered from the VIRAM, DIVA and Imagine cycle-accurate simulators. Since these are academic research projects, the reported clock speeds are conservative and do not necessarily reflect the potential of these systems if they were to be designed in a commercial environment. Therefore our performance comparison focuses on simulated cycles instead of Mflop/s rates.

F. Kernel Overview

The first scientific kernel we examine is sparse matrix vector multiply (*SPMV*). This is one of the most heavily used algorithms in large-scale numerical simulations, and is a critical component in data mining, as well as signal and image processing applications. For example, when solving large sparse linear systems or eigensystems, the running time is generally dominated by the *SPMV* kernel. The performance of sparse matrix operations tends to perform poorly on modern microprocessors due to the low ratio between arithmetic computation and memory accesses. Additionally, the irregular data access of this algorithm is inherently at odds with cache-based architectures.

Our second benchmark problem is to compute the *Transitive Closure* of a directed graph in a dense representation. Finding the Transitive Closure [9] of a directed graph (also known as shortest path) is an important problem with applications in communications, transportation, and operations research. Unlike *SPMV*, this is a computationally intensive code, requiring $O(n^3)$ operations on an $O(n^2)$ data set. However, blocking this algorithm for efficient cache reuse is nontrivial due to the complex data dependency requirements [29].

Finally we examine the *Neighborhood* Stressmark, taken from the DIS suite, which estimates the GLCM entropy and energy of an input image [10]. In this benchmark, the main computational kernel involves computing histograms of the sums and differences of neighboring pixel values. Because there are dependences involved in updating the histogram, implementing this operation can be quite challenging on data parallel architectures.

It is important to note that although we have attempted to minimize kernel execution costs on all three architectures, performance can inevitably be improved through further program optimization and algorithmic developments. This holds true for just about any benchmarking experiment, and is particularly relevant for our set of experiments; since we are examining emerging technologies whose tools, software environment, and programming paradigms are still areas of active research.

III. Insights Into the Architectures

In order to gain insights into the architectural differences among the processors, we constructed a scalable synthetic probe called *Sqmat.* This specially designed microbenchmark has several tunable parameters used to isolate key characteristics of the systems, and capture the performance crossover point of these radically different technologies.

A. Sqmat Overview

The computational task of *Sqmat* is to square a set of L matrices of size NxN repeatedly M times. By varying N and M, we can control the size of the computation kernel, as well as the number of arithmetic operations per memory access. In addition, by varying the number of matrices (L) we can correlate the vector/stream length with performance. This way we can extract a performance profile of each processor that reveals, in practice, the number of arithmetic operations required to amortize the cost of a memory access.

The squaring of each NxN matrix requires N^3 multiplications and $N^2 \cdot (N\text{-}1)$ additions, while requiring $2N^2$ memory accesses (loading and storing 32 bit words). On VIRAM the minimum number of cycles (algorithmic peak) required to perform M repeated squarings of L matrices is $L \cdot M \cdot (2N^3 - N^2)/8$, since each of the 8 vector lanes can perform one 32-bit flop per cycle. Additionally, the total number of operations per word of memory accessed in VIRAM is $M \cdot (2N^3 - N^2)/2N^2 = M \cdot (2N\text{-}1)/2$. The analysis for DIVA is similar, since the wide ALU can process eight 32-bit words concurrently. However, the situation is somewhat different for Imagine since it contains multiple functional units per cluster and operates in VLIW fashion. To calculate Imagine's algorithmic peak performance, we can effectively ignore the cost of addition operations because Imagine can perform 3 adds and 2 multiplies per cycle, while the *Sqmat* benchmark requires fewer additions than multiplications. As a result Imagine's peak performance for *Sqmat* requires only

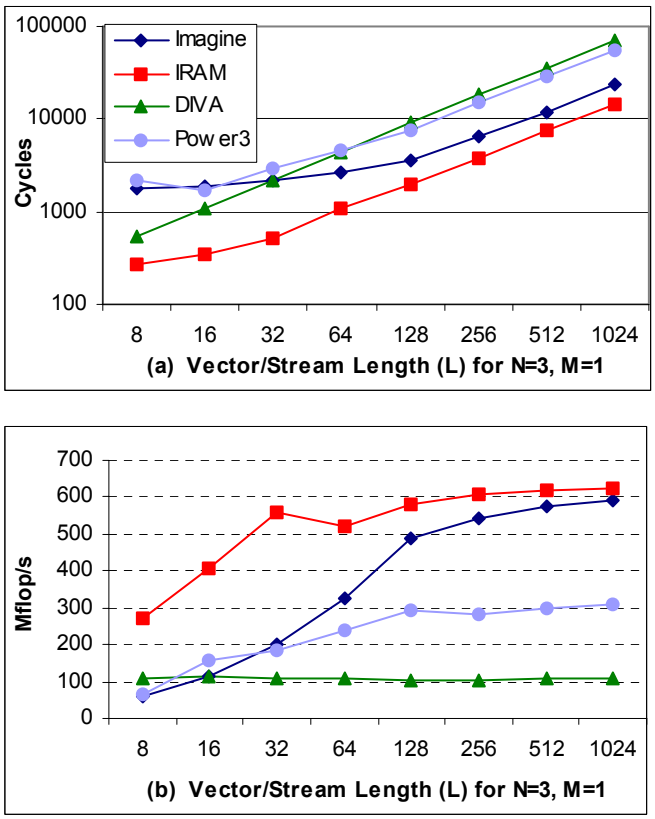

Figure 4: *Sqmat* with low ops per word (a) cycles (b) Mflop/s

$L \cdot M \cdot N^3/16$ cycles, since each of the 8 clusters can perform 2 multiplies per cycle. Additionally, the ratio between the number of multiplies performed per memory access is $M \cdot N^3/2N^2 = N \cdot M/2$. Thus for the *Sqmat* example, Imagine is required to sustain about twice the memory bandwidth of VIRAM and DIVA to keep its functional units optimally saturated. Finally, note that due to limitations imposed by the number of VIRAM vector registers, N could be no larger then 3 for these experiments.

B. *Sqmat Performance*

By varying the number of times the matrix is squared and its size we can determine the performance of each processor in different regimes where we change the number of arithmetic operations per memory access. We examine *Sqmat* under both low and high computational intensities, while highlighting the relevant architectural features and performance crossover. Our goal is not use *Sqmat* for simply benchmarking these systems, but rather as a tool for gaining insight into their key architectural features.

1) *Low Operations per Memory Access*

Our first experiment examines the performance of *Sqmat* when computational intensity is low. Figures 4a and 4b show cycles and Mflops (respectively), for a single matrix squaring (M=1) of a 3x3 matrix (N=3), with varying vector/stream lengths (L=8..1024). Limiting this example to only a single squaring of the matrices causes relatively few operations per word of data access and results in high stress on the memory system. In addition, the short vector/stream lengths deleteriously affect the performance of all the architectures. As the vector length increases, we can examine the architecture's ability to overlap computation with data access.

Power3 results are provided for a baseline comparison. For efficient local register use, each matrix is copied into a local array. Additionally, the matrix multiplications are hand unrolled to allow for maximal use of the multiple functional units.

For the VIRAM experiments, each matrix is read into the vector registers, squared the appropriate number of times in the vector lanes and written back to memory. Figures 4a and 4b show that VIRAM performance starts low but quickly grows with L to a reasonable fraction of *Sqmat's* algorithmic peak performance, achieving 560 Mflops (and 35% of peak) when $L \geq 32$. The vector pipelines effectively hide memory access overheads by overlapping loads with arithmetic operations. In addition, the on-chip DRAM allows for high bandwidth and low latency memory access. VIRAM thus achieves a surprisingly large fraction of its peak performance considering the low volume of required computations and short vector length.

The DIVA version of *Sqmat* also performs the matrix multiply in the register file, and exploits superword-level parallelism. Each matrix row is loaded into a wide register (we chose not to keep more than one row per register to avoid overheads due to packing and unpacking data). The data is then replicated into a set of wide registers in order to perform the matrix multiply. Since each register can keep up to eight single-precision floating-point values, the registers are not fully utilized. More importantly, the wide ALU performs only three useful operations per cycle, out of the eight 32-bit datapaths that are capable of performing eight floating-point operations per cycle.

As can be seen in Figures 4a and 4b, the stream length does not affect performance of DIVA because the first DIVA PIM chip does not support overlapping computation and memory access. Therefore each memory access incurs the full memory access overhead, and even though the on-chip latency is low (3 and 7 cycles for page-mode and random-mode accesses, respectively), the impact is significant. Thus the low computational intensity of M=1 is too small to amortize the cost of memory access on DIVA, causing the system to achieve no more than 8% of its algorithmic peak. A closer analysis reveals that on average 50% of the time is spent transferring data from the on-chip memory.

Imagine's *Sqmat* implementation stores each matrix as a multi-word entry in the data stream. Each arithmetic cluster receives a single matrix (one stream element), performs the matrix multiplication in local registers and writes out the resulting matrix as an element of the output stream. Since Imagine's stream model requires large number of arithmetic operations per memory access to effectively use the underlying hardware, this computational balance is poorly suited for the Imagine architecture. The computational rate is too low to amortize off-chip memory bandwidth, and the SRF is not being used effectively since there is no producer-consumer locality in this example. Performance for low L is

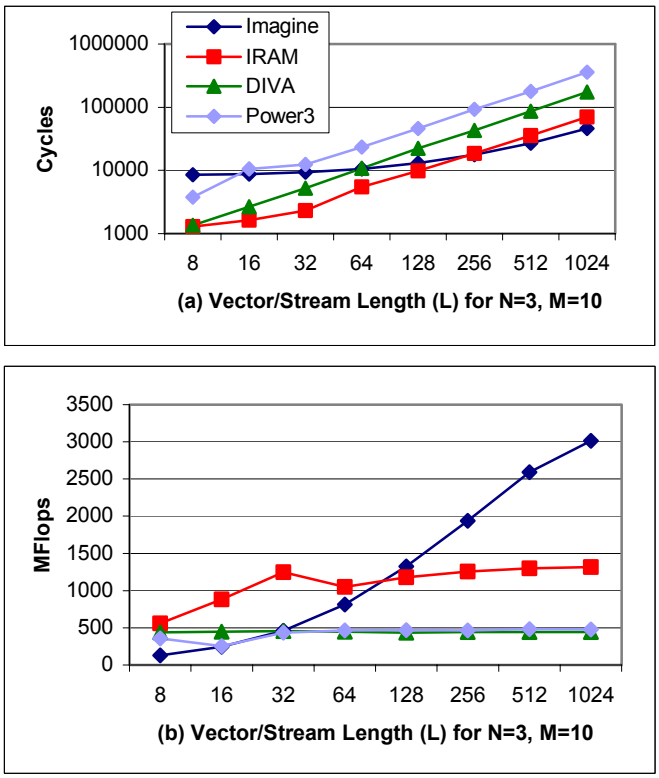

Figure 5: *Sqmat* with high ops per word (a) Cycles (b) Mflop/s

rather poor, achieving 200 Mflop/s (2% of algorithmic peak) for L=32.

Another requirement for good streaming performance is that the stream must be long enough to hide memory latency. *Figure 4* shows that as L is increased from 8 to 1024, peak performance gradually improves, but plateaus at about 590 Mflop/s (about 7% of peak). For each kernel called, there are a number of overheads, including: sending the instructions from the host to the microcontroller, scheduling the SRF, and filling/draining the software pipeline. Thus performance is expected to improve with larger L since these costs are amortized. Additionally, increasing the stream size helps amortize expensive off-chip memory latency.

This example demonstrates that the architectural balance of VIRAM (and to a lesser extend DIVA) is better suited for this difficult class of problems, characterized by low computational requirements and relatively short vector lengths.

2) High Operations per Memory Access

Figures 5a and *5b* show performance results for a computationally intensive *Sqmat* experiment where each (N=3) matrix is repeatedly squared 10 times (M=10) for a variety of vector/stream lengths (L=8..1024). As expected, all architectures perform better as M increases since there is more required computation for each word of data access.

VIRAM now shows high performance $L \geq 32$, achieving over 1180 Mflop/s (78% algorithmic peak), about a factor of two improvement over the M=1 benchmark. Since this is relatively close to its peak, only a slight increase to 1317 Mflop/s is attained for the largest vector length (L=1024). For DIVA, performance is now 440Mflop/s (about 33% of algorithmic peak) regardless of the vector length. Once again this is an artifact of the preliminary chip version that does not support overlapping computation with pipelined memory fetches. However, since the computational intensity has grown by a factor of 10, the cost of memory transfers is amortized and the average time due to memory latencies is reduced to only 15% of the execution time.

Performance on the Imagine architecture is now most impressive, achieving over 3Gflop/s for the largest stream length of 1024. The computational requirement of this benchmark is sufficient to effectively utilize the large-scale processing power of Imagine. One reason for the impressive (factor of 5) improvement in Imagine's performance is that the computational kernel is now significantly bigger. For small M, the number of arithmetic operations per kernel call is small, and the fixed overheads of each kernel call can dominate performance. These overheads include reading and writing from the SRF to the clusters, and filling/draining the kernel pipeline. It is also important to note that although Imagine is showing impressive raw performance, it still achieves less than 40% of its algorithmic peak for this experiment, even though the ratio between operations and memory accesses is now 15. This shows that for the Imagine architecture, a very large computational intensity is required to fully utilize the tremendous processing power of the underlying hardware.

3) Performance Crossover

Finally, we can examine Figures 4b and 5b to find the performance crossover points of the architectures. For the low operations experiment (M=1), there is no performance crossover with regards to VIRAM. The VIRAM architecture has an advantage due to its low latency memory access and ability to effectively process short vector computations.

For the high computational intensity example (M=10), Imagine's performance starts below DIVA and VIRAM for small stream lengths; but as the vector/stream length increases ($L \geq 32$ and $L \geq 128$ respectively), the raw processing power advantages of Imagine become apparent. Codes characterized by this balance of computational intensity and memory requirements would greatly benefit from Imagine's streaming architecture. In fact, increasing the computational intensity results in even more performance (in *Sqmat*) and this is further evidenced by Imagine achieving over 13 Gflop/s on Complex QR decomposition [19].

IV. Sparse Matrix Vector Multiplication (SPMV)

The Sparse Matrix-Vector Multiplication (*SPMV*) algorithm requires random memory access patterns and a low number of arithmetic operations. It is common in scientific applications, and appears in both the DIS [10] and NPB [3] suites as a kernel of a Conjugate Gradient solver. For the *SPMV* kernel we examine several implementation strategies, each highlighting different aspects of the underlying architecture. We chose two matrices for this experiment, each with different characteristics that enable us to explore how

architectural and programming differences affect performance. The first matrix *LSHAPE* is from Harwell-Boeing collection and represents a finite matrix problem. It is a 1008x1009 matrix with an average of 6.8 nonzeros and a maximum of 7 nonzeros per row. Our second matrix *LARGEDIS* is the same one used in previous IRAM experiments [13], and contains a pseudo-random pattern of non-zeros using a construction algorithm from the DIS specification [10], parameterized by the matrix dimension, and the number of nonzeros. This input matrix size is 10000x10000 with an average of 18 nonzeros and a maximum of 82 nonzeros per row.

A. Implementation Details

We consider two algorithmic approaches for *SPMV* on VIRAM [13], each reflecting a different optimization strategy for vector architectures. Our first strategy uses the segmented sum (*Segsum*) algorithm, originally developed for the Cray PVP [6]. The data structure is an augmented form of the commonly used Compressed Row Storage (*CRS*), with some additional control complexity. Since VIRAM's performance suffers from strided memory accesses due to its limited number of address generators, we modified the original Cray code to use unit stride. Our second approach uses the *Ellpack* (or Itpack) format [20], which forces all rows to have the same length by padding them with zeros. This implementation increases the number of operations performed, but increases data parallelism by allowing vectorization across the rows.

Due to the mixed regular/irregular nature of data accesses, DIVA's *SPMV* (also using CRS) only exploits superword-level parallelism for the regular portions of the computation. The dense vector accesses are loaded into wide registers, and the dense vector multiplies are performed in parallel in the wide floating-point unit. Some of the address computation is also performed in parallel.

The DIVA implementation performs the accumulations into the sparse matrix in a sequential fashion. Selective execution is used to perform the accumulate in only one element of the superword currently in a wide register. Further performance improvements are obtained by reordering memory accesses and grouping streaming accesses to the dense arrays to achieve page-mode memory access latencies.

A key component of Imagine's streaming paradigm is that the computational clusters can only access data in a sequential fashion from the SRF. However *SPMV* requires irregular data access to properly index the source vector. Therefore, in Imagine's *SPMV* implementations, the data are properly reordered from main-memory into the SRF to avoid the need for any indirect addressing during computation. Additionally, the indexed source vector stream is expanded to as many elements as in the sparse matrix, since it is not possible to arbitrarily access the vector data.

Our first Imagine implementation (*streams*) leverages the stream concept of producer-consumer locality. Here, in addition to the matrix and indexed vector, the computational kernel receives a third (sentinel) stream indicating which nonzeros entries are at the end of a row. Based on this information, the arithmetic clusters selectively sum two elements if they are determined to be on the same row. The partial sum is repeatedly passed through the computational kernel until the dot product summation is complete. Our second Imagine strategy is similar the VIRAM *Ellpack* algorithm. This approach fills the rows of the sparse matrix such that each has the same number of non-zeros. Each of the eight arithmetic clusters then performs all of the required floating point operations on a given row, and outputs the corresponding entry of the result. This results in a very simple kernel whose performance is dependent on the length of the row.

For the Power3 implementation, we implemented a variety of matrix storage formats, with the CRS version producing the best performance.

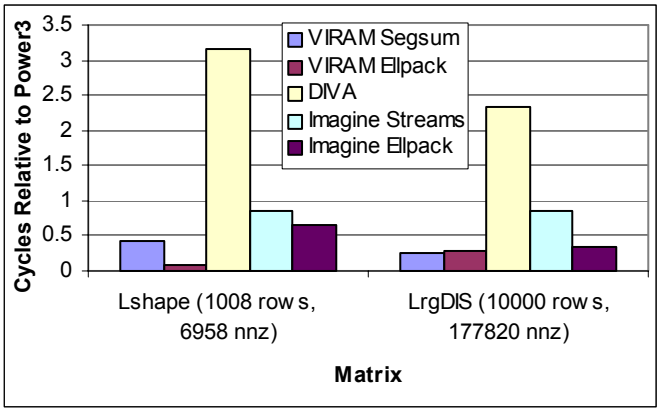

Figure 6: SPMV Performance relative to Power3

B. Performance Results

Figure 6 presents the relative number of Power3 cycles necessary to compute *SPMV* for the LSHAPE and LARGEDIS matrices. Note that that algorithmic peak differs for each architecture: on VIRAM and DIVA the peak is 8 operations per cycle (one for each vector lane/32-bit datapath of a wide FPU), while on Imagine arithmetic peak performance is 32 operations per cycle (2 multiplies and 2 adds for each of 8 clusters).

VIRAM's performance on *SPMV* is surprisingly good considering that on average only a small number of the row entries are nonzeros (7 and 18 respectively). The S*egsum* algorithm, specifically designed for vector architectures, allows VIRAM to compute the *SPMV* iteration about 2.4 and 3.8 times faster than the Power3 (in cycles) for the Lshape and LargeDIS matrices respectively. The *Segsum* approach allows the vectorization across multiple rows with varying numbers of nonzero entries. VIRAM's *Ellpack* also shows impressive performance, reducing Power3's required cycles by factors of 10 and 3.3.

DIVA's poor performance on *SPMV* is due to the lack of parallelism, even more so than VIRAM, since exploiting superword-level parallelism [23] on DIVA requires each superword operand to be in a wide register. If the objects in a superword operand reside in contiguous memory locations, they can be loaded directly into a wide register (but it may be necessary to align the superword). Otherwise the objects need to be packed into a superword, either in memory or in registers [34]. Since in LSHAPE the maximum number of nonzeros

per row is 7, DIVA's *SPMV* does not exploit SLP, since the cost of packing a variable number of nonzeros in a superword would offset the benefits of parallelism. For LARGEDIS the average number of nonzeros per row is 18, allowing SLP to be exploited to some extent.

Imagine's *Streams SPMV* implementation required fewer cycles than the Power3 (about 80%), but achieved lower performance than VIRAM. We believe that this was partially due to the unpredictable length of the output streams after each kernel cycle, which caused the stream scheduler to function inefficiently. Using the *Ellpack* format on Imagine, improved the data-parallelism, causing the total number of cycles to reduce (dramatically for the LargeDIS matrix). Note, that since that for both the VIRAM and Imagine Ellpack implementations, the matrices are artificially padded with zeros to create symmetric row lengths, thus the fraction of useful operations can be arbitrarily poor depending on the matrix structure. However, this fraction of useful computations would penalize the effective performance of both architectures equally.

V. Transitive Closure

Computing the *Transitive Closure* of a directed graph is an important problem that arises in many applications, including network routing and distributed computing. The classical sequential approach for solving this problem uses the dynamic programming methodology of the Floyd-Warshall algorithm [9]. Although a titled approach is necessary for efficient data reuse, blocking this algorithm is nontrivial due to the complex data dependency requirements. We examine four problem sizes from the DIS specification, consisting of 64, 128, 256 and 512 vertices.

A. Implementation Details

The VIRAM version of *Transitive Closure* is taken from the DIS reference implementation. [10] and uses a dense matrix to represent the distance graph. A small code modification allowed data access in unit-stride fashion, and thus significantly improved performance.

The DIVA version of *Transitive Closure* is also based on the DIS implementation . Here, the two inner loops of the original main loop nest are interchanged, so that the matrix is accessed with stride one in the innermost loop, and spatial reuse can be exploited in wide registers. The DIVA version exploits fine-grained parallelism by performing arithmetic operations on eight 32-bit elements of the matrix in parallel. In addition to the arithmetic operations, it uses the selective execution mode supported by the wide ALU. A wide operation (*wmrgcc*) merges the contents of two wide registers according to condition-code bits, allowing an efficient computation of the minimum value of each pair of elements of two superword operands.

Imagine's *Transitive Closure* implementation is significantly different than the VIRAM or DIVA approach due to the relatively high overhead of off-chip memory transfers. It was therefore necessary to implement a tiled version of the algorithm to minimize data transfer overhead. However,

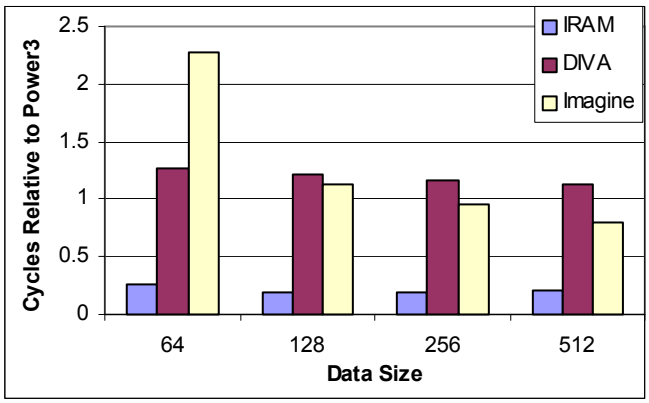

Figure 7: Transitive Closure cycles relative to Power3

Transitive Closure presents a very different set of challenges from those present in dense linear algebra problems such as matrix multiply and FFT, due to complex data dependencies and restrictions of the stream programming paradigm. Some of the algorithmic considerations required to effective utilize the Imagine architecture included: converting each computational block to a data stream, effectively reusing the limited number of addressable registers (256 per arithmetic cluster), managing stream register file reuse in the context of complex data dependencies, and minimizing expensive off-chip memory access operations. Our specialized Imagine implementation reuses data streams aggressively and resulted in memory transfers near the theoretical minimum. Details are presented in [14].

The Power3 implementation of *Transitive Closure* also uses a tiled algorithm to maximize cache-reuse. To allow aggressive compiler optimization, all block entries are copied into local variables for efficient use of local registers. Additionally, block computations are hand unrolled to maximize instruction level parallelism. Details are presented in [14].

B. Performance Results

Figure 7 compares *Transitive Closure* performance (in cycles) between the Power3 and the three emerging architectures. VIRAM achieves excellent performance, requiring only 19%-25% of Power3's cycles. These results confirm the expected advantage for VIRAM on a problem with abundant parallelism and low arithmetic/memory operation ratio (per step). Notice that VIRAM is relatively insensitive to graph size, although we would expect larger problem to perform better due to the longer average vector lengths.

DIVA's *Transitive Closure* takes advantage of the available superword-level parallelism and spatial locality in wide registers. However, this simple implementation does not include optimizations for temporal locality in registers, and uses a limited amount of loop unrolling (just enough to expose superword-level parallelism). For this simple implementation the 1-PIM DIVA achieves an average performance of 1.2x with respect to the Power3. Again, the fact that the first PIM chip has no support for hiding memory latencies results on

memory stall times of 57% of the execution time.

The Imagine implementation was by far the most complex, since blocking was required to minimize the volume of off-chip memory transfer. For the small data size (64 vertices), Imagine requires 2.2X more cycles than the Power3. However, for the larger data sets, Imagine's relative performance improves, achieving 80% of the Power3 cycles for 512 vertices. *Figure 8*, shows a breakdown of cycles spent in computation and memory transfers, and helps explain why Imagine's performance is limited for this computationally intensive benchmark. Notice the total percentage exceeds 100% since certain operations overlap. As *Figure 8* shows, only a small fraction of the total cycles (23%-37%) are actually accounted for by the kernel execution. The remaining cycles are mostly consumed for host and SRF data transfers, while a small fraction is necessary for loading the microcode. See [14] for an extensive analysis of *Transitive Closure* on the Imagine architecture.

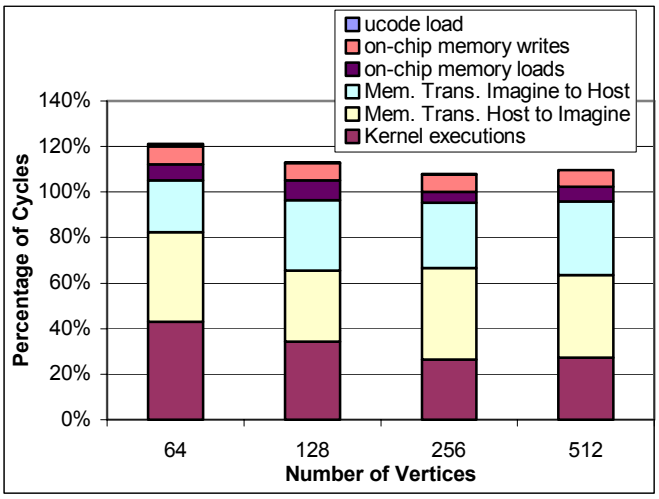

Figure 8: Comparison of Imagine's cycles spent on communication and I/O for Transitive Closure. Totals exceed 100% as certain operations overlap.

VI. Neighborhood

The key kernel of the *Neighborhood* algorithm computes a histogram of a set of integers. Two important considerations govern the algorithmic choices of this benchmark: the number of buckets, *b*, and the likelihood of duplicates. For image processing applications, the number of buckets is large and collisions are common because are typically many occurrences of certain colors (e.g. white) in an image. The possibility of collisions limits parallelism and inhibits an efficient data-parallel implementation. Our experiments examine a 500x500 image from the DIS specification, with pixel depths of 7, 11, and 15.

A. Implementation Details

Different algorithms for computing histograms on vector processors have been proposed in the past [36][1]. However, each of these techniques can exhibit poor worst case time or space performance. For the VIRAM implementation we took advantage of the fact that it is possible to sort a vector register of integers very quickly. This makes use of the *vhalf* instructions that implement "butterfly" permutations. Using these instructions we can efficiently implement Batcher's [4]sort on a vector register. Once this register is sorted it is easy to update the histogram using a diff-find-diff trick [24]. This algorithm has the advantage that its running time is relatively insensitive to the distribution of data values while not using any extra main memory storage.

The Neighborhood algorithm is currently being developed on DIVA and will be available for the final paper.

On Imagine, the histogram in *Neighborhood* is updated in local scratch memory. Because this buffer may be too small to accommodate the entire histogram, it is updated in phases. On each phase a portion of the histogram (representing values between, say, *i* and *i+k*) is loaded into scratch memory. The remaining data array (which hasn't yet updated any portion of the histogram) is streamed through the clusters and those elements corresponding to the current portion of the histogram perform updates. The rest of the data array is saved for the next phase where the next portion of the histogram is updated. This process continues until all the data elements have been exhausted.

The Power3 *Neighborhood* benchmark is implemented directly from the DIS specification, without any additional optimization.

B. Performance Results

Figure 9 presents the relative number of Power3 cycles necessary to compute the Neighborhood using three pixel input depths from the DIS specification, 7-, 11- and 15- bit. For the small bin sizes VIRAM performance is poor, requiring 2.3 and 1.5 times more cycles than the Power3. For these cases, the presence of duplicates in the image data inhibits data-parallelism and thus hurts vectorization performance; however, the short bin depths can actually help improve cache hits on cache-based architectures. We therefore see very high performance on the Power3 without any special optimizations. VIRAM's memory system advantage starts to become apparent for 15-bit pixels, where the histograms do not fit in cache. Here VIRAM shows a 3X improvement in cycles compared to the Power3.

Imagine performance for the 7-bit input is rather poor, requiring 5 times as many cycles as the Power3. This is due to the relatively low computational requirements of the benchmark, which does not allow Imagine to effectively utilize its large set of functional units To date we have been unable to successfully run the Imagine experiments for the larger pixel depths, but expect to have those results complete for the final paper.

VII. Related Work

Over the last few years many techniques have been proposed for addressing the growing processor-memory gap. One area that has received much interest concerns improving the latency tolerance of traditional processors through various degrees of multithreading (such as the Tera MTA and current

processors with SMT). Traditional processors can also be enhanced with more intelligent memory controllers, improving performance for strided and indexed accesses [41]. The idea of adding more powerful data parallel units to contemporary microprocessors [12] has also been explored. There have also been attempts to use embedded processors (for their low power) in a high performance computing context [7]. PIM technology is also explored in [17] and [33].

The architectures in this paper have also been the subject of other benchmarking activities. The performance IRAM, DIVA, and Imagine on signal processing applications is discussed in [37]. IRAM is compared to cache-based processors for scientific and multimedia workloads in [13] and [21].

Many other benchmarks appear in the literature for processor-memory system characterization. The STREAMS benchmark [35] is a standard way of measuring memory bandwidths for copy and simple vector operations. Probes for determining memory system parameters (such as cache sizes and various latencies) are introduced in [32]. For determining the maximum performance of high performance architectures under "real-world" conditions the Linpack benchmark is traditionally employed. Finally, the Livermore Loops [27]are often used to determine how vector systems handle algorithms with differing types of data dependencies.

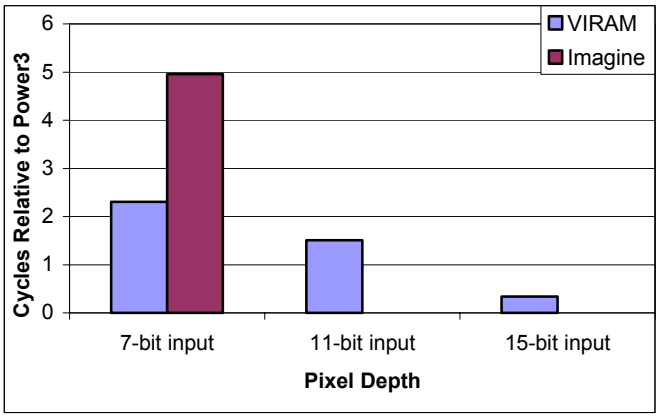

Figure 9: Neighborhood Performance

VIII. Conclusions and Future Work

This paper examines three emerging microprocessor technologies designed to address the processor-memory gap through the use of explicit data parallelism. Our first contribution is the development an adaptable probe, *Sqmat*, which allowed us to evaluate key architectural features. By varying a small set of parameters, we explored performance sensitivity to computational intensity, vector/stream length, memory access patterns, and kernel overheads. *Sqmat* allowed us to gain insight into the balance of the architectures and quantify the computational space best suited for each processing paradigm. Work is currently underway to expand *Sqmat's* functionality and evaluate the architectural balance of leading microprocessor designs.

Next we examined three important computational kernels, each with a different balance computational intensity, memory access patterns, and available data-parallelism. The *SPMV* kernel requires random data access and a low number of arithmetic operations. *Transitive Closure* is a dense code, and can be block to provide a high ration of operations per memory access; however, the tiled approach requires adherence to complex data dependencies. Finally we presented *Neighborhood* whose random data access patterns and potential data collisions are at odds with data parallel programming.

Our benchmark set allowed us to explore several critical components of the underlying data-parallel architectures: staging data to the functional units, overhead of irregular data access, penalty of algorithmic data dependencies, programming complexity and overall performance. Both VIRAM and DIVA have used PIM technology allowing for low-latency and high-bandwidth memory access. This is significantly less expensive than Imagine's off-chip memory access. However, Imagine can utilize a large volume of external memory, whereas VIRAM and DIVA are limited to the small on-chip DRAM before additional programming and performance overheads are incurred.

All three architectures incur a penalty for irregular data access. VIRAM's overhead is the smallest, but suffers a slowdown due to limited number of address generators in comparison to the number of its functional units. DIVA's overhead is relatively high, since extra operations are required to pack data contiguously into superwords. Imagine's stream programming paradigm is restricted to uniform data access; therefore a relatively high penalty must be paid to appropriately reorder irregularly structure data.

Algorithms with data dependencies are inherently at odds with data-parallel architectures. DIVA seems to incur the smallest penalty since each wide register holds only eight 32-bit elements, thus limiting the potential effect of data dependence operations. VIRAM sustains a higher overhead since each vector instruction specifies the execution of 64 elements (or 8 element groups). Imagine has the potential to incur the highest cost, since computations may need to be streamed repeatedly to address complex data dependencies.

Programmability is also an important issue that must be considered for each of the architectures. DIVA presents the simplest migration from scalar programming due to its similarity to conventional microprocessors**.** The VIRAM vector programming paradigm incurs a higher programming complexity but is well understood and can leverage years of algorithmic research as well as sophisticated compiler technologies. Stream program development on Imagine was the most challenging, compared to the better-known vector or SLP approaches. The Imagine programmer is exposed to the memory hierarchy and cluster organization of the underlying architecture, and programming is awkward for irregular applications. Improvement in the quality of the compiler and software development tools, and abstracting lower level details of the hardware, worked currently in progress will be essential in bringing the stream programming model to the

wider scientific community.

Finally, our performance comparisons showed that VIRAM consistently outperformed the superscalar Power3 architecture, with significantly less power consumption. The Imagine architecture's performance was inhibited by the irregularity and complex data dependencies; although it posses tremendous processing potential for properly structured algorithms. Finally DIVA displayed limited performance. This was due to the first DIVA PIM chip, which does not support overlapping computation and memory access. Additionally, our experiments examined a one-PIM DIVA configuration, even though DIVA was designed to contain four nodes per PIM and multiple PIM chips.

Future plans include validating our results on real hardware as it becomes available, as well as examining a broader scope of scientific codes. We plan to evaluate more complex data-parallel systems such as those proposed for the DARPA HPCS initiative. Our long-term goal is to evaluate these technologies as building blocks for future high-performance multiprocessor systems.

ACKNOWLEDGMENT

The authors would like to thank Manikandan Narayanan, Adam Janin for their help with the *Sqmat* benchmark, Xiaoye Li for *SPMV*, the DIVA team at ISI, and the Imagine team most notably Abhishek Das (for the *Neighborhood* implementation on Imagine).

REFERENCES

[1] Y. Abe. Present Status of Computer Simulation at IPP, *Proc. Supercomputing '88*: *vol 2, Science and Applications*, 1988.

[2] S. Andersson, R. Bell, J. Hague, H. Holthoff, P. Mayes, J. Nakano, D. Shieh, and J. Tuccillo. *RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide*, IBM Corporation, 1998.

[3] D.H. Bailey, J. Barton, T. Lasinski, and H.D. Simon (Eds.). The NAS parallel benchmarks. Tech. Rep. RNR-91-002, NASA Ames Research Center, Moffett Field, 1991.

[4] K. Batcher. Sorting networks and their applications. *Proc. AFIPS Spring Joint Compute Conf.*, 1968.

[5] The Berkeley Intelligent RAM (IRAM) Project, Univ. of California, Berkeley, at http://iram.cs.berkeley.edu.

[6] G. Blelloch, M. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Tech. Rep. CMU-CS-93-173, Carnegie Mellon Univ., Pittsburgh, 1993.

[7] The BlueGene/L Team. An Overview of the BlueGene/L Supercomputer. *Proc SC2002*, 2002.

[8] D. Burger, J.R. Goodman, and A. Kagi. Memory Bandwidth Limitations of Future Mircoprocessors. *Proc. ISCA1996*, 1996.

[9] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*, MIT Press, 1990.

[10] DIS Stressmark Suite, v 1.0. Titan Systems Corp., 2000, at http://www.aaec.com/projectweb/dis/

[11] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C.W. Kang, I. Kim, and G. Daglikoca. The Architecture of the DICA Processing-In-Memory Chip. *Proc 2002 Int'l Conference on Supercomputing*, pp.14-25, June 2002.

[12] R. Espasa, F. Ardanaz, J. Gago, R. Gramunt, I. Hernandez, T. Juan, J. Emer, S. Felix, G. Lowney, M. Mattina, and A. Seznec . Tarantula: A Vector Extension to the Alpha Architecture. *Proc. ISCA 2002*, 2002.

[13] B. Gaeke, P. Husbands, X. Li, L.Oliker, K. Yelick, and R. Biswas. Memory Intensive Benchmarks: IRAM vs. Cache-Based Machines. *Proc. 2002 International Parallel and Distributed Processing Symposium,* 2002.

[14] G. Griem. Implementation of Transitive Closure on the Imagine Stream Processor. In Preparation, 2003.

[15] J.L. Hennessy and D.A. Patterson. *Computer Organization and Design*. Morgan Kaufmann, 1997.

[16] The Imagine project, Stanford University, at http://cva.stanford.edu/imagine/.

[17] Y.. Kang, M. Huang, S-M Yoo, Z. Ge, D. Keen, V. Lam, P. Pattnaik, and J. Torellas. FLEXRAM: Toward an Advanced Intelligent Memory System. Proc IEEE International Conference on Computer Design (ICDD), pp.192-201, October 1999.

[18] U. Kapasi, W. Dally, S. Rixner, P. Mattson, J. Owens, and B. Khailany. Efficient Conditional Operations for Data-parallel Architectures *Proc. 33rd Annual International Symposium on Microarchitecture,* Dec. 10-13, 2000.

[19] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang. Imagine: Media Processing with Streams. *IEEE Micro*, Mar/April 2001.

[20] D. Kincaid, T. Oppe, and D. Young. ITPACKV 2D user's guide. Tech. Rep. CAN-232, Univ. of Texas, Austin, 1989.

[21] C. Kozyrakis. A media-enhanced vector architecture for embedded memory systems. Tech. Rep. UCB-CSD-99-1059, Univ. of California, Berkeley, 1999.

[22] C. Kozyrakis, D. Judd, J. Gebis, S. Williams, D. Patterson, and K. Yelick. Hardware/compiler co-development for an embedded media processor. *Proceedings of the IEEE*, 2001.

[23] S. Larsen and S. Amarasinghe. Exploiting Superword-Level Parallelism with Multimedia Instruction Sets. *Pro. 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI*), Vancouver, British Columbia, Canada, July 18-21, 2002.

[24] The MathWorks. How do I vectorize my code? Tech. Note 1109, at http://www.mathworks.com.

[25] P. Mattson. Programming System for the Imagine Media Processor. Ph.D. Thesis, Stanford University, 2002.

[26] Maximizing CRAY T90/J90 Applications Performance - vectorization of C code. *Scientific Computing at NPACI (SCAN)*, 3 (15), July 21, 1999.

[27] F.H. McMahon. LLNL Fortran Kernels. Technical Report, Lawrence Livermore National Laboratory, 1984.

[28] J. Owens, S. Rixner, U. Kapasi, P. Mattson, B. Towles, B. Serebrin, and W. Dally. Media Processing Applications on the Imagine Stream Processor. *Proc. 2002 International Conference on Computer Design*., 2002.

[29] J-S Park, M. Penner, and V. K. Prasanna. Optimizing Graph Algorithms for Improved Cache Performance. *Proc. 2002 International Parallel and Distributed Processing Symposium,* 2002.

[30] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, R. Thomas, C. Kozyrakis, and K. Yelick. Intelligent RAM (IRAM): Chips that remember and compute. *Proc. Intl. Solid-State Circuits Conf.*, 1997.

[31] A. Peleg, S. Wilkie, and U. Weiser. Intel MMX for multimedia PCs. *Communications of the ACM*, 40(1):pp. 24-38, January 1997.

[32] R. H. Saavedra-Barrera. CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking. PhD thesis, University of California, Berkeley, February 1992.

[33] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. Proc ISCA 1996,pp.90-101, May 1996.

[34] J. Shin, J. Chame and M. Hall. Exploiting Superword-Level Locality in Multimedia Extension Architectures. *Journal of Instruction Level Parallelism* (*JILP*), vol. 5(2003), pp. 1-28.

[35] STREAM:Sustainable Memory Bandwidth in High Performance Computers, at http://www.cs.virginia.edu/stream/

[36] K. Suehiro, H. Murai, and Y. Seo. Integer Sorting on Shared-Memory Vector Parallel Computers. *Proc. ICS 98*, 1998.

[37] J. Suh, E-G Kim, S. P. Crago, L. Srinivasan, and M.C. French. A Performance Analysis of PIM, Stream Processing, and Tiled Processing on Memory-Intensive Signal Processing Kernels. *Proc. ISCA2003*, 2003.

[38] W. Thies, M. Karczmarek and S. Amarasinghe. StreamIt: A Language for Streaming Applications. *Computational Complexity*, pp. 179-196, 2002

[39] TOP500 Supercomputer Sites at http://www.top500.org

[40] UPC Language Specification, Version 1.0, at http://upc.gwu.edu

[41] L. Zhang, Z. Fang, M. Parker, B.K. Mathew, L. Schaelicke, J.B. Carter, W.C. Hsieh, and S.A. McKee. The Impulse Memory Controller. *IEEE Transactions on Computers, Special Issue on Advances in High Performance Memory Systems*, pp. 1117-1132, November 2.