Merrimac: Supercomputing with Streams

William J. Dally, Patrick Hanrahan, Mattan Erez, Timothy J. Knight, François Labonté, Jung-Ho Ahn, Nuwan Jayasena, Ujval J. Kapasi, Abhishek Das, Jayanth Gummaraju, Ian Buck

ABSTRACT

Merrimac uses stream architecture and advanced interconnection networks to give an order of magnitude more performance per unit cost than cluster-based scientific computers built from the same technology. Organizing the computation into streams and exploiting the resulting locality using a register hierarchy enables a stream architecture to reduce the memory bandwidth required by representative applications by an order of magnitude or more. Hence a processing node with a fixed bandwidth (expensive) can support an order of magnitude more arithmetic units (inexpensive). This in turn allows a given level of performance to be achieved with fewer nodes (a 1-PFLOPS machine, for example, with just 8,192 nodes) resulting in greater reliability, and simpler system management. We sketch the design of Merrimac, a streaming scientific computer that can be scaled from a $20K 2 TFLOPS workstation to a $20M 2 PFLOPS supercomputer and present the results of some initial application experiments on this architecture.

1. Introduction

Modern semiconductor technology makes arithmetic inexpensive and bandwidth expensive. To exploit this shift in cost, a high-performance computer system must exploit locality, to raise the *arithmetic intensity* (the ratio of arithmetic to bandwidth) of the application as well as parallelism to keep a large number of arithmetic units busy. Expressing an application as a *stream program* fulfills both of these requirements. It exposes large amounts of parallelism across stream elements and reduces global bandwidth by expressing locality within and between kernels.

A stream processor exploits the parallelism exposed by a stream program, by providing 100s of arithmetic units, and exploits the locality of a stream program, by providing a deep register hierarchy. In particular, memory bandwidth is reduced by capturing short-term producer-consumer locality in large *local register files*, and long-term producer-consumer locality in a *stream register file*. This locality might not be captured by a reactive cache. More importantly, the stream register file is *aligned* with individual ALUs and requires only local on-chip communication while a cache requires global on-chip communication.

We are designing Merrimac[1], a scientific computer system tailored to exploit the parallelism and locality of streams. The core of Merrimac is a single-chip (90nm CMOS) stream processor that is expected to have 128 GFLOPS peak performance. This processor chip along with 16 high-bandwidth DRAM chips (2G Bytes of memory) form a single Merrimac node. Application experiments suggest that this single-node Merrimac will sustain up to half of peak performance on a range of scientific applications. With an estimated parts cost of less than $1K per 128 GFLOPS node (including network), we expect a Merrimac machine to provide both capability and capacity — being more cost effective than machines based on commodity microprocessors.

Merrimac employs a *high-radix* interconnection network to connect 16 nodes (2 TFLOPS) on a single board, 512 nodes (64 TFLOPS) in a cabinet, and 8K nodes (1 PFLOPS) in 16 cabinets. The network provides a flat shared address space across the multi-cabinet system with flat bandwidth across a board (16 nodes) and a global bandwidth of 1/8 the local bandwidth anywhere in the system.

We have coded three representative scientific applications as stream programs and measured their performance on a simulated Merrimac node. These initial experiments show that typical scientific applications cast as stream programs maintain a high arithmetic to memory bandwidth ratio and achieve a high fraction of peak performance. The applications simulated have computation-to-memory ratios in the range of 7:1 to 50:1, achieving between 18% and 52% of the peak performance of the machine, with less than 1.5% of data references traveling off-chip.

The remainder of this paper describes stream processors and the Merrimac project in more detail. In Section 2 we see that modern VLSI technology makes arithmetic cheap and bandwidth expensive. Section 3 shows how a stream processor exploits the application locality using a bandwidth hierarchy and application parallelism by using large numbers of ALUs. Merrimac, a supercomputer based on streams, is described in Section 4. We show the performance of a simulated stream processor on a number of applications in Section 5. Issues related to scientific computing with streams are discussed in Section 6

This work was supported in part by the Department of Energy, NNSA, under the ASCI Alliances program (contract LLL-B341491), in part by National Science Foundation Fellowships, in part by Stanford Graduate Fellowships, in part by the DARPA Smart Memories Project (contract MDA904-98-R-S855), and in part by the DARPA Polymorphous Computing Architectures Project (contract F29601-00-2-0085).

SC'03, November 15-21, 2003, Phoenix, Arizona, USA

[1]Merrimac is a Native American word meaning "fast moving stream".

2. VLSI enables inexpensive arithmetic making bandwidth the limiting factor

Modern VLSI fabrication processes make it very inexpensive in terms of both area and power to put large amounts of arithmetic capability on a chip. With arithmetic almost free, global bandwidth, both on-chip and off-chip, becomes the factor limiting performance.

In $0.13\mu m$ CMOS technology, a 64-bit floating-point unit (FPU) (multiplier and adder) has an area of less than 1mm^2 and dissipates about 50pJ of energy per operation [1]. Over 200 such FPUs can fit on a 14mm $\times$ 14mm chip that can be manufactured in volume (including testing and packaging) for less than $100. Even at a conservative operating frequency of 500MHz this gives a cost of 64-bit floating-point arithmetic of less than $1 per GFLOPS and a power of less than 50mW per GFLOPS. Even though one cannot completely fill a chip with FPUs, modern graphics chips come close to realizing these cost performance levels. For example, the nVidia NV30 sustains 100 GFLOPS (32-bit floating point) [2].

The already low cost of arithmetic is decreasing rapidly as technology improves. We describe a CMOS technology by its drawn gate length L. Most chips today are manufactured with $L = 0.13\mu m$. Historical trends show that L decreases at about 14% per year [3]. The cost of a GFLOPS of arithmetic scales as L^3 and hence decreases at a rate of about 35% per year [4]. Every five years, L is halved, four times as many FPUs fit on a chip of a given area, and they operate twice as fast — giving a total of eight times the performance for the same cost. Of equal importance, the switching energy also scales as L^3 so every five years, we get eight times the arithmetic performance for the same power.

Global bandwidth, not arithmetic is the factor limiting the performance and dominating the power of modern processors. The cost of bandwidth grows at least linearly with distance in terms of both availability and power [4]. To keep distances constant across technology generations, we express distance in units of *tracks*. One track (or 1χ) is the distance between two minimum width wires on a chip. In $0.13\mu m$ technology, $1\chi \approx 0.5\mu m$. We can put ten times as many $10^3\chi$ wires on a chip as we can $10^4\chi$ wires. More importantly, moving a bit of information over a $10^3\chi$ wire takes only $1/10^{\text{th}}$ the energy as moving a bit over a $10^4\chi$ wire. In an $0.13\mu m$ technology, for example, transporting the three 64-bit operands for a 50pJ floating point operation over global $3 \times 10^4\chi$ wires consumes about 1nJ, 20 times the energy required to do the operation. In contrast, transporting these operands on local wires with an average length of $3 \times 10^2\chi$ takes only 10pJ, much less than the cost of the operation.

Contemporary architectures are not yet tuned to these developing VLSI constraints. These architectures are unable to use more than a few arithmetic units because they are designed for applications with limited parallelism and are hindered by a low bandwidth memory system. Their main goal is to provide high performance for mostly serial code that is highly sensitive to memory latency and not bandwidth. To exploit the capabilities of today's VLSI technology requires an architecture that can exploit parallelism — to keep large numbers or arithmetic units busy while hiding the ever increasing latency to memory, and locality — to increase the ratio of arithmetic, which is inexpensive, to global bandwidth, which is the limiting factor.

3. Stream Architecture exploits the characteristics of VLSI

A *Stream Processor* is able to take advantage of the large number of arithmetic units that VLSI technology enables without exceeding the bandwidth limitations of the technology by using a *register hierarchy* to exploit locality in the application. This greatly reduces the average distance an operand must travel to reach a FPU. As shown in Figure 1, a stream architecture consists of an array of clusters, each with a set of FPUs, a set of *local register files* (LRFs), and a bank of a *stream register file* (SRF). Each FPU in a cluster reads its operands out of an adjacent LRF over very short, ($\approx 100\chi$), wires. FPU results are distributed to the other LRFs in a cluster and accesses to the local SRF bank are made via the cluster switch over short ($\approx 1,000\chi$) wires. While the SRF is similar in size to a cache, SRF accesses are much less expensive than cache accesses because they are aligned and do not require a tag lookup. Each cluster accesses its own bank of the SRF over short wires. In contrast, accessing a cache requires a global communication over long ($\approx 10,000\chi$) wires. The SRF also plays another crucial role in keeping the arithmetic units busy by allowing the software to hide long memory latencies. An entire stream is transferred between the SRF and the memory with a single instruction. These stream memory operations generate a large number of memory references to fill the very deep pipeline between processor and memory, allowing memory bandwidth to be maintained in the presence of latency. Arithmetic units are kept busy by overlapping the execution of arithmetic kernels with these stream memory operations.

To see how a stream processor exploits locality, consider a simple application expressed as a stream program (Figure 2). This figure shows a synthetic application that is designed to have the same bandwidth demands as the StreamFEM application (Section 5). Each iteration, the application streams a set of 5-word grid cells into a series of four kernels. The kernels operate on the data, performing the number of operations indicated, and pass intermediate results on to the next kernel. To perform a table lookup, kernel K1 generates an index stream that is used to reference a table in memory generating a 3-word per element stream into kernel K3.

Figure 3 shows how the stream program of Figure 2 maps to the register hierarchy of a stream processor. The grid cells start in memory and are read a *strip* at a time into a buffer in the SRF. A typical strip might be 1024 5-word records.[2] Once a strip of cells is in the SRF, kernel K1 is run generating a strip of indices and a strip of intermediate results in the SRF. Kernel K2 is run on the results, generating a second set of intermediate results while the indices are applied to memory to read a strip of table values into the SRF. Table values that are repeatedly accessed are provided by the cache. The process continues until the updates to the strip of grid cells, generated by kernel K4, are written back to memory. Each strip is software pipelined so that the loading of one strip of cells is overlapped with the execution of the four kernels on the previous strip of cells and the storing of the strip before that.

This synthetic application shows how the stream architecture exploits locality. In Section 5 we shall see that actual applications exploit locality in a similar manner. Kernels K1...K4 perform all of their 300 operations out of LRFs, performing 900 LRF accesses per grid point. The streams between the kernels are passed through the SRF generating 58 words of SRF bandwidth per grid point. Finally memory accesses total 12 words. This gives us a bandwidth ratio of 75:5:1, 75 LRF references and 5 SRF references for every mem-

[2]The strip size is chosen by the compiler to use the entire SRF without any spilling.

1-58113-695-1/03 $ 17.00 © 2003 ACM

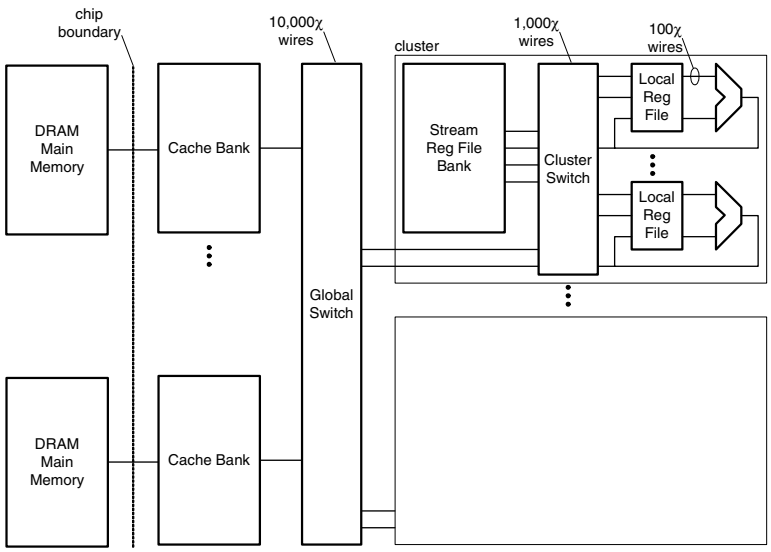

Figure 1: A stream processor consists of an array of clusters each having a number of functional units with local register files and a stream register file bank connected by a cluster switch. The clusters are connected to each other and to cache banks by a global switch. At each level of this hierarchy — local register, intra-cluster, and inter-cluster — the wires get an order of magnitude longer.

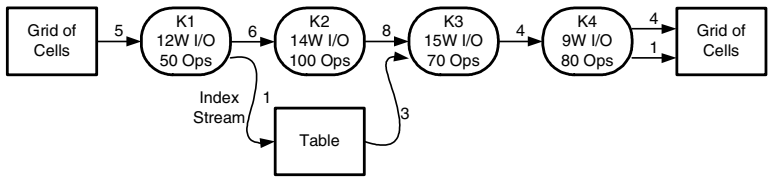

Figure 2: A synthetic stream application, modeled after StreamFEM (Section 5) consists of a set of *kernels* K1 . . . K4 that pass *streams* of data between them.

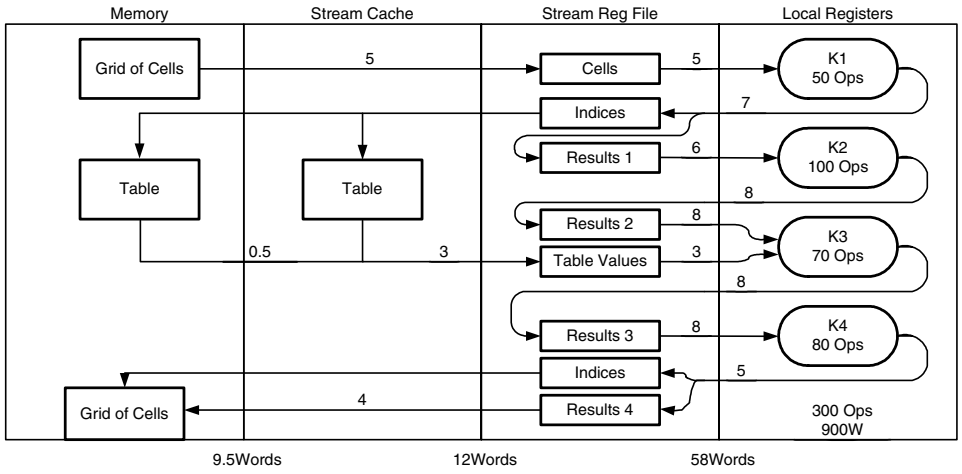

Figure 3: The stream program of Figure 2 is mapped to the bandwidth hierarchy of a stream processor.

ory reference. Put differently, 93% of all references are made from the LRFs, where bandwidth is very inexpensive, and only 1.2% of references are made from the memory system, where bandwidth is expensive for cache hits and very expensive for misses.[3]

A stream processor executes a stream instruction set. This instruction set includes scalar instructions, that are executed on a conventional scalar processor, stream execution instructions, that each trigger the execution of a kernel on one or more strips in the SRF, and stream memory instructions that load and store (possibly with gather and scatter) a stream of records from memory to the SRF. This stream instruction set closely follows that of the Imagine streaming media processor [5, 6].

Merrimac also provides hardware support for a *scatter-add* instruction. This instruction is an example of a new architectural feature that is enabled by programming in streams. A scatter-add acts as a regular scatter, but adds each value to the data already at each specified memory address rather than simply overwriting the data. This type of operation was discussed from a parallel algorithm perspective in [7].

4. Sketch of Merrimac: a Streaming Scientific Computer

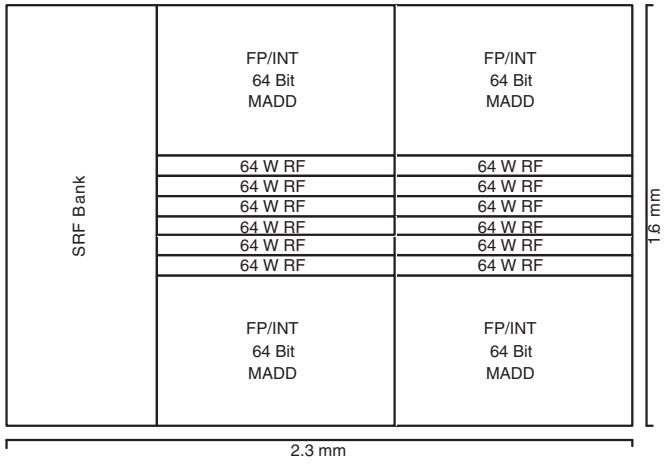

Figure 4: Floorplan of a Merrimac cluster.

Each Merrimac node contains a stream processor (as illustrated in Figure 1) with 16 arithmetic clusters. Each cluster contains four floating-point multiply-add (MADD) units, 768 64-bit words of local registers, and 8K words of stream register file. The entire stream register file has a capacity of 128K 64-bit words, distributed across the 16 clusters. A floorplan for one cluster is shown in Figure 4. Each MADD unit measures 0.9mm $\times$ 0.6mm and the entire cluster measures 2.3mm $\times$ 1.6mm. We conservatively plan to operate with a clock cycle of 1ns (37 FO4 inverters in 90nm[3]) giving a performance of 8 GFLOPS per cluster and 128 GFLOPS across the 16 clusters.

A floorplan of the entire Merrimac stream processor chip is shown in Figure 5. The bulk of the chip is occupied by the 16 clusters. The

[3]Many of our applications have very large kernels that in effect combine several smaller kernels — passing intermediate results through LRFs rather than SRFs. While this increases the fraction of LRF accesses, it also stresses LRF capacity. Ideally, the compiler will partition large kernels and combine small kernels to balance these two effects. We have not yet implemented this optimization.

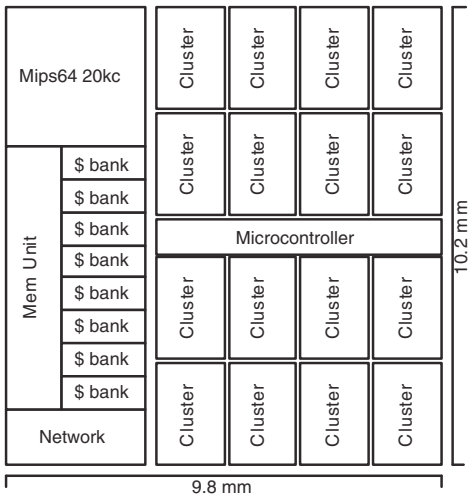

Figure 5: Floorplan of a Merrimac stream processor chip.

left edge of the chip holds the remainder of the node. A scalar processor [8] fetches all instructions, executes the scalar instructions itself, and dispatches stream execution instructions to the clusters (under control of the microcontroller) and stream memory instructions to the memory system. The node memory system consists of a set of address generators (not shown), a line-interleaved eight-bank 64K-word (512KByte) cache, and interfaces for 16 external DRAM chips. A network interface directs off-node memory references to the routers. We estimate that each Merrimac processor will cost about $200 to manufacture and will dissipate a maximum of 31W of power. Area and power estimates in a standard cell process in 90nm technology are derived from models based on a previous implementation of stream processor [1].

Figure 6 illustrates a single Merrimac board containing 16 nodes — 16 128GFLOPS stream processors (Figure 5) each with 2 GBytes of DRAM — and four router chips. The router chips interconnect the 16 processors on the board, providing flat memory bandwidth on board of 20 GBytes/s per node. The routers also provide a gateway to the inter-board network, with a 4:1 reduction in memory bandwidth (to 5 GBytes/s per node), for inter-board references.

Larger Merrimac systems are interconnected by a five-stage folded-Clos [9] network[4] using high-radix routers as illustrated in Figure 7. The routers on each 16-node board serve as the first and last stage of this network. The basic building block of this network is a 48-input $\times$ 48-output router chip. Each bidirectional router channel (one input and one output) has a bandwidth of 2.5 GBytes/s (four 5Gb/s differential signals) in each direction. On each 16-processor board, each of four routers has two 2.5 GByte/s channels to/from each of the 16 processor chips and eight ports to/from the backplane switch. The remaining eight ports are unused. Thus each node provides a total of 32 channels to the backplane. At the backplane level, 32 routers connect one channel to each of the 32 boards and connect 16 channels to the system-level switch. A total of 512 2.5 GByte/s channels traverse optical links to the system-level switch where 512 routers connect all 48 ports to up to 48 backplanes (the figure shows just 32 backplanes).

Table 1 shows the estimated cost of a streaming supercomputer. The processor and router chip are modest-sized (10mm $\times$ 11mm) ASICs in 1000-pin flip-chip BGA packages that are expected to

[4]This topology is sometimes called a Fat Tree [10].

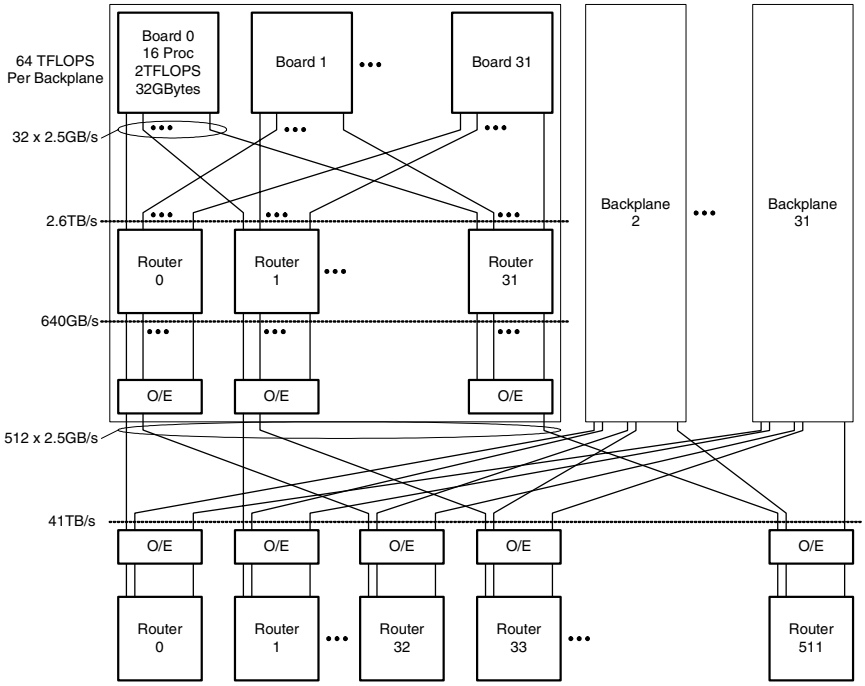

Figure 7: A 2 PFLOPS Merrimac system uses a high-radix interconnection network.

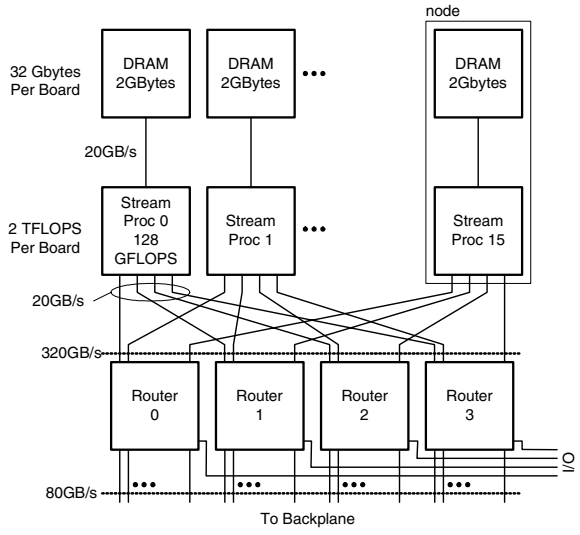

Figure 6: Sixteen 128 GFLOPS stream processors each with 2 GBytes of DRAM memory can be packaged on a single board. The board has a total of 2 TFLOPS of arithmetic and 32 GBytes of memory. Such a board is useful as a stand-alone scientific computer and as a building-block for larger systems.

Item	Cost($)	Per Node Cost ($)
Processor Chip	200	200
Router Chip	200	69
Memory Chip	20	320
Board	1000	63
Router Board	1000	2
Backplane	5000	10
Global Router Board	5000	5
Power	1	50
Per Node Cost		718
$/GFLOPS (128/Node)		6
$/M-GUPS (250/Node)		3

Table 1: Rough Per-Node Budget. Parts cost only, does not include I/O.

cost $200 each in moderate quantities (1000s). DRAM chips are projected to cost $20 each, making DRAM, at $320 the largest single cost item. Board and backplane costs, including connectors, capacitors, regulators, and other components is amortized over the 16 nodes on each board and the 512 nodes in each backplane. The router board and global router board costs reflect the costs of the intra-cabinet and inter-cabinet networks respectively. Supplying and removing power costs about $1 per W or about $50 per 50W node. Overall cost is less than $1K per node, which translates into $6 per GFLOP of peak performance and $3 per M-GUPS[5].

[5]GUPS or *global updates per second* is a measure of global unstructured memory bandwidth. It is the number of single-word read-modify-write operations a machine can perform to memory locations randomly selected from over the entire address space.

Application	Sustained GFLOPS	FP Ops / Mem Ref	LRF Refs	SRF Refs	Mem. Refs
StreamFEM (Euler, quadratic)	32.2	23.5	169.5M (93.6%)	10.3M (5.7%)	1.4M (0.7%)
StreamFEM (MHD, cubic)	33.5	50.6	733.3M (94.0%)	43.8M (5.6%)	3.2M (0.4%)
StreamMD	14.2	12.1	90.2M (97.5%)	1.6M (1.7%)	0.7M (0.8%)
StreamFLO	11.4	7.4	234.3M (95.7%)	7.2M (2.9%)	3.4M (1.4%)

Table 2: Performance measurements of streaming scientific applications

5. Applications exploit the locality of a stream processor

Three scientific applications were used to evaluate the single node performance of the Merrimac stream processor: StreamFEM, StreamMD, and StreamFLO. These applications feature a number of characteristics which are common in scientific applications in general, including regular and irregular multidimensional meshes, multigrid techniques, and particle-in-cell computations.

StreamFEM is a finite element application designed to solve systems of first-order conservation laws on general unstructured meshes. The StreamFEM implementation has the capability of solving systems of 2D conservation laws corresponding to scalar transport, compressible gas dynamics, and magnetohydrodynamics (MHD) using element approximation spaces ranging from piecewise constant to piecewise cubic polynomials. StreamFEM uses the discontinuous Galerkin (DG) method developed by Reed and Hill [11] and later popularized by Cockburn, Hou and Shu [12]. In the present StreamFEM implementation, the limiting procedure of Cockburn et al. has been replaced by variational discontinuity capturing terms as discussed in Jaffre, Johnson and Szepessy [13] with further overall algorithmic simplifications as discussed in Barth [14].

StreamMD is a molecular dynamics solver [15, 16] that is based on solving Newton's equations of motion. The velocity Verlet method (or Leap-frog) is used to integrate the equations of motion in time; using this method, it is possible to simulate the complex trajectories of atoms and molecules for very long periods of time. The present StreamMD implementation simulates a box of water molecules, with the potential energy function defined as the sum of two terms: electrostatic potential and the Van der Waals potential. A cutoff is applied so that all particles which are at a distance greater than r_{cutoff} do not interact. A 3D gridding structure is used to accelerate the determination of which particles are close enough to interact – each grid cell contains a list of the particles within that cell, and each timestep particles may move between grid cells. StreamMD makes use of the scatter-add functionality of Merrimac by computing the pairwise particle forces in parallel and accumulating the forces on each particle by scattering them to memory.

StreamFLO [17] is a finite volume 2D Euler solver that uses a non-linear multigrid algorithm. It is based on the FLO82 code [18][19], which influenced many industrial and research codes. The choice of the code is motivated by the need for an application that is representative of a typical computational fluid dynamics application, without unnecessary complexity. A cell-centered finite-volume formulation is used to solve the fluid equations together with multigrid acceleration. Time integration is performed using a five stage Runge-Kutta scheme.

Table 2 presents measurements from running these three applications on a cycle-accurate simulator of one Merrimac node. These simulations were run on a version of the simulator that included four 2-input multiply/add units per cluster (for a peak performance of 64GFLOPS/node) rather than the four integrated 3-input MADD units (128GFLOPS/node) that is the current design.

The Sustained GFLOPS and FP Ops / Mem Ref columns illustrate the arithmetic intensity of the applications; they are able to sustain from 18% to 52% of the node's peak arithmetic performance, by performing from 7 to 50 floating point operations for each global memory access. Note that only "real" ops are counted in this figure, such as floating point add/mul/compare instructions, and not non-arithmetic ops such as branches. Divides are counted as single floating point operations, even though each divide requires several multiplication and addition operations when executed on the hardware. This leads to the lower performance numbers for StreamMD and StreamFLO – for example, the sustained performance of StreamFLO would double if we counted all the multiplies and adds required for divisions as well.

The right-most three columns list the respective numbers of LRF, SRF, and memory references made by the program, along with the percentage of references satisfied by each level. Note that only a small fraction of references, usually less than 1%, require communication over global ($> 10,000\chi$ or off-chip) wires, and that over 95% of all data movement is on local (100χ) wires (at the LRF level). The register hierarchy of a stream processor exposes costly global communication and allows the locality inherent in applications to be exploited to keep communications local.

Exploiting locality using a register hierarchy increases performance and reduces power dissipation. By performing less data movement per arithmetic operation, we can support a much larger number of arithmetic units before saturating the limited global bandwidth. At the same time power per operation is dramatically reduced by eliminating much of the global communication that dominates power.

6. Discussion

6.1 Streams vs Vectors

Stream processors share with vector processors, like the Cray1 through Cray C90 [20][21], the ability to hide latency, amortize instruction overhead, and expose data parallelism by operating on large aggregates of data. In a similar manner, a stream processor, such as Merrimac, hides memory latency by fetching a stream of records with a single stream load instruction. A kernel is performed on one or more streams of records in the stream register file (SRF) with a single operate instruction. This both amortizes the overhead of the operate instruction and exposes data parallelism.

Stream processors extend the capabilities of vector processors by adding a layer to the register hierarchy, and adding a layer of instruction sequencing that enables them to operate in record (rather than operation) order. The functions of the vector register file (VRF) of a vector processor is split between the local register files (LRFs)

and the stream register file (SRF) of a stream processor. The LRFs stage data between ALU operations to exploit fine-grained producer-consumer locality (sometimes called *kernel* locality). To support a large number of ALUs, they have a very high aggregate bandwidth. Because they exploit only kernel locality, their capacity can be modest, a few thousand words - about the same size as a modern VRF. The stream register file (SRF) of a stream processor stages data to and from memory and stages data to and from the LRFs to exploit coarse-grained (sometimes called outer-loop) producer-consumer locality. Because it is relieved of the task of forwarding data to/from the ALUs, its bandwidth is modest (an order of magnitude less than the LRFs) which makes it economical to build SRFs large enough to exploit coarse-grained locality.

6.2 Balance

The ratios between arithmetic rate, memory bandwidth, and memory capacity on Merrimac are balanced based on cost and utility — so that the last dollar spent on each returns the same incremental improvement in performance. This balancing by diminishing returns gives ratios quite different from the common approach of fixing the ratio of GFLOPS to GBytes irrespective of cost. If we took this approach with Merrimac, we would have to provide 128 GBytes of memory (costing about $20K) for each $200 processor chip making our processor to memory cost ratio 1:100. If one needs 128 GBytes of memory, it is more efficient to provide 64 nodes, even if the additional processors are not required — their cost is small compared to the memory.

A similar argument applies to the ratio of arithmetic to memory bandwidth. Merrimac provides only 20 GBytes/s (2.5 GWords/s) of memory bandwidth for 128 GFLOPS, a FLOP/Word ratio of over 50:1. Many vector machines have FLOP/Word ratios of 1:1 [21], and conventional microprocessors have ratios between 4:1 and 12:1 [22][23]. Providing even a 10:1 ratio on Merrimac would be prohibitively expensive. We would need 80 external DRAMs rather than 16. Interfacing to this large number of DRAMs would require at least 5 external memory interface chips (pin expanders). As with memory capacity, taking this fixed-balance approach to memory bandwidth causes the cost of bandwidth to dominate the cost of processing. Its more efficient to just use Merrimac processor chips to directly interface to 16 DRAMs each. For memory bandwidth dominated computations (e.g., sparse vector-matrix product) most of the arithmetic will be idle. However, even for such computations the Merrimac approach is more cost effective than trying to provide a much larger memory bandwidth for a single node.

6.3 High-Radix Routers

In the 1980s and early 90s, when routers had pin bandwidth in the range of 1-10Gb/s, torus networks gave high throughput while balancing serialization latency against network diameter. For this reason, torus networks were quite popular during this period [24, 25, 26]. Today, with router chip pin bandwidths between 100Gb/s and 1Tb/s possible, a torus can no longer make effective use of this bandwidth. A topology with a higher node degree (or radix) is required. When used in conjunction with *channel slicing*, slicing each node's 20GB/s of network bandwidth across eight 2.5GB/s channels, building routers with high degree (48 for Merrimac) enables a network with very low diameter (2 hops to 16 nodes, 4 hops to 512 nodes, and 6 hops to 24K nodes) compared to a 3-D torus (with a node degree of 6).[6] The use of a Clos network has the added advantage that its hierarchical nature facilitates the use of optical links to cover the long distances required at the top level [27].

[6] If we employed a butterfly rather than a Clos topology these diameters would be nearly halved. Unfortunately a butterfly network is not practical because of its poor performance routing certain permutations.

7. Conclusion

Modern VLSI technology makes arithmetic very cheap (100s of 64-bit FPUs per chip) and bandwidth very expensive (a few words/cycle of off-chip bandwidth). Expressing an application as a stream program exposes parallelism — to take advantage of the large number of arithmetic units and to hide the ever increasing memory latencies — and locality — to reduce the demand on the limited bandwidth. A stream processor exploits this parallelism and locality by providing a deep bandwidth hierarchy that exposes communication so it can be optimized by a compiler. By capturing short-term producer-consumer locality in local registers and long-term producer-consumer locality in a stream register file, a stream processor significantly reduces an application's demand on memory bandwidth.

Merrimac is a stream processor tailored for scientific applications. Merrimac is scalable from a 2 TFLOPS single-board workstation to a 2PFLOPS supercomputer. A 90nm CMOS stream processor chip with a peak performance of 128 GFLOPS enables Merrimac to sustain a high ratio of arithmetic operations to external bandwidth. This allows Merrimac to achieve an efficiency of 128 MFLOPS/$ peak and 23-64 MFLOPS/$ sustained on our pilot applications[7]. A high-radix network gives Merrimac a flat global address space with only an 8:1 (local:global) bandwidth ratio. This gives Merrimac a memory efficiency of 250 K-GUPS/$. This relatively flat global memory bandwidth simplifies programming by reducing the importance of partitioning and placement.

Three representative scientific applications have been converted to stream programs, compiled for Merrimac, and executed on a cycle-accurate simulation of a Merrimac node. These applications all exhibit high locality, maintaining arithmetic to memory access ratios from 7 to 50. Across these applications, over 96% of all data accesses are from local register files and less than 1.5% are to memory. These experiments verify that streams can extract sufficient locality from representative scientific codes to sustain high arithmetic rates with limited memory bandwidth.

The results we present here establish the feasibility of using stream processing for scientific computing — by showing that stream locality exists in representative scientific codes — and suggests that a stream processor can significantly improve the performance per unit cost of scientific computing.

Scientific stream processing raises many interesting questions for future research. Our initial experiments used programs that were manually restructured into stream programs. The development of compilation methods to automate this process of partitioning a vectorized or parallelized code into kernels would make it easier to apply stream processing to enhance the locality of a large volume of existing code. We are also interested in compilation methods that perform transformations on stream programs, splitting and merging kernels to balance register use, and rescheduling kernels and memory operations to most efficiently stage data through the stream register file.

On the architecture front, we are exploring alternative stream register file organizations that appear to offer even greater reductions in required memory bandwidth. We are investigating how to best use a cache in combination with a stream register file and how to give the compiler more control over caching policies. We are also investigating global communication and synchronization

[7] Projected from the experiments of Section 5.

mechanisms that are suitable for use with streams. This includes our scatter-add operation, which reduces the need for synchronization in many applications.

Finally our initial experiments used relativley simple 2D codes running on a single node of a simulated machine. We are currently exploring the properties of larger and more complex 3D codes running across multiple nodes of a simulated machine. Initial indications are positive — that these codes exhibit at least as much 'stream' locality as their simpler counterparts.

8. Acknowledgements

We would like to thank Massimiliano Fatica and Eric Darve from the Mechanical Engineering Department at Stanford, as well as Timothy J. Barth and Alan Wray from NASA Ames for their great contribution to the project, and specifically for providing applications and working with us on their Merrimac implementation. We would also like to thank Bill Reynolds, Parviz Moin, and Juan J. Alonso from Stanford University for their support of this work as part of the Center for Integrated Turbulence Simulation.

9. REFERENCES

[1] Khailany, B., Dally, W. J., Rixner, S., Kapasi, U. J., Owen, J. D., and Towles, B., "Exploring the VLSI Scalability of Stream Processors," *Proceedings of the Ninth Symposium on High Performance Computer Architecture*, Anaheim, California, USA, February 2003, pp. 153–164.

[2] *n*VIDIA®, "*n*VIDIA® GeFORCE^{TM} FX," http://www.nvidia.com/docs/lo/2430/SUPP/PO_GFFX_Consumer_030503.pdf.

[3] Semiconductor Industry Association, *The International Technology Roadmap for Semiconductors*, 2001 Edition.

[4] Dally, W. J. and Poulton, W., *Digital Systems Engineering*, Cambridge University Press, 1998.

[5] Khailany, B., Dally, W. J., Rixner, S., Kapasi, U. J., Mattson, P., Namkoong, J., Owens, J. D., Towles, B., and Chang, A., "Imagine: Media Processing with Streams," *IEEE Micro*, March/April 2001, pp. 35–46.

[6] Kapasi, U. J., Rixner, S., Dally, W. J., Khailany, B., Ahn, J. H., Mattson, P., and Owens, J. D., "Programmable Stream Processors," *IEEE Computer*, August 2003.

[7] Kallinderis, Y. and Vidwans, A., "Generic Parallel Adaptive-Grid NavierStokes Algorithm," *AIAA Journal*, Vol. 32, 1994, pp. 54–61.

[8] MIPS Technologies, *MIPS64 20Kc Core*, http://www.mips.com/ProductCatalog/P_MIPS6420KcCore.

[9] Clos, C., "A Study of Non-Blocking Switching Networks," *Bell System Technical Journal*, Vol. 32, 1953, pp. 406–424.

[10] Leiserson, C. E., "Fat-Trees: Universal Networks for Hardware Efficient Supercomputing," *IEEE Transactions on Computers*, Vol. 34, No. 10, October 1985, pp. 892–901.

[11] Reed, W. H. and Hill, T. R., "Triangular mesh methods for the neutron transport equation," Tech. Rep. LA-UR-73-479, Los Alamos National Laboratory, Los Alamos, New Mexico, 1973.

[12] Cockburn, B., Hou, S., and Shu, C., "TVB Runge-Kutta Local Projection Discontinuous Galerkin Finite Element Method for Conservation Laws IV: The multidimensional case," *Math. Comp.*, Vol. 54, 1990, pp. 545–581.

[13] Jaffre, J., Johnson, C., and Szepessy, A., "Convergence of the Discontinuous Galerkin Finite Element Method for Hyperbolic Conservation Laws," *Math. Models and Methods in Appl. Sci.*, Vol. 5, No. 3, 1995, pp. 367–386.

[14] Barth, T., "Simplified Discontinuous Galerkin Methods for Systems of Conservation Laws with Convex Extension," *Discontinuous Galerkin Methods*, edited by Cockburn, Karniadakis, and Shu, Vol. 11 of *Lecture Notes in Computational Science and Engineering*, Springer-Verlag, Heidelberg, 1999.

[15] Darve, E. and Pohorille, A., "Calculating Free Energies using Average Force," *Chemical Physics*, Vol. 115, No. 20, 2001, pp. 9169–9183.

[16] Darve, E., Wilson, M., and Pohorille, A., "Calculating Free Energies using a Scaled-Force Molecular Dynamics Algorithm," *Molecular Simulation*, Vol. 28, No. 1–2, 2002, pp. 113–144.

[17] Fatica, M., Jameson, A., and Alonso, J. J., "STREAMFLO: an Euler solver for streaming architectures," *submitted to AIAA Conference*, Reno, Nevada, USA, 2004.

[18] Jameson, A., "Analysis and design of numerical schemes for gas dynamics 1. Artificial diffusion, upwind biasing, limiters and their effects on accuracy and multigrid convergence," *International Journal of Computational Fluid Dynamics*, Vol. Volume 4, 1995, pp. 171–218.

[19] Jameson, A., "Analysis and design of numerical schemes for gas dynamics 2. Artificial diffusion and discrete shock structure," *International Journal of Computational Fluid Dynamics*, Vol. Volume 5, 1995, pp. 1–38.

[20] Russell, R. M., "The CRAY-1 Computer System," *Communications of the ACM*, Vol. 21, No. 1, Jan. 1978, pp. 63–72.

[21] Simmons, M. L., Wasserman, H. J., Lubeck, O. M., Eoyang, C., Mendez, R., Harada, H., and Ishiguro, M., "A performance comparison of four supercomputers," *Communications of the ACM*, Vol. 35, No. 8, Aug. 1992, pp. 116–124.

[22] Intel®, "Intel® 850E Chipset," http://www.intel.com/design/chipsets/850e/index.htm.

[23] Intel®, "Intel® Pentium® 4 Processor," http://www.intel.com/products/desktop/processors/pentium4/index.htm.

[24] Dally, W. J., "Performance Analysis of k-ary n-cube Interconnection Networks," *IEEE Transactions on Computers*, Vol. 39, No. 6, June 1991, pp. 775–785.

[25] Kessler, R. E. and Schwarzmeier, J. L., "Cray T3D: a new dimension for Cray Research," *Proc. of the IEEE Computer Society International Conferrence (COMPCON)*, Feb. 1993, pp. 176–182.

[26] Scott, S. L. and Thorson, G. M., "The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus," *Proc. of the Symposium on Hot Interconnects*, Aug. 1996, pp. 147–156.

[27] Gupta, A. K., Dally, W. J., Singh, A., and Towles, B., "Scalable Opto-Electronic Network (SOENet)," *proceedings of Hot Interconnects (HotI) X*, Stanford, California, USA, August 2002.