Stream Programming on General-Purpose Processors

Jayanth Gummaraju and Mendel Rosenblum
Computer Systems Laboratory
Stanford University, Stanford, CA 94305
{jayanth@stanford.edu, mendel@cs.stanford.edu}

Abstract— **In this paper we investigate mapping stream programs (i.e., programs written in a streaming style for streaming architectures such as Imagine and Raw) onto a general-purpose CPU. We develop and explore a novel way of mapping these programs onto the CPU. We show how the salient features of stream programming such as computation kernels, local memories, and asynchronous bulk memory loads and stores can be easily mapped by a simple compilation system to CPU features such as the processor caches, simultaneous multi-threading, and fast inter-thread communication support, resulting in an executable that efficiently uses CPU resources.**

We present an evaluation of our mapping on a hyper-threaded Intel Pentium 4 CPU as a canonical example of a general-purpose processor. We compare the mapped stream program against the same program coded in a more conventional style for the general-purpose processor. Using both micro-benchmarks and scientific applications we show that programs written in a streaming style can run comparably to equivalent programs written in a traditional style. Our results show that coding programs in a streaming style can improve performance on today's machines and smooth the way for significant performance improvements with the deployment of streaming architectures.

Keywords: stream architectures/programming, prefetching, hyper-threading.

I. Introduction

Streaming architectures [1][2][3][4] have demonstrated significant performance advantages over traditional architectures in domains such as signal processing [5], multimedia and graphics [6]. Early results for more general CPU intensive workloads (e.g., scientific applications) also suggest the potential for significant performance benefits [2]. With the announcement of the Cell processor [7], it is likely that streaming architectures will be common in a few years.

While applications such as signal processing require little change to utilize a streaming architecture, most programs are not expressed in a streaming-style. For example, scientific programs typically feature characteristics like unstructured meshes [8], non-linear grids [9], and adaptive griding [10] that do not fall into obvious stream domains. Therefore, these applications have to be recoded, with possibly new algorithms, in order to exploit stream architectures.

A new architecture that requires a different programming model presents challenges for software developers, particularly during a transitional period when traditional architectures are more common. Our work attempts to address this issue by allowing developers to code in a streaming-style and yet efficiently map their programs onto traditional CPU architectures. Using our approach programmers can code in a style that runs well today and be well positioned to exploit the greater parallelism of future streaming architectures.

In this paper we describe techniques for mapping stream programs onto commodity x86 CPUs (e.g., Intel Pentium 4). Even though the commodity CPUs are not originally designed to run stream programs efficiently, we show that effective mapping enables them to run comparably to equivalent programs written in traditional programming styles compiled with the best available commercial x86 compilers. In experiments on real systems we show that stream versions of scientific applications can run as much as 27% faster than the standard C version of the programs. The performance of micro-benchmarks show improvements as high as 92% with the worst case micro-benchmark running 4% slower. Note that these results do not suggest that stream programs perform better than hand-tuned traditional programs. The results show that by mapping stream programs effectively even commercial compilers can generate code that runs well on today's processors.

We present our work as follows. In Section II we describe the salient features of stream programming and differences from traditional programming styles. In Section III we discuss the mapping of stream programming onto general-purpose processors. In Section IV we present an evaluation of the mapping on a hyper-threaded Pentium 4 processor. In Section V we discuss the limitations of stream programming and compare it to other techniques for obtaining high performance. In Section VI we conclude the paper with a brief summary and directions for future work.

II. Stream Programming Paradigm

Stream programming encourages a style of programming that expresses the parallelism inherent in a program by decoupling computation and memory accesses [11][12]. The explicit parallelism and locality of data in a stream program makes it easier to compile efficiently using traditional compiler optimizations. In this section we briefly review

```
for i = 0..n
{
   … = a[index1[i]].val + b[index2[i]].val;
      :
      :
   … = foo (a[index1[i]].val2 * c[index3[i]].val);
      :
   d[i].val = …;
}

for i = 0..n
{
   … = x[index4[i]].val;
      :
      :
   … = bar(d[i]);
      :
   y[index5[i]].val = …;
}
```

Fig. 1. Example of a regular code. This code has two loops. In each iteration of the first loop the data from arrays a, b, and c are loaded from memory using index arrays, and then operated upon to produce a result $d[i]$, which is then stored back to memory. In each iteration of the second loop, data from array x and array d are loaded from memory, operated upon, and the result is stored back to memory at $y[index5[i]]$.

the concepts underlying the stream programming style and describe an execution model for stream programs.

A. Programming in Streaming Style

Stream programming advocates a *gather*, *operate*, and *scatter* style of programming. First, the data is *gathered* into a stream from *arrays* in memory. The data is then *operated* upon by one or more *kernels*, where each kernel comprises of several operations, usually several hundred of them. Finally, the live data is *scattered* back to arrays in memory. In essence, a stream program decouples computation and memory accesses by boosting the memory reads before the computation, and postponing writes of the live data to memory after the computation. Stream programming, therefore, converts the memory latency problem into a memory bandwidth problem.

The *gathers* and *scatters* of data could be from or to sequential, strided, or random locations in an array. Graphics and image processing applications are examples of applications that use sequential loads/stores. Scientific applications, especially irregular grid/mesh computations, exhibit random accesses to memory which makes streaming them more challenging. The randomly accessed locations first need to be collected into *index* arrays before gathering/scattering the data.

To highlight the differences between regular code and stream code we present a simple example (Figures 1 and 2)[1]. There are several differences between the two code fragments. In the stream code the memory operations are completely separated from the computation kernels. The stream code *copies* data into streams from arrays before executing the kernels and *copies* the results from streams to arrays after the kernel executes. The stream memory operations are performed in bulk. Therefore, the data access pattern is explicitly known. In addition, storing of the array d

[1]We have borrowed the syntax for the stream code from Stream Virtual Machine API Specification[13].

```
  streamGather (as, a, index1, 0, n);
  streamGather (bs, b, index2, 0, n);
  streamGather (cs, c, index3, 0, n);
  kernelCall ("kernel1", as, bs, cs, ds);

  streamGather (xs, x, index4, 0, n);
  kernelCall ("kernel2", xs, ds, ys);
  streamScatter (ys, y, index5, 0, n);
```

(a) Stream Operations

```
kernel1 (as[], bs[], cs[], ds[])
{
  for i = 0..n
  {
     … = as[i].val + bs[i].val;
        :
     … = foo (as[i].val2 * cs[i].val);
        :
     ds[i].val = …;
  }
}

kernel2 (xs[], ds[], ys[])
{
  for i = 0..n
  {
     … = xs[i].val;
        :
     … = bar(ds[i]);
        :
     ys[i].val = …;
  }
}
```

(b) Kernel code

Fig. 2. Stream Code: Data from arrays a, b, and c are gathered into streams as, bs, and cs and then operated upon by kernel1 which produces the stream ds. ds and data gathered from an array x are operated upon by kernel2 producing a result ys, which is scattered back to memory (array y). The computation kernels are separated out from the memory accesses. The data inside kernels is mostly read from streams.

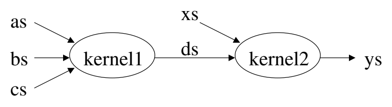


Fig. 3. SDF: Synchronous Data Flow graph of the example in Figure 2. kernel1 takes in as input as, bs, and cs and produces ds. kernel2 takes the output of kernel1 (ds) and xs as input and produces ys.

at the end of the first loop and loading it back in the second loop is completely eliminated in the stream code. In other words, the data produced by *kernel1* is locally consumed by *kernel2*. This locality, called producer-consumer locality, is frequently exploited in stream programs.

Stream programs can be simply represented using Synchronous Data Flow (SDF) graphs[14] (Figure 3). The inputs and outputs to the computation kernels, and the dependencies between the kernels are explicitly indicated. We use this representation to describe the scientific applications in Section IV.

B. Stream Execution Model

A typical procedure for executing a stream program on a stream processor is as follows [15]. The stream program is first compiled to a Stream Virtual Machine (SVM), which is an abstract machine model that captures the essential characteristics of stream architectures. The SVM is then compiled, using a machine specific compiler, to the underlying stream architecture. Using SVM as an intermediate enables development of compilation tools independent of a particular stream architecture.

A Stream Virtual Machine is comprised of processors (control processor, compute/kernel processor, DMA engines), and memories (global memory, local memory (Stream Register File or SRF), and local registers)[2] and defines the following execution model:

- The control processor schedules computation kernels and asynchronous bulk memory operations on the kernel processor and DMA units respectively.
- The DMA units *gather* the stream data needed by the computation kernels from global memory into the SRF.
- The computation kernels operate on the data in the SRF, using the local register file for storing the intermediate temporaries, and writing the live data back to the SRF.
- Once all the computation kernels that have producer-consumer locality finish execution, the DMA units write the live stream data back to global memory.

In order to execute a stream program efficiently, a stream compiler performs several transformations and generates SVM code. The streams are broken down into strips, each typically several thousand bytes long, to insure that the working set of strips is in the SRF. This optimization, known as *strip-mining*, should be performed in such a way that producer-consumer locality of streams is exploited. The streams are *double buffered* so that when one buffer is being loaded from memory, the other (already loaded) buffer can be operated upon in parallel by the computation kernels. In essence, the buffers are *renamed* to eliminate false dependencies. Finally, the stream compiler selectively copies only those fields of the original array record into the SRF that are actually used inside the kernels.

An additional optimization we found useful was to reorganize the fields of the array records [16] so the fields accessed by kernels can be copied to/from the SRF using optimized block copy routines rather than individual loads and stores. These routines are effective when many fields (order of tens of bytes) have to be copied from the original record.

III. Mapping Stream Programming Paradigm to the General-Purpose Processor

Mapping the stream execution model to the general-purpose processor involves mapping both the processors and memories. Some of these mappings are straightforward – SVM local registers and global memory map to the register file and main memory of a general-purpose processor. Mapping SRF and the three processors (control, compute, and DMA) is more challenging.

We present our mapping in two parts. We first discuss mapping the SRF to a cache and then present our mapping of computation and memory accesses onto one or more *threads*. Mapping to multiple threads is particularly interesting given the general trend among general-purpose processors to move toward multi-threading and multiple cores.

A. Mapping Stream Register File to Cache

Programs coded for the SVM assume large, compiler-controlled local memories that have a high bandwidth path both to global memory and the functional units of the processing core. This local memory (SRF) is not present in most general-purpose CPUs. Fortunately, these CPUs do have some on-chip memory in the form of processor caches.

While processor caches are not typically compiler-controlled, we can effectively pin a cache-sized range of memory addresses in cache and use it as the stream register file (SRF) by carefully managing memory accesses in the mapped program. We size the SRF so it comfortably fits into the cache and control non-SRF memory references to ensure that they do not trigger replacement of the SRF from the cache. This ensures that the processor brings the SRF into the cache and does not write it back or replace it until the program has finished executing. We allocate the SRF such that it leaves one or two cache lines in each set available for non-SRF data.

Computation kernels access data only in the SRF and hence will not suffer cache misses to global memory. Gather and scatter operations are implemented by copying data in or out of the SRF. Because the SRF is effectively pinned in the cache, these copy operations will likely suffer memory stalls when reading or writing global memory but will *hit* on the stream data in SRF. In this way the *streamGather* and *streamScatter* operations behave as they would on a stream architecture. Program accesses to non-SRF memory occur primarily during gather and scatter operations. These references can exploit instructions that control the cacheability of data so that SRF is not replaced.

On the Pentium 4 processor we sized the SRF to fit into the on-chip L2 cache. The cache is large (1MB), has high associativity (8-way) and provides fast access (25 cycles) from the functional units. The L2 cache is also connected to main memory by a 6.4GB/s front-side bus providing high bandwidth access to main memory. We avoid interference in the cache with non-SRF data arrays (from/to which streams are loaded/stored) by using Intel's *non-temporal load/store* instructions. The gather operation uses non-temporal prefetch instructions to load data-arrays into SRF and the scatter operation uses non-temporal store instructions to store data-arrays to memory.

Besides memory references generated by gather and scatter operations, program instruction fetches, stack accesses, and

[2]The SVM execution model for more than one node contains multiple sets of these processors and memories and network links to connect the nodes. In this paper, we focus only on mapping a single node of SVM to the general purpose processor.

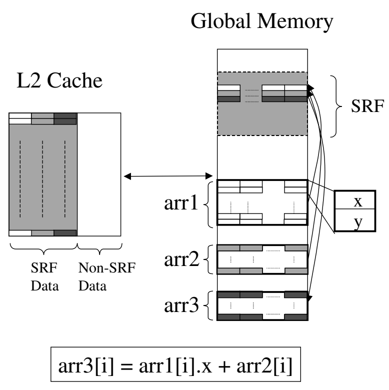


Fig. 4. Conceptual view of mapping SRF to the memory system of general-purpose processors: A simple example where elements of two arrays ($arr1$ and $arr2$) are summed together and stored in a third array ($arr3$). Each element of $arr1$ is a record comprising of two fields, x and y, and only x is used in the summation.

operating system code and data generate memory references that could potentially conflict with the SRF. Our experience with scientific programs (containing hundreds of local variables) has shown that this is not a problem. Measurements of cache miss rates on the SRF show a negligible number of misses and hence, has an undetectably small effect on the overall performance.

Figure 4 shows a conceptual view of mapping SRF to the memory system for a simple example. SRF is mapped to a contiguous region in the main memory. SRF also occupies a significant portion of the L2 cache comprising the working set of *strips*. The figure illustrates a simple example in which the contents of two arrays are added together and stored in a third array. Elements of $arr1$ and $arr2$ are non-temporally prefetched into the non-SRF portion of the cache. The useful data from the arrs are copied into the SRF and then a kernel for performing the summation is invoked. The kernel executes and stores the result in the SRF. Finally, the result is copied from the SRF to $arr3$ using non-temporal stores.

Evaluation of SRF mapping on Pentium 4: Stream programming converts the memory latency problem to a memory bandwidth problem. Therefore, it is important to optimize for memory bandwidth in order to map the SRF effectively. Using the system described later in Section IV, we characterized the memory system bandwidth behavior for different scenarios. We measured the bandwidth at which we could gather and scatter data to/from the SRF as we varied the memory access pattern.

Figures 5(a) and 5(b) shows the bandwidth, measured in GB/s, we achieve for *streamGather*s of 4 bytes of data from either (a) sequential records or (b) randomly selected records. We vary the size of the record from 4 bytes up to 128 bytes, the size of the L2 cache line of the machine. The field size is kept constant at 4 bytes, representing the range of gather operations of 4 byte data from records of different size. Figures 5(c) 5(d) shows similar data for (a) sequential and (b) random *streamScatter*s.

The overall behavior we see is dependent on how the cache and memory system optimizations on the Pentium 4 support the memory reference patterns we generate. The Pentium 4 is quite good at streaming sequentially through memory whereas random memory accesses are not optimized. In the following, we describe the behavior of the memory system first without using prefetch and non-temporal instructions (indicated by the black lines in Figure 5) and then using prefetch and non-temporal instructions (indicated by gray lines in Figure 5).

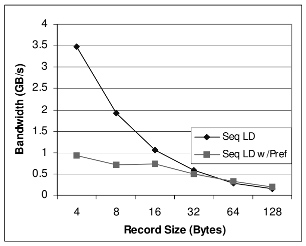
Sequential performance is very good when the record size is close to the access size but drops quickly as the record size is increased. The behavior is dominated by the cache-line loading and hardware prefetch of the processor. Reading 4 bytes from a 4 byte record into the SRF is effectively byte copying from memory into SRF which is stored in the cache. The processor and its prefetch logic can do this at close to the speed of the memory bus.

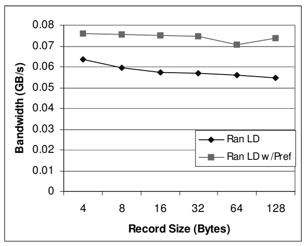
As we increase the record size, but continue to use only 4 bytes per record the cache-line fetches are less effective and our bandwidth drops. This corresponds to strided memory access where the record size is the stride distance. By the time we get to 128 byte records we are using only 1/32 of the data fetched by the processor so even though the memory bus is fully-utilized, we get only $141MBytes/s$.

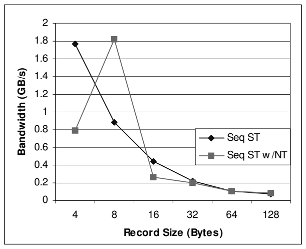
For random indexed gather operations we effectively read from random memory locations and copy them into the L2 cache. The cache line fetches generated by the processor to our load request for 4 bytes are wasteful since there is no locality in the reference pattern to exploit. Consequently our bandwidth is low ($63MB/s$). We believe that more than missing in the cache, missing in the TLB is the dominant factor for the lower bandwidth. Each memory reference suffers the delay of Pentium hardware page table walk before the cache miss can be issued to the memory subsystem.

Figures 5(c) and 5(d) shows a very similar behavior for both sequential stores and random scatters. For sequential stores we have similar behavior to the sequential loads except that to do the store the L2 cache first reads the cache line and then later writes it back. This limits the maximum bandwidth to half the sequential load performance. Like the random gather case, the random scatter performance is dominated by TLB, cache miss overheads, and inability to use hardware prefetcher resulting in significantly lower bandwidth than the sequential case.

As mentioned in the previous section, we desire to tag the cache lines fetched from memory so they will not cause replacement of the cache lines making up the SRF. On the Pentium 4 processor this is done differently for loads and stores. For loads, we need to use prefetch instructions with a special non-temporal hint. For stores, the Pentium 4 processor supports a special non-temporal store instruction (movntq).


(a) Sequential Loads


(b) Random Gathers


(c) Sequential Stores

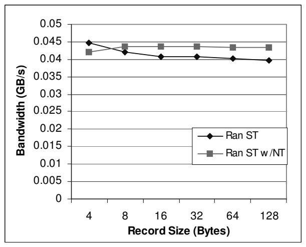

(d) Random Scatters

Fig. 5. Effects of changing record sizes on memory bandwidth usage for Sequential Loads/Stores and Random Gathers/Scatters

The gray lines in Figure 5 show the performance of the bandwidth tests with these non-temporal hints.

Our results with non-temporal memory accesses were mixed. For operations already highly optimized by the hardware such as sequential loads that reduce to a memory copy, the non-temporal support actually reduced performance significantly for loads and had a mixed effect for sequential stores. For the random cases the non-temporal support significantly improved performance by as much as 32%.

Our experience with coding of the bulk memory operations on the Pentium 4 lead us to believe there is an opportunity for even further performance improvements. Although our results came in below block copy performance others have reported, it seems like it should be possible to specially code these block memory copies based on the access type and pattern and also the processor's micro-architecture to get the maximum possible performance out of the memory system.

B. Mapping Computation and Memory Accesses

In this section we discuss the second part of our mapping scheme – scheduling computation kernels and memory accesses onto the hardware contexts of a general-purpose processor. We first describe a low overhead implementation of a distributed work-queue method for scheduling computation/memory tasks. Later, we present mappings of this implementation onto one or more hardware contexts of a general-purpose processor.

1) Scheduling Using a Distributed Work Queue: The output of a stream compiler targeting the SVM is a set of asynchronous bulk memory accesses that gather or scatter data in or out of the SRF, and computation kernels that operate on data in the SRF. Along with these tasks is the list of dependencies between the different operations. The objective is to execute the computation kernels and memory accesses as fast as possible by overlapping the two within the constraints of these dependencies.

To schedule the compute and memory tasks we use a hybrid approach employed in the stream scheduler of Imagine [17] and also advocated by the SVM Execution Model [15]. In the hybrid approach the dependencies between the computation and memory tasks are determined at compile time and scheduling of the tasks is performed at runtime. This approach has the advantage of handling compile time unknowns like conditionals, execution time of kernels, etc. For example, if kernel A or kernel B is executed depending on the resolution of a condition at runtime, a purely compiler based schedule would not be efficient. The stream scheduler also determines the optimal strip-sizes of streams depending on the flow rates of streams, SRF size, etc. We next describe our proposed low overhead implementation of the hybrid

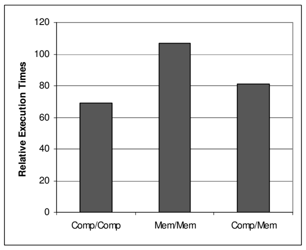


Fig. 6. Computation and Memory overlap: The figure depicts three scenarios for the hyper-threaded Pentium 4 processor with two contexts: a. Both contexts perform computation, b. Both contexts perform memory accesses, and c. One context performs computation and the other performs memory accesses. Each bar represents the normalized execution time to a hyper-threaded processor running one thread in the ST (Single-Thread) mode and taking 100 execution units.

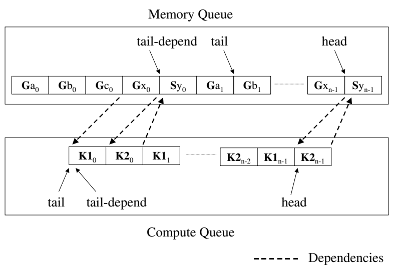


Fig. 7. Distributed work-queue. The upper case letters `G`, `S`, and `K` refer to Gather, Scatter, and Kernel operations; the lower case letters (`a`, `b`, `c`, `x` and `y`) refer to the streams; the subscripts $0...n-1$ refer to the number of the strip; *head* points to the last enqueued task; *tail* points to the task currently being executed; and *tail_depend* points to the oldest task in the queue that hasn't yet been executed.

approach using a distributed work queue.

Based on our experience with running multiple tasks on the Pentium 4, we adopted a scheme that limits the use of parallel hardware contexts to overlapping bulk memory accesses with computational kernels (as opposed, for example, to overlapping multiple bulk memory accesses). In an experiment shown in Figure 6, we examined the synergy and destructive interference on running parts of computation intensive kernels and memory accesses at the same time to evaluate how efficiently the hardware contexts overlap the two operations. The results are normalized to the time it would take to perform both the operations in series using the hyper-threaded processor in single-thread mode[3].

From Figure 6 we can see that there are performance advantages to running computation in parallel with memory accesses and with other computation. The overall execution time is reduced by 20% to 30%. Trying to overlap two bulk memory accesses results in destructive interference slowing down the operations by 6%.

We believe that the improvement in overlapping computation and memory is primarily because each thread is efficiently using mutually exclusive resources of the processor (ALU units vs. memory system resources (e.g., ld/st queues, caches))[4]. Overlapping two computation operations provides benefits primarily because of TLP (Thread-level Parallelism) – the Instruction Window of the processor has instructions from both the threads providing more variety of instructions for scheduling. However, depending on the instruction mix the benefits might vary significantly. Overlapping two memory operations doesn't help at all because both threads compete for the memory system resources, which is the primary bottleneck in this experiment.

The chief challenge we faced was to efficiently schedule the tasks on the hardware contexts within the constraints. To do this we use a carefully crafted distributed work queue scheme. This scheme uses two queues – one queue is dedicated to holding memory tasks (*memory queue*) and the other queue is dedicated to holding compute tasks (*compute queue*). The control thread enqueues the Gather/Scatter and Kernel tasks into the memory queue and compute queue respectively, along with the dependence information. The compute thread and memory thread dequeue and execute the tasks after ensuring that the task has no outstanding dependencies. Figure 7 illustrates this method using the example presented in Section II (Figure 2). The figure shows a scenario when the tasks are executing out-of-order. The memory thread has finished executing several gather operations (Ga_0, Gb_0, Gc_0, Gx_0, and Ga_1) and the compute thread is still executing $K1_0$. The scatter task Sy_0 in the memory queue has not been executed yet, indicated by *tail_depend* pointing to Sy_0. This is because Sy_0 depends on the kernel task $K2_0$ which has not yet executed. This scenario is common if the amount of computation per memory reference (also called *arithmetic intensity*) is very high.

The dependence information between tasks is encoded using bit-vectors. Each element of the queue maintains a bit-vector indicating which tasks it depends on. For example, the Scatter task Sy_0 maintains dependence information with $K2_0$. Therefore, whenever a task finishes execution the dependences should be cleared. A bit-vector representation is useful because setting and clearing dependence information could be performed rapidly (using simple *or* and *and* instructions, for example). However, if the number of tasks is too large then the bit vector representation gets cumbersome. We handle this problem by enqueuing at most a fixed maximum number (e.g. 64) of elements in the queue at any given time.

2) Mapping to Hardware Contexts: We next present a mapping of the distributed work-queue method to processors that support two hardware contexts. Not only does this mapping provide a starting point for processors that support multiple contexts, but it also enables us to evaluate our mapping scheme on a real system that has two contexts (Hyper-threaded Pentium 4 processor).

Mapping computation and memory accesses to two hardware contexts is a challenging problem because the computation kernels, asynchronous bulk memory accesses, and the control task should be time-multiplexed between the two contexts. Because the control task is used only sporadically, a possible implementation is to have one context dedicated to memory accesses and have the other context switch between control thread and computation thread. This implementation is likely to be better than dedicating the computation to one context and switching between control and memory tasks on the other context because the control thread operations like enqueuing of tasks are easily overlapped with the *first* (gather) and *last* (scatter) operation of the software pipelined stream program.

In the proposed method one thread context effectively *fetches* into the cache the data needed in the other thread context. It is implicitly assumed here that there is a level in the cache hierarchy that is shared among multiple contexts. This is true in the existing multi-threaded processors, and is also true in current (IBM Power 4) and future (Intel Montecito) multi-core processors.

The proposed implementation is easily extended to processors that support more than two hardware contexts. The challenge is to divide up the computation and memory tasks into sub-tasks to maximize performance.

For processors that are limited by a single hardware context, a plausible implementation is to have three threads (one each for computation, memory, and control) time-multiplex on the processor. This approach, however, has large synchronization and thread switch overheads because the

[3]In Single Thread (ST) mode, the hardware switches to executing a single context, dedicating all the resources of the processor to this context

[4]Although the ld-st queue is partitioned ([18]) it doesn't affect the results much because the memory bandwidth rather than the number of ld-st slots is the bottleneck

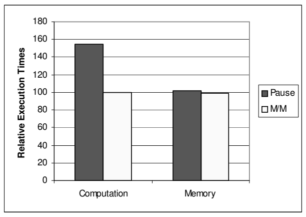


Fig. 8. Pause vs. Monitor/Mwait: The effects of busy-waiting in the Pentium 4 processor with two hardware contexts using (a) PAUSE instruction or (b) monitor/mwait instructions. The execution times are normalized to one thread performing the same computation or memory accesses and taking 100 execution units.

threads would have to time-multiplex frequently to effectively overlap computation and memory accesses. A possibly better implementation is to software pipeline the Gather, Kernel, and Scatter stages on a single thread.

Evaluation of inter-thread communication mechanisms on Pentium 4: As discussed above, an important concern in using the proposed mapping scheme for two contexts is that the compute thread and memory thread would have to communicate with each other after every *gather*, *scatter* and *kernel* stage. This overhead is exacerbated if streams are strip-mined with fewer elements leading to a larger number of strips. Therefore, a fast inter-thread communication support is essential for good performance.

The Pentium 4 hardware contexts are organized for easy integration into modern operating systems. The contexts are exported to look like a shared-memory multiprocessor to the system software. Although easing porting, the thread abstractions make it difficult to quickly dispatch tasks from the distributed work queue. We were faced with the trade-off between quick tasks dispatch and increased resource consumption when idle.

One way of implementing the tasks queue is to have one context, for example, the one doing memory accesses, to spin waiting for the arrival of work in the form of a scatter or gather operations. By using Intel's PAUSE instruction, the resource consumption of the busy waiting spin loop can be reduced[5]. This kind of spin loop allows the dispatch of operations in as little as 175 cycles.

An alternative approach is to use the new Intel SSE3 extension instructions MONITOR and MWAIT. These instructions appear to be intended precisely for the purpose of inter-thread communication. They allow the hardware context to put itself into a sleep state so that all resources are given to the other context. A single write to an address from the other context will wake up the sleeping context. We measured a dispatch time of 680 cycles using this scheme. Activating the sleeping hardware context takes significantly longer than simply feeding commands to an already active context.

[5]A plausible reason for this behavior is that PAUSE provides a hint to the hardware to ensure that there is at most one outstanding load for checking the loop condition at any given time. However, we couldn't find any documentation from Intel corroborating this hypothesis.

Figure 8 shows an experiment where we investigated the runtime overheads of these schemes. Although the PAUSE based loop gives the smallest wakeup latency, the resources consumed spinning greatly impacts the performance of compute intensive tasks running in the other context. However, it has only a negligible impact on memory intensive tasks. The MONITOR/MWAIT instruction has negligible impact on both computation and memory tasks but at a cost of higher wakeup latency.

Other approaches such as using OS level primitives to deschedule and wakeup a hardware context can achieve low idle time overheads but have significant wakeup latency measured in the tens of thousands of processor cycles.

Since we had enough idle hardware context in our scheme we adopted the MONITOR/MWAIT approach in our tests. A common example of this scenario is a compute bound program where the memory thread is idle most of the time.

IV. Experimental Evaluation

In this section we present the evaluation of the stream programming paradigm. We compare the performance of conventional programs and their streaming versions on an architecture (Intel Pentium 4 processor) designed for running conventional programs efficiently. This section is divided into three major parts. First, we describe the experimental framework that we used for performing our experiments. Second, we demonstrate using micro-benchmarks the range of speedups that can be obtained by coding the programs in a streaming-style. Third, we present the results obtained by running a few scientific applications.

A. Experimental Framework

All our experiments were performed on a DELL Dimension 8300 with Intel 3.4GHz Pentium 4 processor (Prescott core) running Linux (Red-Hat's Fedora for SMP). The Prescott core is hyper-threaded with a maximum of two hardware contexts. The L2 cache size is 1MB with 128 byte line size. The Front Side Bus is 800MHz and the chipset used is i925X. The processor has some new instructions in its ISA (PNI) including MONITOR/MWAIT.

To ensure that we use both the hardware contexts of the HT processor, we mapped two software threads (spawned using *pthreads*) to these contexts. For effective mapping the processor was kept mostly idle except for the OS using resources occasionally. We also ran each experiment for several hundred time steps (typically several seconds). To determine the execution time we used a low overhead instruction $rdtsc$ which reads the time stamp counter of the processor.

To get MONITOR/MWAIT to work on the Prescott core was a big challenge. For reasons unknown to us, MONITOR/MWAIT instructions are only available in ring 0 or the kernel mode of the Prescott processor. We overcame this limitation by installing a patch in the Linux kernel that enables us to run a user process in kernel mode.

For each of the micro-benchmarks we performed the following:

1) Coded the programs in C
2) Re-wrote the programs in streaming-style
3) Compiled the stream code by hand to SVM code, which we discuss below in more detail
4) Linked the program with our library for distributed work-queue scheduling and compiled using Intel C Compiler for Linux (*icc* with -O3 option). In our library, we optimized the bulk memory operations like streamGather and streamScatter using software prefetch and non-temporal instructions
5) Executed the program on the Pentium 4 processor
6) Compiled and executed the regular code using icc with -O3 option
7) Determined speedup computed as the ratio of the execution time of regular code to that of the stream code

We performed the following compilation steps by hand:

- Strip-mined the streams in such a way that the working-set of strips fit in the SRF
- Double buffered the strips, so that when one buffer is being loaded from memory, the other (already loaded) buffer is operated upon by the computation kernels
- Aligned fields within records for fast block copies to/from SRF
- Fused adjacent kernels when input streams are shared between them
- Added dependencies between the stream operations and kernels

These compilation steps are easily performed by traditional optimizing compilers. In general, stream programming simplifies a few compiler analyses and makes a few others redundant. For example, streams, by definition are *copies* of arrays and don't alias each other. Therefore, streams are exempt from any *alias analysis*. Thus, transformations like *strip-mining* and *double buffering* become easier. By moving up all the memory references to the *gather* stage, software *prefetching* also becomes easier to implement. The compiler does not need to address difficult issues of prefetching like determining the exact prefetching distance. Determining dependencies between tasks is a straightforward data-flow pass on the SDF graph.

B. Evaluation of Micro-benchmarks

In this section we present results obtained by running a few micro-benchmarks using our mapping scheme. The primary goal of these experiments is to estimate the range of speedups that can be obtained if programs are written in a streaming-style. We created a set of benchmarks that vary the major attributes of stream programming including the memory access patterns (i.e., types of gathers and scatters), ratio of the amount of computation to the amount of memory access, and the amount of producer/consumer locality. We evaluated the following micro-benchmarks:

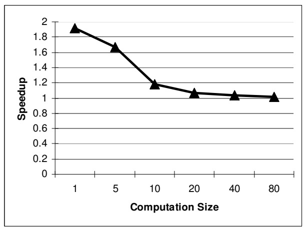
- LD-ST-COMP is a simple loop that traverses two arrays sequentially, computes a result, and stores the result sequentially into a third array. LD-ST-COMP represents programs that do only sequential access to memory. Ex. Parts of streamFEM (AdvanceCell: Figure 10(a)) and streamCDP (FindMaxAndUpdate: Figure 10(b))have this behavior.
- GAT-SCAT-COMP is similar to LD-ST-COMP except that the arrays are read/written in a random order. GAT-SCAT-COMP represents programs using indexed memory accesses to perform non-sequential gathers and scatters from memory. Ex. streamSPAS (Figure 10(d)) and parts of streamFEM (GatherCell: Figure 10(a)).
- PROD-CON accesses memory in a random order and runs two computation loops such that the output of first loop is input to the second one. The first loop reads in two arrays randomly, computes a result, and stores the result sequentially into an output array. This output array (read sequentially) and another array (read randomly) are operated upon, and the results are stored randomly into a final output array. Ex. neo-hookean (Figure 10(c)) has abundant producer consumer locality although it uses sequential loads and stores.

Figure 9 shows speedups obtained by rewriting the micro-benchmarks in a streaming-style. For low *COMP* values each program is completely memory bound and the speedup results from improved memory bandwidth utilization. Unlike the regular code, where the computation and memory accesses are intermixed, in stream programs the loads/stores are separated from computation enabling the optimization of bulk memory loads and stores. This is particularly visible in the speedups obtained in the LD-ST-COMP case, where the hardware prefetcher couldn't improve the performance of the regular code even though the data accesses for individual arrays were sequential because the data accesses were intermixed.

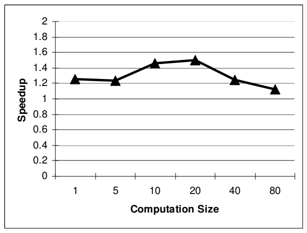
As the computation size increases the speedup improves in GAT-SCAT-COMP and PROD-CON because the processor is able to more effectively overlap computation and memory in the stream code than the regular code. This trend is not observed for LD-ST-COMP because the program is compute bound even at low $COMP$ values. At large sizes of computation there is hardly any speedup in any benchmark because the processor is able to overlap memory accesses and computation efficiently even for regular codes. Finally, PROD-CON shows higher speedup than GAT-SCAT-COMP even though both of them use random loads/stores. This is because PROD-CON has memory bandwidth savings by not writing back the intermediate stream to memory due to producer-consumer locality.

C. Scientific Applications

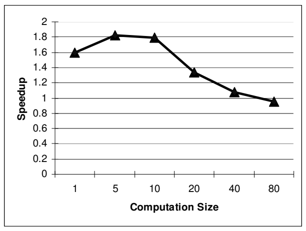
In this section we discuss results obtained by mapping four scientific applications (10 kernels) on the hyper-threaded pro-



(a) LD-ST-COMP

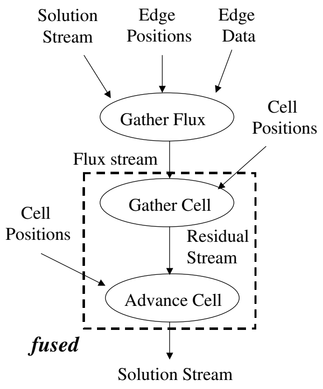


(b) GAT-SCAT-COMP

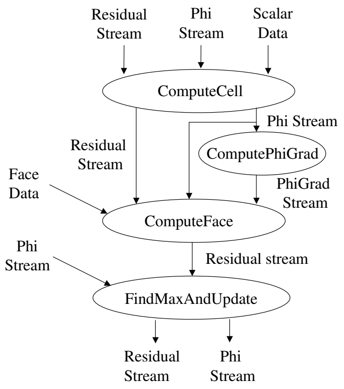


(c) PROD-CON

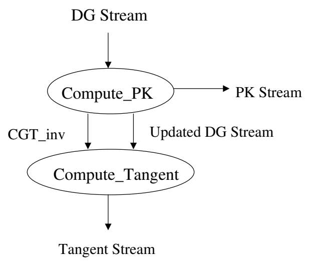
Fig. 9. Regular code vs. Stream code with varying computation sizes for three micro-benchmarks. The size of the computation performed with each loaded value is varied as shown in the figure. $COMP = 1$ roughly corresponds to an execution time of 50 cycles.



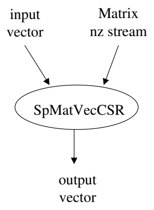
(a) FEM: Data are gathered (from random locations) into GatherFlux and GatherCell kernels, AdvanceCell is a relatively small kernel that loads and stores data sequentially. Because the data are gathered randomly in the GatherCell kernel, there is no straightforward producer-consumer locality between the GatherFlux and GatherCell kernels.



(b) CDP: Residuals, phis and scalar data are input into ComputeCell which generates the updated residuals and phis. The phis are passed through ComputePhiGrad which computes phi gradients. The phi gradients and phi values for the faces are gathered and along with face data input to ComputeFace. This kernel produces residuals which are scattered and added back to cells. FindMaxAndUpdate finds maximum of the residuals and updates residual and phi streams.



(c) NEO-HOOKEAN: Computes stresses in solids. Data from DG stream is loaded sequentially, operated upon by Compute_PK kernel which generates three streams. Two of these streams (CGT_inv and updated DG stream) are read sequentially, operated by a second kernel (Compute_Tangent) to generate a Tangent stream.



(d) SPAS: Sparse Matrix-Vector Multiplication. The input vector corresponding to non-zero (nz) elements of the matrix is gathered and the non-zero elements of the matrix are sequentially loaded from an array. The kernel (SpMatVec) performs a dot product of the two streams and stores the result sequentially to memory.

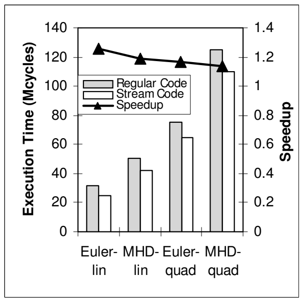
Fig. 10. Important portions of SDFs of scientific applications relevant to stream programming

cessor. For each of the scientific applications, we performed the same set of steps described in Section IV-A. However, most of the applications were originally written in Fortran by programmers specializing in specific application domains (Fluid Dynamics, Solid mechanics, etc). These applications were re-written in C and then converted to stream code in collaboration with the application programmers.

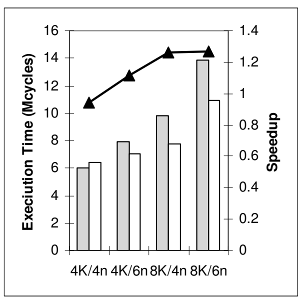
We evaluated scientific applications that feature characteristics like regular/irregular meshes, particle-in-cell computations, and mechanics of solids, which are common in scientific applications:

- streamFEM: Implementation of the Discontinuous Galerkin (DG) Finite Element Method (FEM)[8]
- streamCDP: Implementation of a Large Eddy Simulation (LES)[19]
- neo-hookean: Extends the neo-hookean finite elasticity material model to the the compressible range [20]
- streamSPAS: Computes matrix vector multiply using compressed sparse row storage [21]

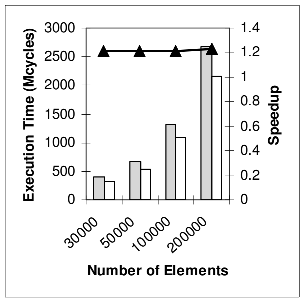
1) streamFEM: The test case we used for evaluating streamFEM (Figure 10(a)) is that of a blast wave computation. With small extensions this application is useful for shock wave capturing. We evaluated the application for two PDE equation sets: Euler (4 PDEs) and Magneto-hydrodynamics (MHD - 6 PDEs) and for two polynomial function spaces: linear (3 degrees of freedom) and quadratic (10 degrees of freedom). We ran these experiments for 4816 triangular cells. These parameters directly affect the amount



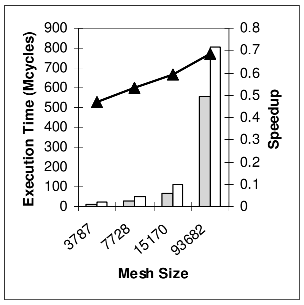
(a) FEM



(b) CDP



(c) NEO-HOOKEAN



(d) SPAS

Fig. 11. Regular code Vs. Stream code. (a) varying record size and computation, (b) varying number of elements and computation, (c) and (d) varying the number of elements.

of computation and record sizes for the data.

Figure 11(a) shows the speedups obtained by rewriting streamFEM in streaming-style. We get speedups between 1.13x to 1.26x. The speedup is mainly due to two reasons. First, the compute operations and memory operations are efficiently overlapped especially for smaller problem sizes (Euler-lin and MHD-lin). As the problem size increases the application becomes compute bound and the effective overlap is reduced leading to smaller speedups. Second, GatherCell and AdvanceCell kernels are fused into a single kernel. The observation that both kernels share the same input streams led to this optimization.

2) streamCDP: streamCDP is a transport advective equation solver typically used for large eddy simulations (LES) (within jet engine simulation routines, for example). This application has been implemented using a second-order WENO (Weighted Essentially Non-Oscillatory) [19] scheme.

streamCDP (Figure 10(b)) posed several challenges for streaming. There is limited straightforward producer-consumer locality between kernels. Either the output data generated by the producer kernel are randomly *scattered* or the input data to the consumer kernel are randomly *gathered*. Next, the ComputeFace kernel contains a data-dependent conditional. Therefore, either data needed on both sides of the conditional had to be gathered or an index stream needed to be generated to gather the required data. For our implementation we chose the former because of high spatial locality between data needed on both sides of the conditional. Finally, we considered fusing ComputeCell and ComputePhiGrad kernels to reduce memory bandwidth usage by reusing the input streams. We decided against fusing the kernels because some amount of re-computation was necessary for correctness, and for our data-sets computation rather than memory bandwidth was the dominant performance limiter.

Figure 11(b) shows the results obtained by writing this application in streaming-style. We considered a cubic mesh (6n or 6 neighbors) and a square grid (4n) and for each case we varied the number of elements (4096 and 8192). The results show that the performance improves with increasing neighbors and increasing mesh size (.94x to 1.27x speedup). This behavior is expected because streamCDP is compute bound for small problem sizes and the memory bandwidth requirement grows faster than computation with increasing problem size.

3) neo-hookean: This application (Figure 10(c)) uses material properties to compute stresses and the constitutive tangent matrix of the material. This application has abundant producer consumer locality, as seen from Figure 10(c).

Figure 11(c) shows the results obtained by writing neo-hookean in streaming-style. The results show that we get speedups between 1.21x and 1.23x. The primary reason for the speedup is abundant producer consumer locality between the kernels. The two intermediate streams are never written back to memory resulting in huge memory bandwidth savings (approximately Number of elements * 144 bytes). We next present an example where the code written in streaming style actually performs worse than the regular code.

4) streamSPAS: streamSPAS (Figure 10(d)) computes matrix vector multiply using compressed sparse row storage. The basic idea of the algorithm is to store only the non-zero elements of the matrix and the corresponding indexes to these elements, and compute the matrix vector product.

Figure 11(d) shows the results of re-writing $streamSPAS$ in streaming-style. The matrices that we used for our experiments come from 3D FEM discretization. In these experiments the ratio of number of nonzero elements to number of rows in the matrix is kept constant (approximately 46).

The results show a slow-down for the meshes under consideration primarily because when data are gathered into the input vector stream, several elements are copied multiple times. For every non-zero element in the matrix, one element is copied from the input vector into the stream register file. This is done to keep the input vector data contiguous in the SRF when performing the dot product. This approach disadvantages the stream code performance especially for small mesh sizes where the cache is effective for the regular code. By using better techniques for the stream code, like matrix blocking, the slow-down can be avoided. As the mesh size increases with the same ratio of non-zero elements to number of rows, the stream code starts performing better. We believe this is because the mesh gets sparser and this results in fewer *extra* copies in the SRF.

V. Discussion and Limitations

A. Speedup Limitations

Although our study was limited to applications that have both streaming and traditional implementations, we believe our results will hold for many but not all scientific applications. Notable exceptions include highly-tuned applications and applications that do not appear to map well onto the stream programming paradigm.

Significant scientific applications such as GROMACS [22] have already been highly tuned to run on general-purpose processors taking advantage of the caches, special instructions such as SSE, prefetching, etc. It is unlikely that a stream version of these applications will see speedup using our system since if there was a faster way to run the application it would have been coded that way already. Our approach is to have the programmer code in a streaming style and be able to automatically map to a high performance implementation without the time consuming low-level optimizations.

Not all types of applications appear amenable to streaming. SPECint benchmarks, for instance, that use data structures such as trees and linked-lists are difficult to code in a streaming style. Therefore, it is important to determine the characteristics of applications that make them good candidates for streaming on general purpose architectures. Some of these characteristics include applications with: memory bottlenecks, large numbers of elements (much bigger than the cache size), huge records, and producer-consumer locality. In addition, it might not always be straight-forward to write applications in streaming-style. As seen in the streamSPAS example, coding in streaming-style could potentially hurt performance if several copies of the original data is made using streams.

Although we can program Pentium 4 like a stream processor, the Pentium 4 architecture and micro-architecture are unable to take full advantage of it. This is because a large fraction of the chip area is dedicated to logic that extracts fine-grained parallelism, rather than to functional units that performs the actual computation. In addition, the asynchronous bulk memory transfers are affected by TLB mapping, limiting the bandwidth utilization to a small fraction of the peak memory bandwidth (as shown in Section III). We believe that changes to the micro-architecture like adding more functional units and increasing TLB mapping could substantially improve the performance of stream programs.

B. Compiler Optimizations and Stream Programming

The speedups we showed in the experimental section were based on *standard* C implementations using a modern compiler system. In theory one could write a compiler that could take a standard C program and output a binary that executed much in the same way as our mapping. A variety of compiler optimizations have been developed in research compilers that come pretty close to doing this. Examples include affine mappings [23], global cache reuse [24], integrated loop and data transformations [25], and cache conscious data layout [16], to name a few. Stream programming facilitates several of these optimizations by separating computation and memory accesses, reducing aliasing problems, and making explicit the locality of data. This enables even commercial compilers to successfully optimize difficult cases such as irregular grids. Moreover, stream programs are written using domain-specific information which is difficult to extract automatically by optimizing compilers.

Hyper-threading has been used for prefetching, with one thread performing prefetches and the other thread running regular code [26][27]. In our scheme we not only prefetch the data with a second thread but also copy it into the cache. Unlike software prefetching using a compiler [28][29] our scheme does not need to worry about prefetch distance since we are operating with bulk asynchronous memory gathers/scatters. We effectively turn what was a latency problem for the program into a bandwidth problem.

A big benefit of stream programming is that it forces the programmer to think about memory accesses and compute operations separately. This could affect the choice of the algorithm (s)he makes, leading to better mapping by the compiler for huge performance gains. For example, in neo-hookean the data access pattern was slightly changed to re-write the code in streaming style leading to code that had producer-consumer locality.

C. Stream vs Vector Programming Styles

Stream and vector programming styles share several similarities. Both provide scatter and gather operations. Both operate on a series of elements of a homogeneous type. However, vector elements are typically limited to the basic types whereas stream elements can have arbitrary types (records with potentially multiple fields). In addition, vector programming is limited to simple, predefined operations like vector addition and subtraction. In contrast, stream programming allows the use of general operators defined by the programmers known as kernels which typically have several hundred operations.

Intel's Streaming SIMD Extensions (SSE) add *vector*-oriented capabilities to general-purpose processors. They provide the ability to efficiently transfer multiple data elements between memory and a small number of 128-bit vector-like registers. In contrast, stream programming encourages the use of bulk memory transfers (e.g., hundreds or thousands of bytes) and computation kernels with typically several hundred instructions. In this paper we exploited the efficient memory access operators provided by SSE to facilitate the bulk memory transfers. Having hardware support for bulk memory scatter-gather could be an attractive alternative to this approach [30]. In addition to providing optimized memory transfers, SSE provides instructions that operate in SIMD mode on data elements using the vector-like registers. These operations could potentially be used to optimize computational kernels in the stream programming paradigm.

VI. Conclusions and Future Work

For CPUs with dynamically scheduled pipelines like the Pentium 4, cache miss latency can have a large impact on performance. Sophisticated logic in the processor attempts to speculatively execute ahead and discover cache misses in advance to mitigate these effects. Our experiments show that by extracting out memory references from the computation and using a hardware context to force feed these references into the memory system in parallel with the computation, we can utilize the hardware more effectively than is automatically extracted by the pipeline. Stream programming provides precisely this information needed to partition the memory and computation across the hardware contexts.

We hope that this work will stimulate enough interest to write more programs in streaming-style. Building an existing base of programs could create an evolutionary path for the deployment of new streaming architectures.

In the future, we would like to investigate in more detail the architectural enhancements to the existing general purpose processors that would improve the performance of stream applications. In addition, we would also like to exploit other forms of parallelism manifested by stream programming like thread-level parallelism.

References

[1] U. Kapasi, W. Dally, S. Rixner, J. Owens, and B. Khailany, "The Imagine stream processor," in *Proceedings of Int'l Conference on Computer Design*, Sep 2002.

[2] W. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J-H A., N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck, "Merrimac: Supercomputing with streams," in *Supercomputing (SC)*, Phoenix, Arizona, November 2003.

[3] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, Jae-Wook Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw microprocessor: a computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, pp. 25–35, March 2002.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," in *Proceedings of SIGGRAPH*, 2004.

[5] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. Meli, C. Leger, A. Lamb, J. Wong, H. Hoffman, D. Maze, and S. Amarasinghe, "A stream compiler for communication-exposed architectures," in *Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[6] J. Owens, U. Kapasi, P. Mattson, B. Towles, B. Serebrin, S. Rixner, and W. Dally, "Media processing applications on the Imagine stream processor," in *Proceedings of the IEEE Int'l Conference on Computer Design*, Sep 2002.

[7] H. P. Hofstee, "Power efficient processor architecture and the Cell processor," in *Proceedings of the 11th Int'l Symposium on High Performance Computer Architecture*, Feb 2005.

[8] T.J. Barth, "Simplified discontinuous Galerkin methods for systems of conservation laws with convex extension," in *Discontinuous Galerkin Methods*, vol. 11 of *Lecture Notes in Computational Science and Engineering*. Springer-Verlag, Heidelberg, 1999.

[9] M. Fatica, A. Jameson, and J. Alonso, "Streamflo: an Euler solver for streaming architectures," in *IAA Paper 2004-1090, 42nd Aerospace Sciences Meeting and Exhibit Conference*, Jan 2004.

[10] Y. Kallinderis and A. Vidwans, "Generic parallel adaptive-grid Navier-Stokes algorithm," *AIAA Journal*, vol. 32, 1994.

[11] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Int'l Conference on Compiler Construction*, Apr. 2002.

[12] I. Buck, "Brook Specification v0.2," merrimac.stanford.edu/brook/brookspec-v0.2.pdf, October 2003.

[13] P. Mattson, W. Thies, L. Hammond, and M. Vahey, "Streaming Virtual Machine specification 1.0," Tech. Rep., 2004, http://www.morphware.org.

[14] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, January 1987.

[15] F. Labonte, P. Mattson, I. Buck, C. Kozyrakis, and M. Horowitz, "The Stream Virtual Machine," in *Proceedings of the 2004 Int'l Conference on Parallel Architectures and Compilation Techniques*, France, September 2004.

[16] T. Chilimbi, M. Hill, and J. Larus, "Cache conscious structure layout," in *Programming Languages Design and Implementation*, May 1999.

[17] P. Mattson, *A Programming System for the Imagine Media Processor*, Ph.D. thesis, Stanford University, 2002.

[18] D. T. Marr, F. Binnas, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-threading technology architecture and microarchitecture," *Intel Technology Journal*, vol. 6, Issue 1, 2002.

[19] D. I. Pullin and D. J. Hill, "Computational methods for shock-driven turbulence and LES of the Richtmyer-Meshkov instability," *USNCCM*, 2003.

[20] Mode Piezoceramic Actuators, "Active constrained layer damping with extension and shear," citeseer.ist.psu.edu/641428.html.

[21] N. Goharian, T. El-Ghazawi, D. Grossman, and A. Chowdhury, "On the enhancements of a sparse matrix information retrieval approach," *Int'l Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA)*, 2000.

[22] "Gromacs," www.gromacs.org.

[23] A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine transforms," in *Proceedings of the Annual Symposium on Principles of Programming Languages*, 1997.

[24] C. Ding and K. Kennedy, "Improving effective bandwidth through compiler enhancement of global cache reuse," *Journal on Parallel and Distributed Computing*, 2004.

[25] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, "Improving locality using loop and data transformations in an integrated framework," in *Int'l Symposium on Microarchitecture*, 1998.

[26] D. Kim, S. Liao, P. Wang, J. Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. Shen, "Physical experimentation with prefetching helper threads on Intel's hyper-threaded processors," in *Int'l Symposium on Code Generation and Optimization*, March 2004.

[27] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," in *Proceedings of 28th Int'l Symposium on Computer Architecture*, July 2001.

[28] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *Fourth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Apr 1991.

[29] D. Ortega, M. Valero, and E. Ayguade, "A novel renaming mechanism that boosts software prefetching," in *Int'l Conference on Supercomputing*, 2001.

[30] J.B. Carter, W.C. Hsieh, L.B. Stoller, M.R. Swanson, L. Zhang, and S.A. McKee, "Impulse: Memory system support for scientific applications," *Journal of Scientific Programming*, vol. 7, 1999.