Appears in the proceedings of 16^{th} Int'l Conference on Parallel Architectures and Compilation Techniques (PACT), Sep '07

Architectural Support for the Stream Execution Model on General-Purpose Processors

Jayanth Gummaraju* Mattan Erez** Joel Coburn* Mendel Rosenblum* William J. Dally*

*Computer Systems Laboratory, Stanford University
**Electrical and Computer Engineering Department, The University of Texas at Austin

Abstract

There has recently been much interest in stream processing, both in industry (e.g., Cell, NVIDIA G80, ATI R580) and academia (e.g., Stanford Merrimac, MIT RAW), with stream programs becoming increasingly popular for both media and more general-purpose computing. Although a special style of programming called stream programming is needed to target these stream architectures, huge performance benefits can be achieved.

In this paper, we minimally add architectural features to commodity general-purpose processors (e.g., Intel/AMD) to efficiently support the stream execution model. We design the extensions to reuse existing components of the general-purpose processor hardware as much as possible by investigating low-cost modifications to the CPU caches, hardware prefetcher, and the execution core. With a less than 1% increase in die area along with judicious use of a software runtime system, we show that we can efficiently support stream programming on traditional processor cores. We evaluate our techniques by running scientific applications on a cycle-level simulation system. The results show that our system executes stream programs as efficiently as possible, limited only by the ALU performance and the memory bandwidth needed to feed the ALUs.

1 Introduction

Recently there has been much interest in both research and the commercial marketplace for architectures that support a stream-style of execution [15, 17, 23, 2, 6]. Although initially targeted at applications such as signal processing that operate on continuous streams of data, stream programming has broadened to encompass general compute intensive applications. Research has shown that stream architectures such as Stanford Merrimac [15], Cell Broadband Engine (Cell) [17], and general-purpose computing on graphic processing units (GP-GPUs) [11] deliver superior performance for applications that can exploit the high bandwidth and large numbers of functional units offered by these architectures.

Stream processors (SP) require a different programming abstraction from traditional general-purpose processors (GPPs). To get performance benefits, stream processors are programmed in a style that involves bulk loading of data into a local memory, operating on the data in parallel, and bulk storing of the data back into memory. This style of programming is key to the high efficiency demonstrated by SPs.

Although current multicore GPPs such as those from Intel and AMD, lack the peak FLOPS and bandwidth of stream processors, their likely ubiquitous deployment as part of industry standard computing platforms make them an attractive target for stream programming. It is desirable to effectively use these commodity general-purpose multicores rather than targeting only special purpose stream-only processors such as Cell or GP-GPUs.

One problem with this approach is that although peak FLOPS and memory bandwidth of general-purpose processors are improving and narrowing the gap between them and stream processors, GPPs lack some of the features that are key to the high-efficiency of stream processors. In this paper we examine these differences in detail, and propose and evaluate extensions to a general-purpose core to allow it to efficiently map the stream programming style. To simplify our discussion we focus on single core behavior but operate under the assumption that the core is part of a multicore system used in a streaming style.

Our work shows that although stream cores and general-purpose cores appear very different to the programmer, the underlying implementations are similar enough that only relatively minor architectural extensions are needed to map stream programs efficiently. Our basic approach is to examine the key features of stream processors such as Cell and Merrimac and determine how to best emulate them on a general-purpose core. By using some architectural features in an unintended way (e.g. treating a processor cache as a software managed local memory) and judiciously using a software runtime, we found that the only required architectural extension is a memory transfer engine to asynchronously bulk load and store operand data to and from the local memory.

In this paper we describe the design of the memory transfer engine we call the stream load/store unit (SLS unit). The SLS unit can be logically viewed as an extension and generalization of a traditional hardware memory prefetch and writeback unit that is able to transfer large groups of potentially non-contiguous memory locations to and from the cache memory. We also show how the SLS unit aligns data before being transferred to the cache so that it can directly feed into short-vector SIMD units such as the SSE units of an x86 processor. We claim that the SLS unit is a relatively minor extension leveraging much of the existing functionality and data-paths in a general-purpose core, requiring less than 1% increase in die area.

We show that our extensions allow a traditional GPP core to efficiently execute the stream programming model. This means that performance will be limited by either the operation rate of the ALUs (peak FLOPS) or memory bandwidth needed to fetch the operands depending on if the application is compute or memory bound. We demonstrate this with four real scientific applications that have been coded in a stream style. We also show the potential improvement we get over the same program written in a conventional style run on the same GPP.

The paper is organized as follows. We start by comparing and contrasting the architectures used for traditional GPPs and the new breed of architectures for stream computing in Section 2. In Section 3 we show how stream programs can be mapped onto the GPP core by focusing on the SLS unit extension. In Section 4 we present the evaluation of our extensions using simulation. We present additional related work in Section 5, and conclude in Section 6.

2 General-Purpose and Stream Programming and Architectures

In this section we compare and contrast the programming model and micro-architecture of two different architecture classes: a general-purpose processor (GPP) architecture and a stream processor (SP) architecture. While the former is optimized to run applications written in conventional von Neumann style where the parallelism and data locality is automatically extracted from sequential code, the latter is optimized to run applications written in a stream-style where both parallelism and data locality are explicitly expressed by the programmer.

2.1 General-Purpose Programming and Architecture

The programming model typically used for GPPs is exhibited by the familiar sequential languages (e.g., C, FORTRAN, Java). Conceptually, instructions execute sequentially and in program order, often with frequent control transfers. Requests to memory are performed on a per-use basis resulting in memory accesses that are of single-word granularity. This programming model is most suited for applications that have fine-grained control and uncertainty both in control flow and data accesses.

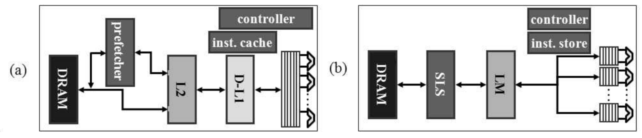


Figure 1. Sketch of canonical (a) GPP core (b) SP core

Figure 1(a) depicts an abstraction of a GPP core and the memory subsystem showing the key elements involved in fetching and processing instructions/data. This abstraction is useful in drawing parallels with a stream micro-architecture (Figure 1(b)) to be discussed in the next section.

The controller is responsible for fetching instructions in control-flow order and extracting parallelism from the instruction sequence. Because the GPP needs to support arbitrary instruction sequences that potentially have little explicit parallelism, it uses several speculative hardware structures and static/dynamic scheduling to extract parallelism and drive multiple functional units. Along with conventional functional units such as ALUs/FPUs, modern processors typically feature short-vector SIMD units (e.g., SSE/Altivec/VMX units), which substantially increase the compute power of GPPs. However, the utilization of these units is usually low because it is difficult to automatically generate code to feed these units efficiently.

Global memory accesses can have immediate effect on subsequent instructions in this programming model, so great emphasis is placed on a storage hierarchy that minimizes data access latencies rather than increasing data bandwidth. The storage hierarchy is composed of the central register file at the lowest level, followed by multiple levels of cache memories. Caches work well for most control intensive applications that access a limited working set of data, but compute and data intensive applications need additional hardware structures and software techniques. A hardware prefetcher is one such structure which attempts to predict and prefetch data using the data access pattern. If the prediction is both timely and correct, the memory access latency is completely hidden.

2.2 Stream Programming and Architecture

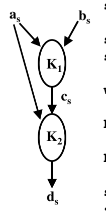
Stream programing, on the other hand, provides an efficient style to represent compute or memory intensive applications that have large amounts of data-parallelism, that are less control-intensive, and that have memory accesses that can be determined well in advance of the data use. The computation is decoupled from memory accesses to enable efficient utilization of computation units and memory bandwidth.

Although originally intended for applications that follow restricted, synchronous data flow, stream programming has recently been shown to work well for more general applications (e.g., irregular scientific applications) [32, 15]. Several software systems have been created to support the development and compilation of stream programs for stream processors. (e.g., Brook [11], Sequoia [18], StreamIt [31]).

In the stream programming model complex *kernel* operations execute on collections of data elements referred to as *streams*. Kernels are programmer defined procedures of arbitrary complexity (typically several hundred operations), and stream elements are records with multiple data fields (typically tens of bytes/record).

The stream programming model advocates a *gather–compute–scatter* style of programming[1]. Data is *gathered* in bulk from arbitrary memory locations in main memory (MM) into a local memory (LM). This involves an asynchronous

[1]Note that only the style of programming and execution changes. We can continue to use existing sequential languages (e.g, C, Fortran) with a few additional library calls for the bulk memory operations.



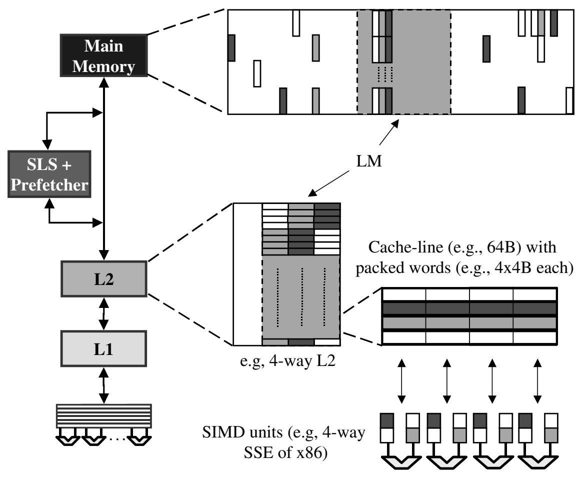
```
for: i = 0 … numStrips - 1
{
  streamGatherSeq (a_s, a, hdl_1, …); // a_s[j] ← a[j]

  streamGatherSeq (idx1_s, idx1, hdl_2, …); // idx1_s[j] ← idx1[j]
  streamGatherIdx (b_s, b, idx1_s, hdl_3, …); // b_s[j] ← b[idx1_s[j]]

  wait (hdl_1, hdl_2, hdl_3); // synchronization routine

  K_1 (a_s, b_s, c_s);

  K_2 (c_s, a_s, d_s);

  streamGatherSeq (idx2_s, idx2, hdl_4, …);
  streamScatterIdx (d_s, d, idx2_s, hdl_5, …); //d[idx2[j]] ← d_s[j]
}
```

Figure 2. Example stream code after stream compilation. Kernels K_1 and K_2 operate on gathered input stream data a_s, b_s, and c_s, and produce output d_s scattered back to memory.

copy of data from the MM address space to the LM address space. Computation *kernels* directly operate on the stream data from LM and the *produced* results are stored back into the LM. The *consumer* kernels use these results during execution and store their results back to LM, and so on. Finally, only the live data from the LM is *scattered* back in bulk to main memory. The data transfer between address spaces involves explicit *copying*, and hence, in programming language terms the parameters to the kernel are passed-by-value, as opposed to a pass-by-reference approach where pointers to the scalar data would be passed. Passing parameters by value eliminates aliasing between LM and MM data and enables data arrangement in LM for feeding SP ALUs.

In order to map a stream program onto a stream processor several simple transformations are performed by a stream compiler. Streams are broken down into strips, each typically several thousand bytes long, to insure that the working set of strips fits in the LM. The strips 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. The compiler also inserts synchronization routines between asynchronous bulk memory operations/kernels (Figure 2). Finally, a run-time system schedules these operations on the hardware. A more detailed description is available in [22, 16].

The stream processor architecture is designed specifically to exploit the stream execution model, and hence, is considerably different from the canonical GPP architecture. Analogous to the GPP core in Figure 1(a), Figure 1(b) shows an abstraction of a canonical SP core and the memory subsystem hierarchy depicting the key structures involved in data/instruction fetching and processing. The control structure, which is much simpler than that of the GPP, fetches statically scheduled instructions into the instruction store. The SP decouples the non-deterministic off-chip accesses from the execution pipeline by allowing the functional units to only access explicitly software managed on-chip LM. This LM (FIFO buffers in RAW [23], and more flexibly addressable local memory in Cell and Merrimac [15]) serves as a staging area for the bulk memory operations and has an address space different from the MM address space, matching the programming model. The functional units can directly address the locations in the LM namespace, making memory reference latencies short and predictable. The functional



Main Memory
SLS + Prefetcher
L2
L1
LM
Cache-line (e.g., 64B) with packed words (e.g., 4x4B each)
e.g, 4-way L2
SIMD units (e.g, 4-way SSE of x86)

Figure 3. GPP architecture with stream extensions

units also have a large number of local registers to store intermediate values. External memory accesses are performed in bulk by asynchronous SLS units. Thus, the memory latency problem in GPPs is transformed into a memory bandwidth problem in SPs.

3 Stream Extensions for General Purpose Processors

As discussed in the previous section, SP and GPP architectures differ significantly in their implementation. However, as can be seen from Figure 1 the overall structure is similar, with the major components serving equivalent purposes in both architectural styles. In this section we show how to exploit these similarities and extend the GPP architecture to support stream programs.

Our architecture is depicted in Figure 3 and described in detail in the subsections below. Our overall strategy for mapping a SP onto a GPP is to either minimally extend the hardware capabilities, or use software to emulate the stream functionalities on the existing hardware structures. The resultant mapping is as follows: part of the second-level (L2) cache serves as the LM; the hardware prefetcher is extended to support (a) programmable asynchronous bulk memory transfers of the SLS unit and (b) hardware packing of data for short vector SIMD units (e.g., SSE units of Intel); the execution pipeline with its short vector units and tightly integrated first-level (L1) cache supplies the parallel execution substrate; the GPP instruction cache and controller serve as the stream instruction store and control; and the memory controller and DRAM interfaces remain unmodified.

Our extensions are designed for a single core of a multicore GPP with private L1 caches sharing a common L2 cache (other configurations are left for future work). A multicore processor requires a run-time system to judiciously schedule and partition work across the processor cores, similar to hardware score-boarding used in SPs such as Merrimac [15]. We focus on optimizing the performance for a single core in such an environment.

3.1 Software Managed Local Memory

Modern GPPs contain large on-chip memories in the cache hierarchy, and these can be used as the stream LM as shown in Figure 3. Our architecture uses an unmodified GPP cache hierarchy, with a portion of the L2 cache acting as the LM following the methodology of [16]. Relying on the large capacity and high associativity of modern GPP L2 caches allows us to map the LM to a portion of the MM address space with little modification to the hardware structures, therefore addressing the issue of having a separate address space for the LM. Conceptually, most *ways* in each set of the L2 cache are used for the LM, and the remaining ways are available for non-LM data such as instructions, kernel local variables, and global variables.

We prevent the automatic eviction of LM data in L2 cache using the cache-control bits typically available in processor caches (e.g., non-temporal bits set by prefetchnta/movntq in x86, DLOCK bit set in PowerPCs). It is important to note that even if LM data get evicted from the cache, correctness is not affected.

We chose to use the L2 cache, rather than the L1 data cache as the LM because the L1 is tightly integrated with the execution pipeline and has a relatively small capacity and associativity compared to the L2 cache.

Alternative design options require extensive modifications to the cache structure, either by adding a dedicated software controlled LM, or by using hybrid reconfigurable structures. Adding a separate LM reduces the effective die area available for the cache, thus affecting the performance of GPP codes which typically do not use the LM. Hybrid software/hardware managed on-chip memory [24, 20] make the cache design more complicated by adding new data paths and control logic.

3.2 Asynchronous Bulk Memory Transfers

SPs feature asynchronous units (SLS units) to transfer data in bulk between the main memory and the LM. Such bulk transfers of data are essential to break the von Neumann bottleneck of single word loads and stores. Bulk transfer primitives such as stream gather and scatter are provided to directly program these units. Hence, for GPPs to effectively execute stream programs it is imperative to support a structure analogous to the SLS unit.

To accomplish this goal, our key insight is that the hardware prefetcher in GPPs, like the SLS unit of a SP, asynchronously transfers data in bulk between on-chip memory and off-chip DRAM. However, the streaming SLS unit in SPs can be explicitly programmed using predetermined access modes, whereas the hardware prefetcher is speculation-based and cannot be controlled by the programmer. Therefore, we decided to augment the hardware structures of the prefetcher to allow for explicit and programmable software control of bulk memory transfers. This unit performs efficient transfer of data to/from LM without affecting the execution core, optimizes for memory bus and DRAM bandwidth, and packs data efficiently for short-vector SIMD units.

Figure 4 shows a diagram of our hybrid SLS/prefetcher unit. Shaded components in the figure were added to support explicit asynchronous bulk transfers, and we estimate their hardware cost to be $0.8mm^2$ in a current $65nm$ process based on the area models and physical implementation of the Imagine Stream Processor [7]. This small area overhead is less than a 1% increase in a $85mm^2$ GPP core die area.

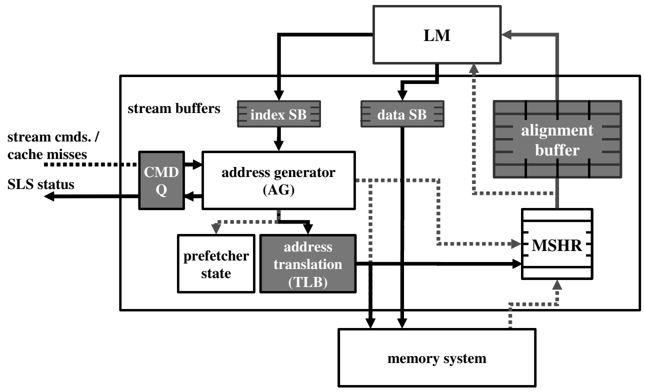


Figure 4. The SLS unit. Components augmenting the prefetcher are shaded, and the added datapaths appear as solid lines. These include control and finite state machine structures for generating accesses, stream buffers (SB) and alignment buffers (ABs) for transferring data to/from the LM, MSHRs (Miss Status Holding Registers for outstanding misses) for communicating with MM, and a SLS TLB for address translation.

3.2.1 Communicating Control Information

The execution core and SLS unit communicate control information twice for each bulk stream memory transfer, once to set up and launch the SLS operation and again to synchronize with the execution core on SLS completion.

The execution core communicates with the SLS unit using an optimized memory-mapped I/O interface. The arguments corresponding to an SLS operation (e.g, stream start address, numElmts, etc.) are transmitted using multiple store instructions to a special address. Once all the arguments arrive the SLS unit enqueues the operation. The operation is launched when there are no outstanding dependencies with other SLS and kernel operations. When a SLS operation completes, the SLS command queue is signalled so that a new SLS operation can be launched.

The execution of a kernel operation and/or SLS operation may be dependent on each other. Therefore, upon completion the execution core and/or the SLS unit update hardware status registers which are checked before executing.

3.2.2 Programmable Memory Access Generation

Our SLS architecture supports five memory access modes useful for stream programming [11, 22]: *strided gather, strided scatter, indexed gather, indexed scatter,* and *scatter-add*. In addition, the SLS unit also supports a *hardware prefetch* mode which is triggered based on the prefetcher state and incoming cache misses.

A strided gather copies a stream of regularly spaced off-chip memory locations to a contiguous block in the LM. Our SLS implements a 1.5-dimensional access pattern that expresses a sequence of fixed-size contiguous word blocks (records) that are spaced at an interval of stride words. A strided scatter performs the converse operation of copying a contiguous block from the LM to a potentially sparse off-chip memory range. Indexed gathers and scatters perform similar operations except that the interval between records is variable and is supplied as a stream of absolute indexes that are read from the LM. Scatter-add [4] is an atomic read-modify-write scatter that accelerates super-position type reductions common to stream applications in the scientific domain.

The *address generator* (AG) in the SLS unit produces a stream of single-word virtual addresses according to the access mode. The AG can interleave accesses from multiple consecutive records enabling the hardware to perform alignment for short-vector execution (see Section 3.3). The AG generates as many memory addresses every cycle as required to saturate the DRAM bandwidth – 2 addresses per cycle in the case of a 2GHz processor with a 6.4GB/s DDR2 or a 10.6GB/s DDR3 interface. Each virtual address generated must undergo physical address translation before being sent to the memory system. Since the AG throughput must remain high, we provide a TLB unit within the SLS hardware. This TLB operates on pages with very large granularity [30] to allow for high throughput with inexpensive hardware. The SLS TLB need not have the same entries as the execution core TLB, and it can autonomously query the OS page table.

3.2.3 LM/SLS/Main Memory Data Transfer

We carefully manage the transfer of data between LM, SLS, and MM to make the optimal use of each data-path and handle any coherence issues that arise between the LM/MM data. The data transferred between LM and MM is staged in the SLS unit before copying to the destination memory. This intermediate step provides several opportunities for collecting, and possibly re-arranging data for maximum utilization of both the datapath and the destination memory.

To understand the flow of data between SLS and the memories, consider an *indexed gather* operation. The index stream is first loaded from the LM to the index stream buffer in the SLS unit. Using these indexes and the base address of the data array (specified when programming the SLS call), the AG generates a series of addresses and sends requests to the MM. Once the responses arrive the data is collected in the SLS before writing to the LM. *Indexed scatter* works similarly except that the data is read from the LM and written to the MM. *Strided gathers/scatters* also have a similar data flow except that indexes are no longer required.

Transferring data between the SLS and LM requires three logical ports to the LM: one port for writing data into the LM, a second port for reading data from the LM, and a third to read index values for gathers and scatters. Since the LM is implemented within the L2 cache, we share the single physical port to L2 and time-multiplex it with dedicated buffers for the three logical ports. Hence, whenever data is transferred to/from LM, the SLS performs an arbitration for the L2 port. In addition, while transferring data to the LM the SLS sets the cache-control bits of the corresponding cache line to prevent the automatic eviction of LM data.

We perform several optimizations for efficient transfers between SLS and LM. Using SLS buffers (stream/index SBs) the data is transferred at the granularity of the L2 cache line size (typically 64-128 bytes), ensuring that the full width of the L2 physical port is used. We limit the number of requests generated by the AG (2 words/cycle) such that the L2 port is only accessed once every 4-8 cycles, which leaves sufficient bandwidth in the L2 port for feeding the execution core. To minimize core stalls on L1 cache misses, we give higher priority to demand fills from the core. The SLS overlaps the address/request generation to the memory system with the transfer of data from/to LM for the following cache line by pipelining the SLS operations into stages. Finally, we use alignment buffers (ABs) to pack the data into long words for efficiently feeding the short-vector SIMD units of the execution core. This is key to achieving good performance and is discussed in detail in the next section.

Transferring data between the SLS and main memory provides several opportunities for optimizing memory bandwidth. Since data for memory read operations are returned from the memory system at a granularity of a DRAM burst, we match the SLS request size to the DRAM burst size (e.g., 32 Bytes for DDR2). We accomplish this using MSHRs (Miss Status Holding Registers) which collate all requests going to the same DRAM burst into one request to the main memory. This operation is highly effective when there is locality between requests and this is fairly common because each element of the stream is usually several bytes long. We study the effects of burst-size in Section 4. Additionally, the combination of MSHRs and alignment buffers provides ample space for reordering memory operations to maximize DRAM throughput [27]. The depth of the AB and number of MSHRs can grow quite large since they must hold enough accesses to saturate the DRAM bandwidth ($DRAM_{latency} \times DRAM_{bandwidth}$) as well as extra buffering for reordering. In the case of a 2GHz processor core with a 6.4GB/s DDR2 memory system we use a 512-byte AB and 256 MSHRs and estimate their area at $0.4mm^2$ (0.5% of a typical $85mm^2$ GPP die).

Maintaining coherence and consistency between LM/MM data is usually straightforward because of the stream program semantics described in Section 2.2. As an artifact of the stream programming model, data is *copied* between the address spaces and the LM/MM data do not alias each other. However, coherence could potentially be violated during SLS transfers to/from LM if the main memory does not have the most recent value for the MM data arrays. The SLS unit maintains coherence by leveraging the coherence protocol used by the hardware prefetcher for the underlying multi-core GPP. It first checks for recent updates in the L2 tag array and/or the L1 tag arrays (write-back)/L1 write buffers (write-through) before requesting data from main memory. However, this scenario is uncommon because the data-sets are huge (much bigger than the L2 cache size) and most of the cache is used for the LM.

An alternative to a hardware SLS unit is to emulate the SLS functionality in software either using one thread of an SMT capable GPP [16], or a processor core of a multi-core GPP. The major disadvantage of this software approach is that a full hardware execution context is dedicated to performing SLS transfers and cannot be used for additional computation. Also, software emulation requires sharing the instruction fetch and execution bandwidths between the actual computation and the SLS, potentially reducing the performance of compute-intensive stream applications.

3.3 Parallel Execution Using Short Vector SIMD Units

The SLS unit ensures that all the data needed by a computation kernel is present in the LM prior to its execution. This enables the kernels to execute at maximum rate, limited only by the computation resources of the GPP core. In addition to ALUs/FPUs, modern GPP cores feature short-vector SIMD units (e.g, Intel SSE, AMD 3DNow!, IBM VMX) which constitute the bulk of the compute power. To achieve the highest performance on these SIMD units, we use the SLS unit to also pack and align the data in memory so it can be directly fetched into the SIMD registers. For modern SIMD units, this involves placing 4 single-precision or 2 double-precision floating-point words into a single 128-bit location of the LM.

To limit the need for software padding or packing instructions operating on the LM, the SLS unit's address generator issues requests in an order that ensures that the alignment buffer collects the same field of two or four consecutive records into a single 128-bit LM location. The result is that fields from different records are packed together and can be fetched into the SIMD unit using a single instruction such as the `movaps` SSE instruction.

Arranging data to align to the SIMD boundaries is a significant issue in the stream programming model because each stream element is typically a record with multiple fields and cannot be directly loaded into the SIMD unit. Figure 5 illustrates the packing and alignment problems normally solved in software when using SSE instructions. When indirect (gather) record accesses are performed, software must pad the data in memory to allow 128-bit aligned accesses as well as dynamically pack the gathered data into short-vector registers. In contrast, Figure 6 shows the stream implementation of the same sample code using our SLS architecture. Our architecture's hardware support eliminates the need for special padding or packing instructions. Even greater benefits could be realized should wider SIMD units become available.

Note that applications expressed in the stream style make it easy to exploit the SIMD units because memory accesses and computation are decoupled. The bulk operations naturally expose the parallelism to the SIMD units. Due to dependencies across loop iterations, horizontal computations, data-dependent control flow, and non-contiguous data layout, compilers for GPPs often have difficulties using these units. Most programs that use SIMD units are explicitly coded in assembly language or use compiler intrinsics.

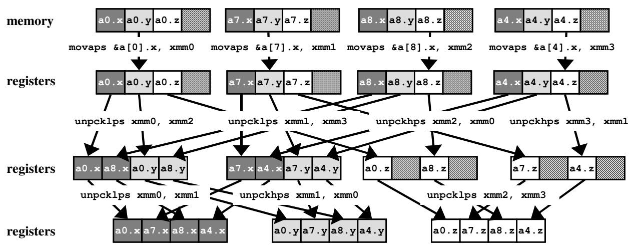


Figure 5. Short vector packing and alignment in software.

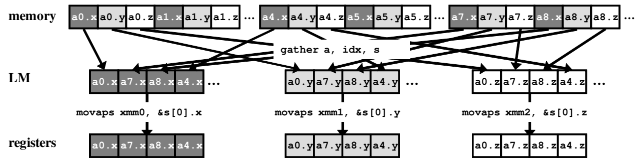


Figure 6. Short vector packing and alignment using streaming hardware SLS mechanism.

4 Evaluation

In this section we evaluate our proposed stream extensions to the GPP architecture. We focus our evaluation on a single core of a multicore GPP running stream programs. Our analysis shows that the stream processing extensions we propose efficiently support the stream execution model by providing high utilization of memory and compute resources similar to canonical SPs. This results in optimal use of the short-vector SIMD units, leading to significant speedups over the conventional implementations.

4.1 Experimental Setup

We evaluate four scientific applications that feature regular and irregular mesh constructs and linear algebra operations, which are common in scientific applications. The applications were originally written by programmers specializing in their specific application domains (fluid dynamics and solid mechanics). The four applications and characteristics of the datasets are summarized in Table 1, and further details are available in [16]. These applications display several challenging characteristics including non-affine and data-dependent array references, and a wide range of compute to memory ratios.

The overall evaluation methodology is as follows. The C/Fortran conventional implementations are first re-written in a stream-programming style [11]. We transform the stream program using standard stream compiler transformations discussed in Section 2.2 into stream code similar to Figure 2. Both the conventional and stream codes are compiled using alpha-gcc 2.95.3 with -O3 level optimizations to generate optimized alpha binaries. These binaries are then run on the simulation system described below to collect execution statistics.

We modified the M5 simulator [10] to reflect our architectural modifications described in Section 3. M5 is a cycle-accurate simulator of a modern GPP and was configured to

Application	Description
FEM [8]	2D Discontinuous Galerkin finite element method code for fluid dynamics. It uses a 4 816 element unstructured mesh, and solves for either the *Euler* or *magnetohydrodynamics* (MHD) equations. The code can also be parametrized for linear (lin), quadratic (qd), or cubic (cub) interpolation. FEM performs mostly gather and scatter memory operations of records spanning $5-80$ words and has three compute-intensive kernels.
CDP [19]	3D *large eddy* fluid dynamic finite volume method simulation on an irregular mesh. The *elmts_nb* and *finer_nb* datasets have a mix of tetrahedrons, prisms, pyramids, and cubic elements with 3 800 and 29 095 total control volumes. The *amr_nb* dataset has 5 416 cubic elements that have a connectivity of $2-8$ due to adaptive mesh refinement. CDP performs gather, scatter, and scatter-add memory operations to records of $1-8$ words and has four kernels.
SPAS [33]	Part of a sparse algebra suite; computes a compressed sparse row matrix vector multiplication on a 9 978, 19 094, 37 918, or 73 053 row matrix. SPAS uses unit-stride stream loads and stores and has one main kernel.
NEO [9]	A *neo-hookean* solid mechanics code that models a finite elasticity compressible material. The application uses a structured grid with 30 000, 50 000, 100 000, or 200 000 elements. NEO uses unit-stride stream loads and stores and has five kernels.

Table 1. Application and dataset description.

execute the Alpha instruction set in system-call emulation (SE) mode. Additionally, since an accurate simulation of the DRAM throughput is critical for stream programs, we augmented M5 by integrating the DRAMsim DRAM simulator [34]. The baseline machine parameters reflecting a modern GPP processor are detailed in Table 2. For both the stream and conventional programs we prefetch subsequent lines for an instruction miss in the L2 cache. In addition, for the conventional programs we use an aggressive stream-based hardware prefetcher which prefetches four subsequent lines on every data miss in the L2 cache.

Parameter	Value	Parameter	Value
Core frequency	2GHz	DRAM architecture	DDR2
Pipeline	OOO	DRAM bandwidth	6.4GB/s
L2 size	1MB	DRAM burst	32/64 bytes
L2 associativity	8	SLS AG bandwidth	2addr/cycle
L2 line size	64 bytes	SLS ABs	8 (512 bytes)
FSB bandwidth	6.4GB/s	SLS MSHRs	256

Table 2. Baseline machine parameters.

Since the M5 simulator and Alpha ISA do not support short-vector SIMD execution, we measured the performance and hardware alignment benefits using real Pentium 4 hardware with SSE3 instructions within kernels. We also studied the impact on overall performance by simulating the varying degrees of SIMD unit utilization using the M5 simulator.

4.2 Stream Execution

In this section we show that our extensions enable the GPP core to behave similar to a canonical SP core by efficiently overlapping computation with memory accesses and achieving high memory throughput limited only by the DRAM architecture.

Figure 7 shows the fraction of the total execution time during which kernels are running on the execution pipeline assuming full SIMD utilization (Section 4.3)(% Kernel) and the fraction of time spent performing memory accesses for the DDR2-based memory systems (% SLS). Either % Kernel or % SLS should be 100% to indicate complete overlap. The figure also shows the percentage of peak memory bandwidth utilization, on average, in each of the benchmarks.

Overall, we see that similar to canonical SPs, there is almost full overlap of kernel computation and SLS memory operations. When the applications are compute bound (FEM and NEO), all the time is spent performing useful computation in the kernels, and the time spent in SLS operations is completely hidden. Similarly, when the applications are memory bound (CDP and SPAS), all the time is spent in SLS memory operations fetching only non-speculative data, and the computation time is completely overlapped. Also notice that the memory bound applications achieve very high fractions of peak memory bandwidth (CDP: 80% of peak at 5.1GB/s, SPAS: 94% of peak at 6.1GB/s). Although the overall memory throughput of compute bound applications is much lower, high throughput is achieved while transferring the data (e.g, FEM: 4.6GB/s during data transfer which occurs for 15% of total run-time).

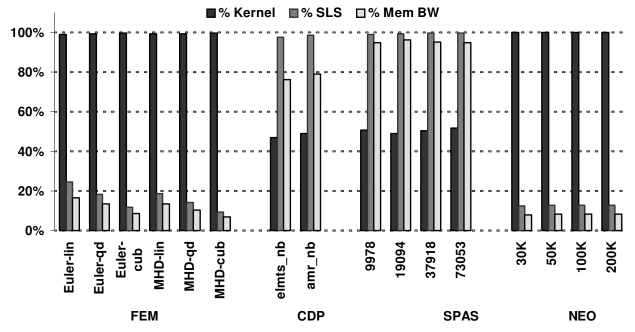


Figure 7. Compute and memory resources utilization.

Although we achieve high memory bandwidth utilization using a SLS unit, not all data transferred is useful data due to the DRAM architecture and the data-access patterns. To study these effects, we ran four different SLS calls corresponding to different data access patterns – unit-stride load, indexed gather, unit-stride store, and indexed scatter. While the stream load and store requested consecutive 4-byte words from memory, gather and scatter requested 12-byte (3-word) records randomly from a large array.

Figure 8 shows the measured memory throughput for DDR2-32, DDR2-64, and DDR3-64, where 32 and 64 refer to the minimum DRAM transaction size (burst size). We measured the throughput for different numbers of outstanding requests to memory, which are controlled by varying the number of alignment buffers.

We see that the memory bandwidth utilization for sequential memory accesses is much higher than that for random memory accesses (Figure 8). In fact, using just 8 ABs we achieve very close to the theoretical peak bandwidths of 6.4GB/s (DDR2) and 10.6GB/s (DDR3) for sequential reads. When the memory accesses are non-sequential the memory bandwidth utilization drops to about $1/6$th to $1/8$ the peak bandwidth. This is primarily because modern DRAM sys-

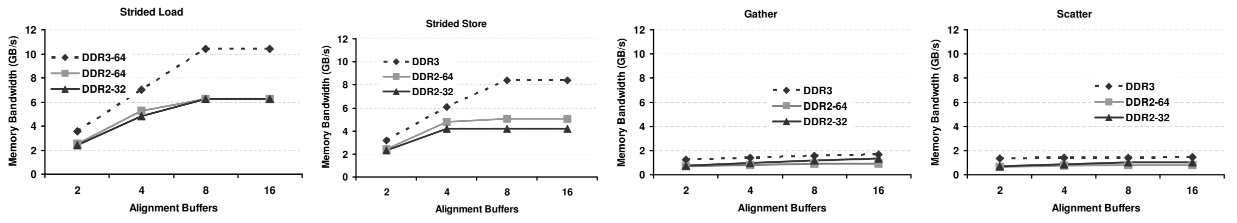


Figure 8. Sensitivity of memory bandwidth performance for stream memory operations to AB depth and DRAM architecture.

tems are optimized for sequential memory accesses, sometimes at the expense of random accesses. For example, the minimum burst size in DDR3 is 64 bytes, indicating that at least 64 bytes are transmitted even if the requested number of bytes is much lower. These experiments with stream applications illustrate the need for efficient support for random memory accesses in DRAM technologies [5] which is an issue not specific to our proposed architecture. In a multicore framework, these effects could be further exacerbated if multiple SLS units issue requests to the DRAM simultaneously.

4.3 Effects of SIMD Execution

The conventional codes we used in this study are unable to benefit from the SIMD unit present on most GPPs. Both the open-source and commercial compilers available to us were unable to generate short-vector instructions for the loops in our applications. A majority of our codes contain non-affine accesses and data dependent gathers and scatters, thus making it difficult to automatically generate short-vector SIMD instructions.

To evaluate our claim that the gather/scatter style of stream computation takes better advantage of short vector instructions, we performed two experiments. First, we studied the benefits we could achieve using the current x86's SSE3 extensions on several important kernels from our streamified applications. Second, we studied the effects of a more capable execution substrate on overall performance by simulating a system with varying degrees of SIMD unit utilization.

Table 3 lists the computation kernels from the stream code we hand-converted to SIMD format for the Pentium 4 architecture which supports SSE3. Our methodology involves two steps. First, data is streamed and packed into the LM such that it is aligned to the SIMD width. Second, we emulate a trivial compiler pass where loops are simply unrolled four times to use 4-wide SIMD operations on the densely packed stream data arrays. We measured the potential benefits of hardware alignment by comparing the execution times of two versions of the kernels – one version aligns data inside the kernel using SSE alignment instructions (software alignment) and the second version aligns data prior to the execution of the kernel, representative of using the SLS unit (hardware alignment). Our converted routines were compiled with Intel's ICC compiler and the performance was measured on a 3.4GHz Intel Prescott core.

Figure 9 shows the performance of five kernels using SSE instructions, with both software and hardware alignment, relative to the baseline implementation with no SSE instructions. The results indicate that even with trivial compiler support many kernels show significant speedups ($\sim$ 3.4x on average) using SSE. The main causes of speedups lower than the expected 4x were inefficient kernel code, and resource restrictions in the Pentium 4 SSE hardware. For example, *GatherFlux* and *GatherCell* contain several unaligned table lookups even though the input data stream was aligned and packed for the SIMD unit. It is possible to optimize these kernels and attain significant performance improvements with better compiler analyses than our evaluation methodology assumed. In some cases, minor code modifications were required to use SSE. This was the case for *InitRes* and *ComputeMax* which indirectly improved speedup to over 4x.

App	Kernel	Description
FEM	GatherCell	Computes on cells, gathering data from neighbors.
FEM	GatherFlux	Computes on faces, gathering data from neighbors.
CDP	InitRes	Computes residuals based on neighbor information.
CDP	Face	Updates faces using neighboring control volumes.
CDP	CompMax	Reduces residuals at end of each iteration.

Table 3. Kernels evaluated on Pentium 4.

Our hardware-assisted alignment and packing shows performance improvements of 13% on average and up to 22% over software alignment using packing instructions. Performing alignment and packing in the SLS unit frees up processor resources (SIMD registers, load/store unit, cache space) and better utilizes functional units for increased performance. This is because alignment in software requires packing instructions which reduce the utilization of the SIMD unit. The alignment using the SLS unit, on the other hand, is performed while assembling the data in the alignment buffers and hence, has no additional overheads. Kernels such as *Face* see a substantial benefit because they operate on data elements across loop iterations and therefore perform frequent vector gathers. Conversely, kernels that access sequential aligned data, such as *ComputeMax*, see no improvement from this hardware mechanism.

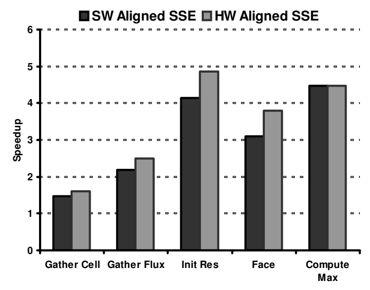
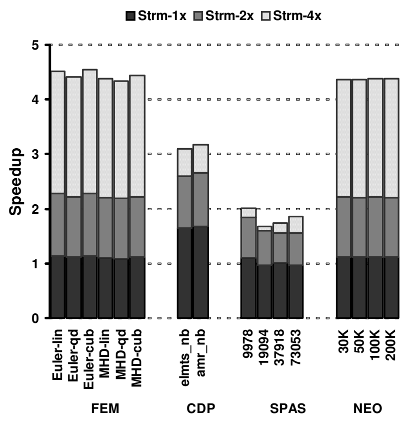


Figure 9. Speedups due to SSE instructions

To study the effects of a more capable execution substrate on *overall* performance, we simulated a system with



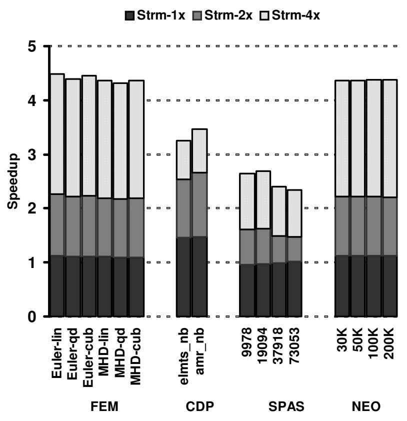


Figure 10. Range of speedups of stream programs over conventional programs depending on the effectiveness of SIMD hardware using (a) DDR2 and (b) DDR3 DRAM. The conventional codes do not utilize the SIMD unit.

the assumption of faster kernel executions (because alpha ISA lacks short-vector support). Kernels are run faster by performing fewer loop iterations within the kernel. Memory traffic is simulated in full using the SLS unit. Figure 10 shows the speedups of stream programs over the conventional programs with varying utilization of the SIMD unit for the stream programs, and varying bandwidth/latency of the DRAM system. The faster kernel execution with SIMD units results in significant speedup for compute-bound programs but less speedup as programs become memory-limited.

To study the sensitivity of overall application performance to the utilization of the SIMD units, we look at the range of speedups we could obtain if the SIMD unit is 0%, 50%, and 100% utilized. The utilization varies depending on the amount of control and integer code within the kernels, and the limitations in hardware/compiler as discussed above. We measure the execution speedup of the applications assuming kernel speedups of 1x, 2x, and 4x, corresponding to the SIMD utilizations, and also vary the memory bandwidth using DDR2 and DDR3 memory systems (Figure 10).

For the scenario where a SIMD unit is not used (i.e., SIMD utilization is 0%), indicated by Str-1x in the figure, we see total run time speedups of about 0.96-1.8x (1.2x on average)[2] using DDR2. The speedup obtained is due to improved utilization of the memory bandwidth by the SLS unit and effective latency hiding of the stream model. Therefore, applications which are heavily compute-bound do not see significant speedups without SIMD execution. The minor ($< 5\%$) slowdown observed in a few of the SPAS datasets is a result of the effective hardware prefetching of the sequential loads in the GPP case and the small overhead of synchronization between SLS calls and the execution pipeline in the stream case. We conclude that in the stream case, the SLS engine is able to better exploit the lower available memory bandwidth of DDR2, and hence, increases pressure on compute resources. Notice that when bandwidth is increased (DDR3), even conventional codes begin to perform better. Therefore, we see lower speedups compared to DDR2 in the Str-1x case.

When the SIMD unit utilization increases to 50% and 100% (2x and 4x kernel speedups), the performance of the compute-bound FEM and NEO applications grows linearly achieving a maximum speedup of almost 4.5x over the conventional implementation. The performance of CDP and SPAS does not scale as well because memory bandwidth limits their performance. However, as computation throughput increases memory throughput becomes the performance bottleneck. Therefore, we observe higher speedups in CDP and SPAS for the higher bandwidth DDR3 when the SIMD units are better utilized (Str-2x, Str-4x).

5 Related Work

Much previous work in DRAM controllers [14, 28, 21], processor-in-memory systems [12], and vector gather/scatter units [26] has aimed to improve memory/DRAM bandwidth utilization by remapping sparse *vector* data, reordering memory requests, and/or adding vector units/register files to the execution core. Our SLS design also improves memory system behavior and core execution using related techniques. However, our SLS unit is specifically designed to work in concert with our other proposed extensions to efficiently support the *stream* programming model, which has fundamentally different attributes (e.g., locally addressable on-chip memory, bulk computations/memory accesses on complex records) from conventional programming models and overcomes the limitations of traditional vector processing. Unlike earlier approaches, our SLS unit is designed as a minimal extension to the on-chip hardware prefetcher of a modern GPP by reusing most of its existing components and datapaths. No ISA extensions or OS intervention are needed to program the SLS unit. Furthermore, the SLS unit has unique advanced capabilities tailored for parallel execution such as hardware alignment for short-vector SIMD operations, which is critical to achieving high performance in stream programs.

Research on prefetching techniques also relates to the work presented in this paper, because the SLS unit can be envisioned as a programmable extension to a hardware prefetcher (Section 3). Recently, several sophisticated processor-side and memory-side prefetchers have been proposed (e.g., [25] [29]). Although the hardware prefetchers are effective when there is a clear pattern in the memory accesses, they fail when accesses are arbitrary and data-dependent. Such "random" access patterns are common

[2]Using a base configuration with no hardware data prefetcher, we computed much higher speedups of 1.15-3.8x (2.0x on average).

in scientific applications. Another approach to prefetching data into the conventional cache hierarchy is to use software prefetch instructions [13]. Although prefetch instructions can be inserted into the program by the compiler for prefetching arbitrarily random accesses, there are uncertainties in prefetch distance and a possibility for reduced memory throughput and increased access latency as a result of unintentional cache eviction. Moreover, mis-speculated prefetches waste memory bandwidth which could adversely affect performance in memory-bound applications.

There are several alternative approaches to target the SIMD units of GPPs including compiler optimizations (e.g., auto-vectorization for Cell [3]) and assembly-level hand-optimizations (e.g., GROMACS [1]). Compiler optimizations, although effective for well written programs that access structured data with affine index expressions, are not as capable of parallelizing applications that access data in an arbitrarily data-dependent order. Low-level hand-optimizations, on the other hand, are usually tedious and time-consuming. By using the stream programming model and a SLS engine to automatically align data for the SIMD units, we complement these approaches and enhance "SIMD-ization" of complex applications.

6 Conclusions and Future Work

Traditionally, general-purpose processor architects have incorporated features that first appeared in special-purpose processors. In the same vein, we propose a simple set of extensions to incorporate the essential features of special-purpose stream processors into general-purpose hardware.

In this paper we have shown how the stream programming model can be efficiently run on GPPs with minimal hardware extensions. The extensions we proposed require less than 1% additional die area and are capable of significantly speeding up compute/memory intensive applications by better utilizing the memory bandwidth and functional units already existing in the processor.

In the future, we will extend this work to address more issues pertaining to multiple cores of a GPP. Several new factors become important with multiple cores, including optimization for local vs. global memory bandwidth, placing and migrating the data for the LM, and synchronization between kernels running on different cores.

References

[1] Gromacs. www.gromacs.org.

[2] NVidia G80. www.nvidia.com.

[3] A.E. Eichenberger et al. Optimizing Compiler for a Cell Processor. In *Parallel Architectures and Compilation Techniques*, 2005.

[4] J. Ahn, M. Erez, and W. J. Dally. Scatter-Add in Data Parallel Architectures. In *HPCA*, Feb 2005.

[5] J. H. Ahn, M. Erez, and W. J. Dally. The design space of data-parallel memory systems. In *SC'06*, November 2006.

[6] B. Khailany et al. Imagine: Media processing with streams. *IEEE Micro*, pages 35–46, March/April 2001.

[7] B. Khailany et al. Exploring the VLSI Scalability of Stream Processors. In *HPCA*, Feb 2003.

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

[9] Y. Basar and M. Itskov. Constitutive model and finite element formulation for large strain elasto-plastic analysis of shells. In *Journal of Computational Mechanics*, Jun 1999.

[10] N. Binkert, E. Hallnor, and S. Reinhardt. Network-oriented full system simulation using M5. In *CAECW*, 2003.

[11] 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.

[12] C. Kozyrakis et al. Vector IRAM: A Media-oriented Vector Processor with Embedded DRAM. In *Hot Chips*, 2000.

[13] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *ASPLOS*, Apr 1991.

[14] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, and S. McKee. Impulse: Memory system support for scientific applications. *Journal of Scientific Programming*, 7, 1999.

[15] W. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *SC*, Nov 2003.

[16] J. Gummaraju and M. Rosenblum. Stream Programming on General-Purpose Processors. In *International Symposium on Microarchitecture*, November 2005.

[17] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA*, Feb 2005.

[18] K. Fatahalian et al. Sequoia: Programming the Memory Hierarchy. In *SC*, Nov 2006.

[19] K. Mahesh et al. Large eddy simulation of reacting turbulent flows in complex geometries. *ASME J. of Applied Mechanics*, May 2006.

[20] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *ISCA*, 2003.

[21] D. Kim, M. Chaudhuri, M. Heinrich, and E. Speight. Architectural support for uniprocessor and multiprocessor active memory systems. *IEEE Transactions on Computers*, 2004.

[22] F. Labonte, P. Mattson, I. Buck, C. Kozyrakis, and M. Horowitz. The Stream Virtual Machine. In *Int'l Conference on Parallel Architectures and Compilation Techniques*, September 2004.

[23] M. B. Taylor et al. The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22:25–35, March 2002.

[24] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart memories: A modular recongurable architecture. In *Proceedings International Symposium on Computer Architecture*, 2000.

[25] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. *IEEE Micro*, 25(1):90–97, 2005.

[26] F. Quintana, J. Corbal, R. Espasa, and M. Valero. Adding a vector unit to a superscalar processor. In *ICS*, 1999.

[27] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA*, Jun 2000.

[28] S.A. McKee and Wm.A. Wulf . A Memory Controller for Improved Performance of Streamed Computations on Symmetric Multiprocessors. In *International Parallel Processing Symposium*, April 1996.

[29] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *ISCA*, 2002.

[30] M. Talluri and M. D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *ASPLOS*, 1994.

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

[32] U. Kapasi et al. Programmable Stream Processors. *IEEE Computer*, August 2003.

[33] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. *SC*, 2002.

[34] D. Wang, B. Ganesh, N. T. K. B. A. Jaleel, and B. Jacob. DRAMsim: A memory system simulator. In *SIGARCH Computer Architecture News*, September 2005.