

Compiling for Stream Processing

Abhishek Das
Stanford University

William J. Dally
Stanford University

Peter Mattson
Stream Processors, Inc.

abhishek@cva.stanford.edu billd@cva.stanford.edu mattson@streamprocessors.com

ABSTRACT

This paper describes a compiler for stream programs that efficiently schedules computational kernels and stream memory operations, and allocates on-chip storage. Our compiler uses information about the program structure and estimates of kernel and memory operation execution times to overlap kernel execution with memory transfers, maximizing performance, and to optimize use of scarce on-chip memory, significantly reducing external memory bandwidth. Our compiler applies optimizations such as strip-mining, loop unrolling, and software pipelining, at the level of kernels and stream memory operations. We evaluate the performance of our compiler on a suite of media and scientific benchmarks. Our results show that compiler management of on-chip storage reduces external memory bandwidth by 35% to 93% and reduces execution time by 23% to 72% compared to cache-like LRU management of the same storage. We show that strip-mining stream applications enables producer-consumer locality to be captured in on-chip storage reducing external bandwidth by 50% to 80%. We also evaluate the sensitivity of performance to the scheduling methods used and to critical resources. Overall, our compiler is able to overlap memory operations and manage local storage so that 78% to 96% of program execution time is spent in running computational kernels.

Categories and Subject Descriptors

D.3.2 [Programming Languages]: Language Classifications—*Specialized application languages*; D.3.4 [Programming Languages]: Processors—*Compilers*; C.4 [Performance Of Systems]: Modeling Techniques

General Terms

Languages, Management, Performance, Experimentation

Keywords

Stream Programming model, StreamC, Task level parallelism, Producer-consumer locality, Stream scheduling, coarse-grained operations, Stream Operation Precedence (SOP)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FACT'06, September 16–20, 2006, Seattle, Washington, USA.
Copyright 2006 ACM 1-59593-264-X/06/0009 ...\$5.00.

graph, SRF allocation, Strip-mining, Software-pipelining, Scoreboard slot assignment

1. INTRODUCTION

The Stream Programming model [14] provides a consistent structure to make high-level data flow and memory access patterns explicit, and limits low-level control flow. Stream programs divide an application into a series of kernels (computation-intensive functions) that operate on streams of data elements, with little or no dependency between each element's computation. Expressing an application as a stream program exposes locality at multiple levels. Kernels keep temporary values local and expose instruction level parallelism (ILP). Streams expose data-level parallelism (DLP) and producer-consumer locality between kernels: as one kernel produces stream elements, the next kernel consumes them in sequence. The Stream programming model is well-suited to media and scientific applications [14] [18] [16] [8], the dominant workload of today's systems. These applications are very compute-intensive, often performing 100 to 200 arithmetic operations for each element read from memory. They contain widespread parallelism and regular communication patterns; operations on one data element being largely independent of operations on other elements[20]. They also exhibit scarce temporal locality, and abundant producer-consumer locality between processing blocks.

Stream processing [3, 7] offers high performance and programmability using the stream programming model. Kernels are run on clusters of arithmetic units, controlled by an on-chip micro-controller. A bandwidth hierarchy of local register files, a global stream register file (SRF), and memory, exploits the kernel and producer-consumer locality by keeping most data movement local, and requiring only a small fraction of bandwidth to access memory. This bandwidth efficiency enables stream processors to efficiently make use of large numbers of arithmetic units. Keeping the stream register file (SRF) on-chip and only allowing sequential accesses, allows for faster communication. This on-chip SRF makes it possible to hide memory latency, by pro-actively fetching the streams so that they are ready when needed. The scalar control thread, running on a host processor, dispatches instructions to the stream processor, which manage the resources in the stream processor, like setting up the streams in the SRF, and instructing the micro-controller to execute kernels. These instructions are written onto a scoreboard managed by the on-chip stream controller.

Stream processing poses several new challenges for compilation. To extract maximum performance, the arithmetic

clusters need to be kept busy most of the time, doing useful computation. Kernel execution and memory transfers must be overlapped to hide memory latency. Intelligent SRF allocation must be employed to exploit producer-consumer locality and minimize use of off-chip bandwidth. Use of scoreboard slots must also expose concurrency in the instructions dispatched by the stream program executable. We present Stream Scheduling, a framework to address these challenges by leveraging knowledge of the stream programming model, as well as to perform other high-level optimizations on a stream program: strip-mining, loop unrolling, and software pipelining. Stream scheduling has several key innovations and supporting optimizations, including:

- application of data-flow analysis to streams of data
- a coarse-grained operation ordering technique to maximize task-level parallelism, including software-pipelining of main loops
- an unique allocation process that combines allocation of on-chip SRF with spilling and pre-fetching, while exploiting concurrency
- a method to automatically estimate the best strip-size
- scoreboard slot assignment to expose concurrency at run-time

We have implemented the complete compilation system for stream processing, from high level language extensions to code generation for Imagine and Merrimac. Using cycle accurate simulation of these stream processors, we show that Stream Scheduling achieves high performance for a suite of media-processing and scientific applications. We also evaluate the strength of each optimization technique in Stream Scheduling, across these applications, by monitoring different resource usages. Stream Scheduling can easily be extended to other architectures, which follow similar bandwidth hierarchy (Cell [19], ClearSpeed [4]).

Organization of the remaining paper is as follows: Section 2 briefly describes stream processing, followed by section 3 describing the compilation system; section 4 explains Stream Scheduling in detail and section 5 presents the evaluation results with discussion.

2. STREAM PROCESSING

In this section, we provide a brief description of the Stream Programming model [14] and the Stream processor architecture [3] [7].

2.1 Stream Programming Model

A stream program organizes data as streams and expresses all computation as kernels. A stream is a sequence of similar data elements, such as 8-bit pixels for an image, and are defined by a regular access pattern. Data is re-organized, when required, into sequential streams using strided and indexed access patterns.

A kernel consumes a set of input streams and produces a set of output streams. It typically loops through all input stream elements, performing a compound stream operation on each element, and appends the result(s) to output streams. These compound operations, comprising of multiple arithmetic operations, exhibit abundant instruction level parallelism (ILP). Moreover, these operations cannot access arbitrary memory locations, keeping all the intermediate values local to the kernels, and only performing fast sequential accesses on streams. Since each element of the input

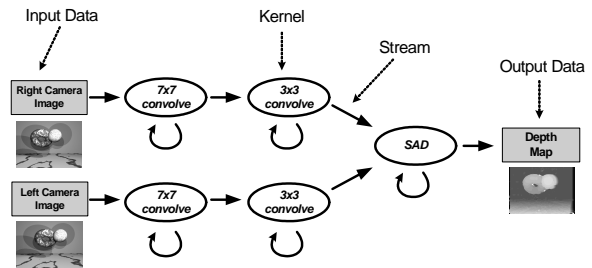


Figure 1: Graphical representation for stereo depth extractor application

stream(s) can be processed simultaneously, kernels also expose large amounts of data-level parallelism (DLP). Kernels are written in KernelC, a language with limited C-like syntax. Communication Scheduling [17], a framework to make efficient use of arithmetic and interconnect resources, is used to compile kernels.

We provide a C++ extension, called StreamC, to define the high-level control- and data-flow in stream programs. Kernels are connected by streams being produced or consumed, thereby imposing a structure on the stream program. Figure 1 shows a graphical representation of the StreamC program for a stereo depth extractor application [12]. The basic constructs of StreamC expose the parallelism and locality of streaming applications. Data communicated between computation kernels, via intermediate streams, exhibit producer-consumer locality: as one kernel produces stream elements, the next kernel consumes these elements in sequence. This is illustrated by the streams passed between the 7×7 convolution and 3×3 convolution kernels in the Stereo depth extractor application. Memory transfers of streams can execute concurrently with kernels, thereby exposing task-level parallelism in the stream program. Concurrency of memory requests and compound stream operations allow for latency tolerance.

2.2 Streaming Architecture

The stream programming model has been shown to efficiently map to stream architectures, such as Imagine[3] and Merrimac[7]. The architecture for Imagine is illustrated in Figure 2. Identical clusters of functional units operate in parallel on sequential records of streams, in a SIMD (single-instruction, multiple-data) fashion. ILP in kernels is exploited in each cluster by the multiple arithmetic units. The data bandwidth, required by the kernels, is satisfied by a deep storage hierarchy optimized for streams. Intra-cluster bandwidth into and out of the Local Register Files (LRFs), immediately adjacent to the arithmetic units, handles the bulk of data during kernel execution. The Stream Register File (SRF), an on-chip storage, is used for reading and writing streams between kernels. Off-chip memory bandwidth is used only for application inputs and outputs, scatter-gathers, and for intermediate streams that cannot fit in the SRF. The memory system can concurrently support two stream transfer requests.

Merrimac[7], designed for scientific computing, has a similar architecture to that of Imagine, but with a different mix of arithmetic units. It also has twice the number of clusters and a correspondingly larger SRF, along with hardware support for scatter-add instruction in the memory system; scatter-add acts as a regular scatter, but adds each value to

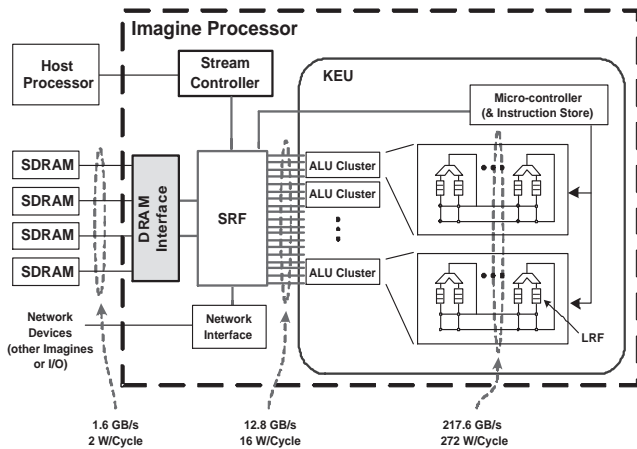


Figure 2: Imagine Stream Architecture

the data already in each specified memory address instead of overwriting them.

The stream processor runs as a co-processor to a host. The host processor executes scalar code compiled from the StreamC program and issues instructions to the stream processor via an on-chip stream controller. A scoreboard in the stream controller buffers these instructions, allowing the host processor to run ahead of the stream processor. Each of these instructions update control registers, transfer streams between the SRF and memory, and execute kernels. Dependencies between instructions, each occupying a unique scoreboard slot, are expressed in terms of the respective slots occupied, and are encoded within the instruction itself.

3. STREAMC COMPILATION

The StreamC compiler analyzes a stream program and applies the knowledge of the high-level program structure to map it to the stream hardware. This compilation process is divided into two phases. The first phase accepts a StreamC program and emits an intermediate C++ program, which is embedded with low-level stream processing instructions, while preserving the original control-flow. In the next phase, a standard C++ compiler compiles and links the intermediate code with a run-time instruction dispatcher for the stream processor, thereby generating the executable for the host processor. The run-time dispatcher coordinates all communication between the host processor and the stream processor, issuing instructions with pre-computed dependency information to the scoreboard. We use a simplistic model of the instruction dispatcher: Upon reaching the call to issue any instruction, the control thread busy-waits until the required scoreboard slot is available. Kernels, written in KernelC and used as basic constructs by StreamC, themselves get compiled by the KernelC compiler [17] to produce microcode. The generated microcode are loaded at run-time to execute kernels. The first phase of the compiler, summarized by the flowchart in Figure 3, is described in the following sections.

3.1 StreamC compiler framework

We use TenDRA [1] to parse the StreamC program into TDF (TenDRA Data Format), which in turn is parsed into an Abstract Syntax Tree (AST). Next, we modify the AST to implement high-level optimizations, such as loop unrolling and function inlining.

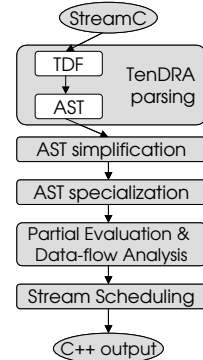


Figure 3: Flowchart of StreamC compilation phases

The StreamC code is compiled into a series of stream operations, each applied to one or more streams. For example, one stream operation might load an entire stream from memory into the SRF, while another executes a kernel on a set of streams. The sequence of stream operations is usually dictated by a small set of input parameters, such as image size in the stereo depth extractor application [12]. For a fixed set of parameters, there are few data-dependent variations in the control flow. StreamC allows the user to fix these parameters using a specialized switch statement. The AST for the program body is replicated for each case statement. The resulting AST is then passed through a partial evaluator to perform constant propagation and to evaluate data-dependent control flow. Since data-flow and dependency analysis need to be performed on stream operations, instead of individual scalar operations, we generate an intermediate representation for the various stream operations and their respective stream accesses: the SOP (Stream Operation Precedence) graph.

Stream Scheduling, described in section 4, uses the SOP graph to schedule and allocate the resources among the stream operations. Following this, every stream operation is converted into a series of low-level stream instructions. These instructions, issued by the host processor at run-time, perform reading and writing of control registers, transfer streams in and out of the SRF, load microcode, and execute kernels. Each of these low-level instructions are then assigned scoreboard slots, and are encoded with their dependencies on each other. Finally, the intermediate C++ output is generated, replacing the stream operations with run-time dispatcher calls to issue these low-level instructions.

3.2 Stream Operation Precedence (SOP) graph

Each node in the SOP graph represents a stream operation, a kernel or memory transfer, while directional edges represent the input and output streams accessed by them. Stream accesses requiring data to be fetched from memory need memory transfers, and are represented by a node for the memory operation, followed by an outgoing edge for the stream. And vice-versa for stores. This ensures that each stream edge has a single producer and one or many consumers, thus representing the data-flow in the stream program. The stream edges are annotated with the respective length and data-range information. We capture control dependencies by making the graph hierarchical. At each level, every basic block of stream operations is represented by a *supernode*, which are connected with other nodes and su-

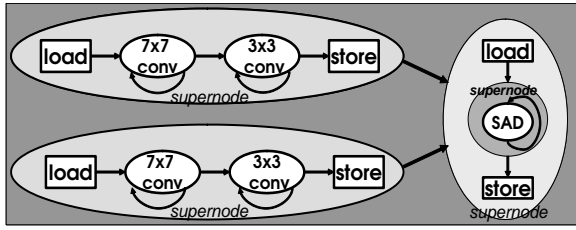


Figure 4: SOP graph for stereo depth extractor

pernodes by control- and stream data- flow. Each of these supernodes contain a SOP sub-graph connecting the nodes within them. The control-flow edges are only used to capture the high-level structure. To eliminate low-level control flow, all function calls are in-lined during AST simplification. Figure 4 shows the SOP graph for the stereo depth extractor application [12], illustrated earlier in Figure 1; loops on image rows are captured by supernodes while the kernels and memory operations are represented as nodes.

4. STREAM SCHEDULING

The primary goal of *Stream Scheduling* is to minimize the execution time of a stream program. Operations on streams must be coordinated to maximize concurrency and to efficiently utilize the on-chip storage (SRF), and other critical resources. Expensive off-chip bandwidth must be used only when necessary, since spilling a stream to memory could require reading and writing thousands of extra data elements. *Stream Scheduling* coordinates both the execution of kernels and the movement of streams in order to minimize program execution time, and allocates resources accordingly. To achieve this, we address the following key challenges:

- extract stream operations that can run concurrently,
- efficiently allocate the SRF to exploit concurrency and producer-consumer locality,
- enhance concurrency and memory latency hiding, through high-level optimizations on the loop structure: strip-mining, loop-unrolling and software-pipelining,
- automatically pick the best strip-size for strip-mining, and
- efficiently allocate the scoreboard to preserve the exposed concurrency.

Thus, *Stream Scheduling* is responsible for determining the relative ordering of operations in a stream program, and the location and size of all streams, in the working set kept in the SRF.

4.1 Stream Operation Ordering

The execution time of a stream program is decreased by keeping the computation units busy and by exploiting concurrency across stream operations. Overlapping memory transfers with useful computation in the clusters, allows hiding of memory latency in the stream program. Hence, before allocating resources to stream operations, an ordering must be generated to expose concurrency. Even though the scoreboard allows re-ordering of low-level instructions at run-time, it cannot overcome any dependencies created by resource conflicts. Moreover, it only has a limited number of slots, while a single stream operation requires multiple such instructions. Instead, stream scheduling can capitalize on its knowledge of program structure to perform global operation ordering.

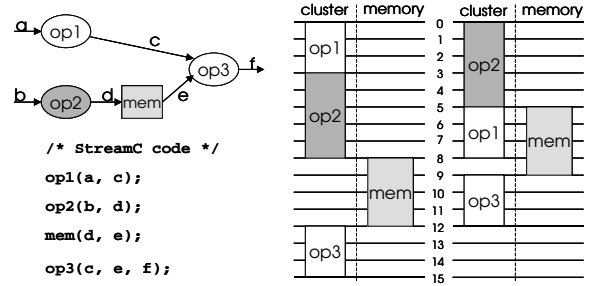


Figure 5: Execution time without and with stream operation ordering

To order operations in the SOP graph, the stream scheduler creates a resource reservation table of the execution resources: a computation unit to run kernels, and multiple units to perform simultaneous memory transfers. Each resource is occupied by an operation for the duration of its execution time. We then use top-down static list scheduling [13] [2] to schedule stream operations on these resources, reducing the total execution time. Operations are prioritized according to their criticality in achieving the minimum total execution time, and scheduled in that order. Each operation is greedily scheduled on the earliest time-step, in which a free resource is found, and all SOP dependencies are satisfied. The graph execution time and node priority are calculated by topological sort [6], using an estimate of execution time for the stream operations. After scheduling the entire graph, a sequential order of stream operations is generated, in increasing order of scheduled start times. This ordering exposes concurrency, which can be exploited during allocation of various resources. This concurrency, however, can increase resource demand. We address this in section 4.2. Figure 5 compares the execution times of a stream program with and without operation ordering (assuming all other resource requirements can be satisfied). Without ordering, *op1* gets issued first and hence serializes the rest of the operations.

The models used by the stream scheduler to estimate execution times for kernel and memory operations are described below. Due to the coarse granularity of stream operations, and the use of execution times for optimization purposes only, rough models suffice.

(a) Model for estimating Kernel execution time

A kernel, written in KernelC, usually has one main loop that goes over the entire stream being consumed. For the kernel to finish, all elements of a stream must be read. Using the following notations:

- $len_s \rightarrow$ length of stream s
- $io_{b,s} \rightarrow$ number of reads/writes of stream s in each iteration of block b
- $l_b \rightarrow$ length of block b (iteration interval for software-pipelined blocks)
- $st_b \rightarrow$ number of stages in block b ($= 1$ for non-pipelined blocks)

we compute Kernel execution time KET as follows:

$$KET = l_b * ((st_b - 1) + \min(len_s / io_{b,s}))$$

In this model, we ignore startup overheads as they are negligible compared to the computation time for long streams of

data. The model becomes non-linear when loops are nested and when a stream is read or written in multiple loop bodies. Conditional loops are profiled to determine the frequency of loop bodies. We use a non-linear solver in matlab [5] to handle these cases.

(b) *Model for estimating load/store run-time*

We compute the time taken for memory loads and stores as a function of the stream access pattern and DRAM characteristics:

$$\text{load/store time} = \text{dram}_{\text{latency}} + \text{len}_s / \text{throughput}$$

While $\text{dram}_{\text{latency}}$ is only the initial latency to activate and fetch data from the DRAM, the throughput of memory operations is determined by the stream length, len_s , and several other DRAM parameters:

- $\text{dram}_c \rightarrow$ number of CPU cycles corresponding to a single DRAM cycle
- $\text{dram}_w \rightarrow$ data width of DRAM banks
- $n_b \rightarrow$ number of DRAM banks
- $\text{dram}_b \rightarrow$ number of words fetched in a burst

Maximum throughput is achieved for sequential (uni-stride) data access in streams:

$$\text{throughput}_{\text{max}} = \frac{n_b * \text{dram}_w}{\text{dram}_c}$$

In order to model all non-sequential access patterns, we consider a random access pattern where only one word in every burst is useful:

$$\text{throughput}_{\text{random}} = \text{throughput}_{\text{max}} * \text{dram}_w / \text{dram}_b$$

The memory bandwidth is shared between concurrent memory transfer operations, thereby reducing the throughput for each operation. For simplicity, we divide the throughput equally among them, ignoring the actual dynamic behavior.

4.2 SRF management

Efficient management of the SRF is essential for good performance, as stream programs often demand more data bandwidth than the available memory bandwidth; the SRF, as a data staging area, needs to satisfy a significant portion of the program’s data accesses. This involves allocating the streams in the SRF and determining when to load them into the SRF, and store them back into memory. The allocation mechanism needs to exploit producer-consumer locality by keeping streams in the SRF between accesses by kernels, thus reducing memory traffic. The allocation must also preserve the concurrency exposed through stream operation ordering; all concurrently accessed streams must be staged together in the SRF.

To handle streams that are too large to completely fit within the SRF, the *double-buffering* technique is applied. While a kernel processes a chunk of the stream, the next part of the stream is concurrently fetched into a disjoint buffer in the SRF. After the kernel exhausts the data in the first buffer, it starts processing the data in the second buffer, while the next part of the stream is fetched into the first buffer in parallel. Continuing to switch data between the buffers in this fashion, streams of arbitrary lengths are handled. *Double-buffering* is also used to ensure that all stream accesses made by any single stream operation can fit in the SRF.

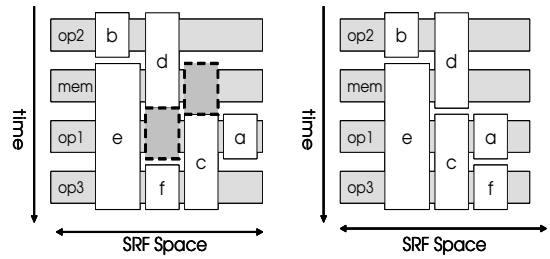


Figure 6: SRF allocation with and without operation overlapping, for example in Figure 5

The core framework for SRF allocation is based on the earlier work of [16]. Each stream access in the SOP graph is assigned to a rectangular logical *buffer* of certain width and height. The width of the buffer, defined by the size of the stream, represents a set of contiguous locations in the SRF. The height of the buffer represents the continuous duration of a stream’s residency, its lifetime, in the SRF. To preserve locality, where possible, all stream operations reading or writing to the same stream share a common buffer, thus determining the buffer’s lifetime. If, however, all the accesses cannot be assigned to the same buffer, the stream must be spilled to memory and fetched later into a different buffer for further accesses. The overall allocation problem requires placing these rectangular buffers in the SRF such that the amount of data in the SRF is never greater than the size of the SRF at any point in time. Buffers with disjoint lifetimes can overlap in SRF space.

The earlier work in [16] makes the naive assumption of each stream operation taking equal time to execute, thus incorrectly modeling the relative lifetimes of streams in the SRF. We address this by using the estimated execution times of stream operations (described in section 4.1). This allows us to represent relative lifetimes of streams, and their respective buffers, more accurately. Buffers for all concurrent stream accesses conflict in their lifetime, preventing them from occupying the same SRF location. This allows the exposed concurrency to be preserved during SRF allocation.

Exploiting concurrency, however, can put too much demand on the SRF and result in unnecessary spilling. We mitigate this using two heuristics. Firstly, while scheduling operations in the resource reservation table (section 4.1), we keep track of the total SRF pressure at every time-step. An operation is not allowed to be scheduled at a time-step if its stream accesses cause the SRF pressure to increase beyond the SRF size. Secondly, the buffer heights are represented in terms of the stream operations accessing the buffer. This defines a range within the sequence of stream operations, generated in section 4.1, after operation ordering. Since the strictly sequential ordering of stream operations doesn’t capture concurrent stream accesses, the buffer heights are extended using *shadows* to include those stream operations which overlap, in time, with the producers and consumers of the stream. These shadows prevent spatial conflicts in the SRF for buffers of concurrent operations, but are reduced first during packing to avoid spilling. Figure 6 demonstrates the SRF allocation for the example stream program of Figure 5. Since the memory load *mem* could execute in parallel with the kernel *op1*, buffers for *c* and *d* have been extended with shadows. By reducing the shadows, buffers for *c* and *d* conflict in SRF space, thereby sacrificing concurrency.

This two dimensional packing problem reduces to a variant of the NP-hard *dynamic storage allocation* problem [9, 10], which is solved using an iterative algorithm that combines operation ordering and SRF packing:

1. Mark stream accesses that require memory transfers to synchronize data between buffers
2. Repeatedly divide each buffer, if the accesses assigned to it can be divided into two disjoint groups, without requiring additional memory accesses
3. Add shadows to buffers based on operation ordering and their execution times
4. Position buffers in the SRF using packing heuristics

The heuristic is based on the notion of trying to form long vertical strips of densely packed buffers, and hence tries to position each buffer at the leftmost possible position. It tries to complete the current strip before starting another, and positions the largest buffers first so that smaller buffers can fill in the cracks.

5. If packing is unsuccessful, apply heuristics to pick a buffer for reduction, and iterate

The first approach is to sacrifice concurrency by reducing the *shadows*. Since double-buffering on a stream applies to a single stream operation at a time, and requires the data to be re-fetched from memory for the next stream access, sizes of these buffers are reduced next. Otherwise, a buffer is reduced by splitting its lifetime into two smaller lifetimes and inserting spills in between. Buffers with the longest interval of time between accesses benefit most from this. Double-buffering is used as a last resort to reduce the width of buffers. Reduction of buffers change the operation execution times, and hence require another pass of operation ordering (section 4.1).

This improved algorithm reduces the amount of spilling and preserves the scope of concurrency between stream operations.

4.3 Strip-mining

A typical stream program consists of a series of stream operations, each producing intermediate streams to be consumed by the next operation in sequence, before producing the final output. However, most applications operate on streams that are larger than the SRF. Hence, these temporary results would have to incur expensive memory transfers using *double-buffering* (section 4.2), losing the benefit of locality and limiting throughput to the available memory bandwidth. To eliminate this bottleneck, the large input stream can be segmented into smaller strips and the entire series of operations applied to one strip at a time, allowing the SRF to stage each intermediate stream without any memory transfers. This high-level optimization, called *strip-mining* [22], can be easily incorporated in stream programs because of the parallelism between each stream element’s computation. Moreover, the sizes of all intermediate and output streams can be easily derived using a simple linear transformation of the initial input stream size. We employ strip-mining in StreamC by allowing the programmer to structure simple loops around these streams, and bounding their sizes as linear functions of a single variable, henceforth called the *strip-size*. The *strip-size* is assigned by a call to *getStripSize()* in the stream program. The stream scheduler picks a value for the strip-size to maximize the working

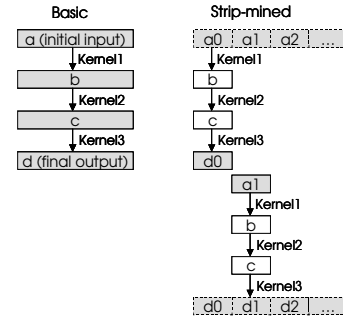


Figure 7: Basic and strip-mined data-flow for a stream program

set size kept in the SRF, determining the stream sizes using the partial evaluator (section 3.1). The loop structure also exposes concurrency among the stream operations in successive iterations, which process independent strips. If any state is retained between strips in successive loop iterations, we put the onus on the programmer to handle them. Figure 7 illustrates basic and strip-mined dataflow for a simple stream program containing three kernels.

A large *strip-size* is desirable to mitigate kernel startup and shutdown overheads, especially when kernels are heavily software pipelined [3]. For any stream operation in the strip-mined loop, the best strip-size should be able to stage all the required streams in the SRF, without double-buffering. It follows that the largest strip-size possible for the strip-mined loop is the minimum amongst the best strip-sizes for each contained stream operation. However, large strip-sizes can cause concurrent stream operations to serialize, and intermediate streams to be spilled during SRF allocation. The stream scheduler performs a binary search between the largest possible and the user-provided strip-size, to find the best performing strip-size for the loop, as summarized in Figure 8. The relative execution times of the strip-mined loop, estimated using the SOP graph scheduling algorithm (section 4.1), are used as the performance metric for comparison. When the loop bounds are not known at compile time, the stream scheduler can only estimate the execution time of a single iteration. In such cases, since the number of loop iterations are directly proportional to the strip-sizes, the execution times are scaled to the ratio of the strip-sizes being compared.

```

1. Schedule loop body with minimum strip-size
   min_len = estimated execution time for strip-mined loop
   strip-size = binaryStripSearch(min_strip-size, max_strip-size, max_strip-size, min_len)

2. binaryStripSearch(min_strip-size, max_strip-size, cur_strip-size, min_len)
   • if (min_strip-size >= cur_strip-size)
     return min_strip-size

   • schedule loop body using cur_strip-size
     cur_len = estimated execution time for loop body

   • best_len = min_len
     if (loop bounds not known) /* scale execution time */
       best_len = min_len x (cur_strip-size / min_strip-size)

   • if (best_len <= cur_len) /* search for smaller strip-size */
     next_strip-size = (min_strip-size + cur_strip-size) / 2
     return binaryStripSearch(min_strip-size, cur_strip-size, next_strip-size, min_len)

   • else /* search for larger strip-size */
     next_strip-size = (cur_strip-size + max_strip-size) / 2
     return binaryStripSearch(cur_strip-size, max_strip-size, next_strip-size, cur_len)

```

Figure 8: Pseudo-code for binary search of best strip-size

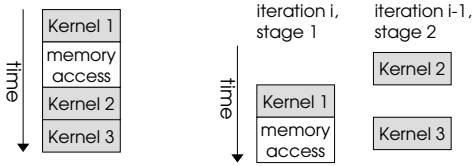


Figure 9: Software-pipelining a loop with sequential memory access

4.4 Loop unrolling and Software pipelining

Since most stream programs are structured around high-level loops, we discuss a couple of well-known loop optimizations and their applicability to the stream programming paradigm. Our discussion applies only to loops in StreamC, structured around stream operations, and not within kernels.

Loop unrolling [11] is a technique to increase concurrency within the loop body. It works by concatenating multiple copies of the original loop body to form a new, larger loop body. Producing a larger loop body provides a larger block of instructions for the scheduler to work with, which gives the scheduler more options when ordering operations. Since the number of loop iterations may not be an integral multiple of the *unroll factor*, code must be generated to check for this case and handle any remaining iterations. StreamC supports loop unrolling using a special directive, which specifies the unroll factor as well as the number of iterations to peel out from the start and end of loop.

Another effective global optimization for loops is *software pipelining* of operations [15]. Software pipelining can reorder operations to expose concurrency between operations in successive loop iterations. It involves dividing a loop into stages and overlapping execution of one stage of an iteration with execution of another stage of a successive iteration. Software-pipelining can be used to hide any stream memory access, that must occur between a pair of sequential kernels. In Figure 9, the second kernel cannot start immediately after the first kernel, and must wait for the intervening memory transfer to complete. Software pipelining hides the latency for this memory transfer operation by overlapping it with execution of a kernel from another stage. Exploiting concurrency across loop iterations also circumvents any serialization caused by memory transfers at the end of a loop iteration and those at the beginning of next iteration. Software pipelining does, however, come at the cost of increased SRF demand since more streams are now live at any particular point in time. Using the execution time models of section 4.1, the stream scheduler implements conventional modulo scheduling [15] on the SOP graph to divide the stream operations into different stages. This creates a modulo ordering of the stream operations. This ordering is used during SRF allocation, which preserves the exposed concurrency using shadows.

4.5 Scoreboard slot assignment

After the SRF is allocated, control registers are allocated to the stream operations in a similar fashion. The stream operations are then broken down into low-level stream instructions. These instructions, encoded with their dependencies on each other, are dispatched by the host processor at run-time, by writing them into a on-chip scoreboard slot.

Mutually independent stream instructions must be written into independent slots to exploit concurrency, as a scoreboard slot is occupied till the instruction finishes executing. In our simplistic model, the host control thread dispatches instructions in a sequence and stalls if the slot required by an instruction is occupied. The stream scheduler assigns slots to instructions and then encodes the dependencies between them statically. It uses a round-robin assignment of scoreboard slots, n in number, allowing the control thread to maintain the largest window (n) of sequential instructions with no slot conflicts. For loops, however, slots may conflict between instructions at the start and end of loop body, creating loop carried dependencies. This is prevented by assigning slots in a modulo fashion at the end of loop, thereby keeping them disjoint from the slots assigned to instructions at the start of the loop.

5. EVALUATION RESULTS AND DISCUSSION

We evaluate the performance of our compilation system on a suite of media and scientific benchmarks, MPEG-2, DEPTH, RTSL and MOLE, which have been shown [14] [18] [16] [8] to be well-suited for stream processing. The MPEG-2 encoding benchmark encodes three frames of a 360x288 24-bit color image, in the following sequence: I-Frame followed by two P-Frames. The depth extraction benchmark (DEPTH) computes depth information from two 320x240 8-bit gray-scale stereo images, based on Kanades algorithm [12]. RTSL is the implementation of an OpenGL-like rendering pipeline in the stream programming model [18]. We render the first frame of the SPECviewPerf 6.1.1 Advanced Visualizer Benchmark into a 24-bit RGB color framebuffer with a window size of 720x720 pixels. This benchmark requires relatively fine-grained control flow since there is uncertainty in the number of triangles generated from the vertices and the rest of the pipeline is dependent on the number. MOLE is an implementation of the molecular dynamics application [8], which models interaction between water molecules using a version of the highly optimized GROMACS code [21]. The kernels are scheduled using the KernelC compiler [17], to make best use of the arithmetic clusters. We compile MOLE on the Merrimac system [7], which was developed for scientific computation, while the other media processing benchmarks are compiled for Imagine [3]. Using a cycle-accurate simulator for stream processors, which accurately models all aspects of kernel execution and memory system, we demonstrate that the stream scheduler makes efficient utilization of resources and achieves high performance for these benchmarks.

5.1 SRF packing

We first demonstrate the utility and importance of the SRF packing algorithm presented in section 4.2. We compare the execution times of each benchmark under two scenarios: allocating streams in the SRF in an *on-demand* manner, and using the packing algorithm. The former method works in operation order, meeting the SRF demand of streams by using the first available free space or spilling the least recently used stream, thus ignoring locality of stream accesses. Since the stream programming model exhibits abundant producer consumer locality, spilling the most aged stream, however, is appealing. This also emulates the behavior of a

LRU cache. We make no other changes in the compilation system for this study.

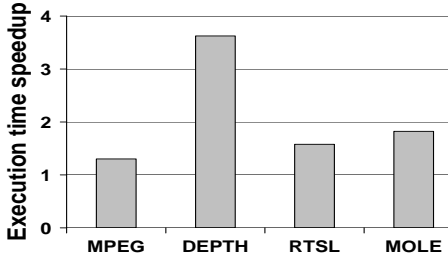


Figure 10: Speed-up achieved using SRF packing algorithm over on-demand allocation

Figure 10 shows that we get considerable speed-up in execution time with the SRF packing algorithm, over the conventional *on-demand* mechanism. Even though MPEG shows less relative speed-up due to abundant producer-consumer locality, Figure 11 clearly shows the SRF packing algorithm reducing memory traffic, significantly, across all benchmarks. The effect is most pronounced for DEPTH since its loop structure has multiple nested levels, and the on-demand mechanism doesn't preserve any locality for loops.

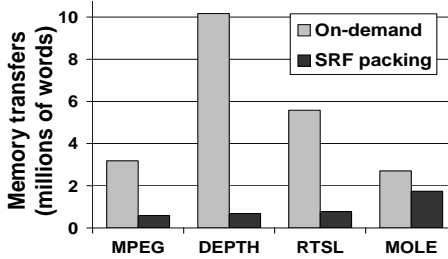


Figure 11: Comparison of memory transfers with and without efficient SRF packing

5.2 Strip-mining

We now demonstrate the effectiveness of *Strip-mining*, which exposes producer-consumer locality and concurrency with smaller manageable working sets. Since MPEG and DEPTH have working sets small enough to tightly fit in the SRF even without strip-mining, we study the effect of strip-mining only on RTSL and MOLE. As explained in section 4.3, the stream scheduler picks a strip size to minimize memory traffic and hide memory latency.

While Figure 12 clearly illustrates the ability of strip-mining to significantly reduce memory traffic, Figure 13 shows the respective execution time speed-up attained. RTSL demonstrates marginal speed-up because frequent occurrence

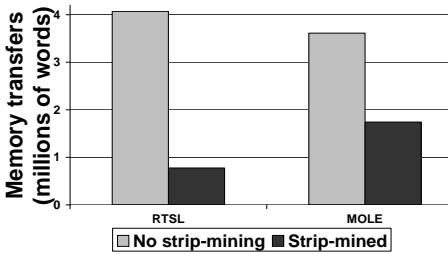


Figure 12: Comparison of memory transfers with and without strip-mining

of data dependent conditionals in the scalar control code causes serialization.

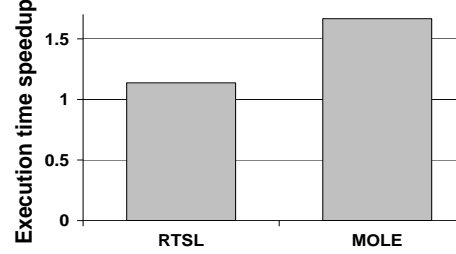


Figure 13: Speedup using strip-mining

5.3 Loop unrolling

All the selected benchmarks are structured around loops, either due to the application's characteristics, or due to strip-mining. Figure 14 shows the speed-up attained with loop unrolling, thus demonstrating its effectiveness in exposing concurrency. RTSL does not get any benefit because of the serialization caused by data-dependent conditional control code. MPEG demonstrates only a marginal improvement, since it extracts significant performance even without unrolling. DEPTH is structured around nested loops, while MOLE has abundant concurrency across strips. Hence, both show significant speed-up with unrolling.

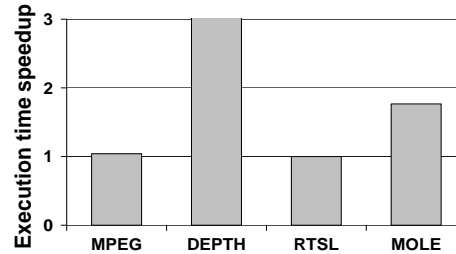


Figure 14: Speedup using loop unrolling

5.4 Operation ordering and Resource allocation

In this section, we evaluate the impact of SRF and scoreboard allocation on performance. We compare the performance achieved by the following stream scheduling variations, described in sections 4.2 and 4.5 respectively.

- *naive*: in-order scheduling with simple round-robin scoreboard slot assignment
- *slot*: modulo scoreboard slot assignment at end of loop body
- *shadow*: SRF allocation using shadows derived from a greedy schedule with execution time models
- *sdw-slot*: combines *slot* and *shadow*
- *reorder*: operation re-ordering and software-pipelining, combined with *sdw-slot*

For maximum performance benefit, RTSL uses strip-mining and DEPTH uses loop unrolling, while MOLE uses both strip-mining and loop unrolling. Figure 15 illustrates the respective execution time speed-up attained by the scheduling variations over the *naive* approach. Without re-ordering, DEPTH benefits most from using *slot* by removing scoreboard slot dependencies across the start and end of loop iterations, effectively avoiding any control thread stalls. While

DEPTH benefits only a little using *shadow* alone, significant speedup is attained with *sdw-slot*. MOLE, in contrast, derives most of its benefit from *shadow* alone since independent strips of the unrolled loop are allocated separate SRF locations, thus preserving concurrency. Across all benchmarks, *reorder* reaps the most benefit of concurrency, attaining significant speed-up for MPEG and RTSL, which were shown to benefit marginally from unrolling.

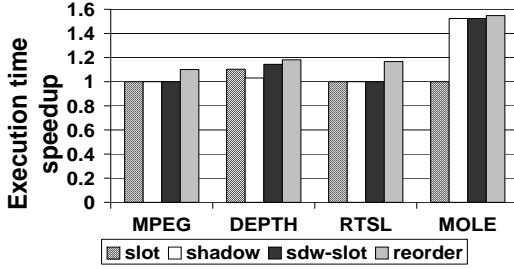


Figure 15: Speedup attained by stream scheduling variants over a naive approach

All benchmarks benefit from software pipelining during operation re-ordering. Since DEPTH and MOLE are already unrolled, they benefit marginally from software pipelining. The current AST, derived from TenDRA, cannot perform automatic transformations to move scalar code blocks and hence cannot reflect this re-ordering during code generation. This is partly because the main focus of the compiler framework was to develop the SOP graph intermediate representation and use it for stream scheduling, and not the chosen AST. We get around this limitation by manually reflecting the re-ordering generated by the stream scheduler, on the StreamC code, and then making another pass through the compiler. Even though this adds burden on the programmer, the results in Figure 15 show that we achieve reasonable performance even without re-ordering, using *sdw-slot*.

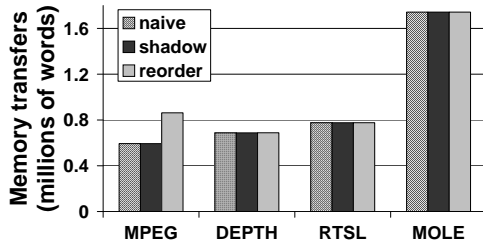


Figure 16: Comparison of memory transfers for stream scheduling variants

The *shadow* and *reorder* variations of stream scheduling exploit concurrency to hide memory latency. This, in effect, increases the size of the working set in SRF. However, this can lead to spills of smaller streams and increase the memory traffic, which is demonstrated clearly for MPEG and DEPTH in Figure 16. In conjunction with the speed-up results in Figure 15, we conclude that the stream scheduler is very effective in hiding memory latency and exploiting concurrency. We do not compare the results for *slot* and *sdw-slot* because *slot* does not alter the SRF packing.

5.5 Resource utilization and Sensitivity study

Next, we evaluate the achieved utilization of the critical resources using stream scheduling. The goal of stream

scheduling is to make the program spend most of its time doing useful computation, in the form of kernels. Accordingly, we define *efficiency* as the fraction of execution time spent in running kernels. Figure 17 uses this metric to demonstrate the effectiveness of stream scheduling in attaining its goal. The relative efficiency of RTSL is low because it stalls the scalar control thread intermittently to wait on results generated by the stream processor. Results are shown using only the *reorder* variant since it has already been shown to perform the best.

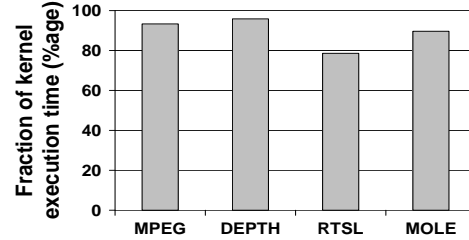


Figure 17: Fraction of program execution time spent in running kernels

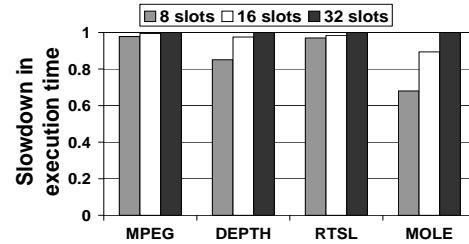


Figure 18: Slowdown in execution time due to reduction in scoreboard slots

Finally, we evaluate the sensitivity of stream scheduling on the number of scoreboard slots. As shown in Figure 18, when the number of slots is reduced from the original system value of 32, there is only a marginal slow-down in most cases. Since DEPTH has an inner loop of strictly sequential kernel calls, enough concurrency cannot be captured with a small window size of 8 slots. MOLE is characterized by 6 memory load operations and one store operation, for every kernel call. After unrolling and software pipelining, concurrency is captured by avoiding the loads to serialize on the store operation for the previous kernel. However, as the window size of scoreboard slots is reduced, conflicts on scoreboard slots cause serialization. Moreover, since the scalar control thread stalls on the first such occurrence, it fails to dispatch any following ready instruction with an empty free slot. The solution is to decouple the dispatcher thread from the control thread. Unless a data-dependent control flow is required, the control thread should enqueue all pending instructions to a buffer, and the dispatcher thread should peek into it to write an instruction to the scoreboard when the required slot is free.

6. CONCLUSIONS

This paper has described a stream compiler that accepts a stream program (in StreamC) and generates efficient code for a stream processor. Our compiler uses the structure of the stream program to schedule computational kernels and stream memory operations to optimize performance. The

compiler explicitly manages on-chip storage (SRF) to make the most efficient use of this scarce resource and to minimize off-chip bandwidth.

Our compiler performs a source-to-source transformation on the input StreamC code. The resulting C++ code is then compiled by a conventional compiler. During the transformation, portions of streams are mapped to the on-chip storage (SRF) and the required stream load and store operations are generated. This packing is done using estimates of kernel and memory operation durations, and using *shadows* to extend the lifetimes of streams. The transformations include applying strip mining, loop unrolling, and software pipelining at the level of kernels and stream memory operations. With strip-mining, our compiler automatically chooses the strip-size that results in minimum execution time — trading off the cost of spills against the overhead of small strips.

We have demonstrated the performance of our compiler on a suite of media and scientific benchmarks. Explicitly managing the on-chip storage results in large improvements in both execution time (23% to 72%) and memory traffic (35% to 93%) compared to managing this same storage using a cache-like LRU policy. The strip-mining transformation also results in large improvements in memory traffic (50% to 80%) by capturing producer-consumer locality. The loop unrolling transformation significantly improves execution time for some benchmarks (up to 70%) by exposing additional stream operation parallelism. Overall, our compiler is able to overlap memory operations and manage local storage so that 78% to 96% of program execution time is spent running computational kernels.

7. REFERENCES

- [1] The tendra open source project.
<http://www.tendra.org>.
- [2] T. Adam, K. Chandy, and J. Dickson. A comparison of list scheduling for parallel processing systems. *J. Communications of the ACM*, vol. 17, no. 17, pages 685–690, December 1974.
- [3] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das. Evaluating the imagine stream architecture. In *Proceedings of the Annual International Symposium on Computer Architecture*, June 2004.
- [4] ClearSpeed. CSX600 datasheet.
<http://www.clearspeed.com/downloads/CSX600Processor.pdf>, 2005.
- [5] T. F. Coleman and Y. Li. An interior trust region approach for nonlinear minimization subject to bounds. *SIAM Journal on Optimization*, pages 418–445, 1996.
- [6] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1989.
- [7] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J. H. Ahn, N. Jayasena, U. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, page 35, Phoenix, Arizona, 2003.
- [8] M. Erez, J. H. Ahn, A. Garg, W. J. Dally, and E. Darve. Analysis and performance results of a molecular modeling application on merrimac. In *Proceedings of the SC'04 Conference*, Pittsburgh, Pennsylvania, November, 2004.
- [9] J. Fabri. Automatic storage optimization. In *Proceedings of the ACM SIGPLAN 1979 Symposium on Compiler Construction*, pages 83–91, 1979.
- [10] J. Gergov. Algorithms for compile-time memory optimization. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 907–908, 1999.
- [11] D. R. Kaiser. *Loop Optimization Techniques on Multi-Issue Architectures*. PhD thesis, University of Michigan, 1994.
- [12] T. Kanade, A. Yoshida, K. Oda, H. Kano, and M. Tanaka. A stereo machine for video-rate dense depth mapping and its new applications. In *Proceedings of the 15th Computer Vision and Pattern Recognition Conference*, pages 196–202, San Francisco, CA, June 18–20, 1996.
- [13] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, no. 11, pages 1023–1029, November 1984.
- [14] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang. Imagine: Media processing with streams. *IEEE Micro*, pages 35–46, Mar/Apr 2001.
- [15] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation (PLDI)*, pages 318–328, 1988.
- [16] P. Mattson. *A Programming System for the Imagine Media Processor*. PhD thesis, Stanford University, 2001.
- [17] P. Mattson, W. J. Dally, S. Rixner, U. J. Kapasi, and J. D. Owens. Communication scheduling. In *Proceedings of the 9th international conference on Architectural support for programming languages and operating systems(ASPLOS)*, volume 35, pages 82–92, Cambridge, MA, November 2000.
- [18] J. D. Owens. *Computer Graphics on a Stream Architecture*. PhD thesis, Stanford University, 2002.
- [19] D. Pham, S. A. M. Bollinger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. S. an M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first generation cell processor. In *ISSCC Digest of Technical Papers*, San Francisco, CA, Feb, 2005.
- [20] S. Rixner. *Stream Processor Architecture*. Kluwer Academic Publishers, Boston, MA, 2001.
- [21] D. van der Spoel et al. Gromacs user manual version 3.1. <http://www.gromacs.org>, 2001.
- [22] M. Wolfe. More iteration space tiling. In *Proceedings of the Supercomputing 89*, pages 655–664, New York, NY, November 1989.