Analysis and Performance Results of a Molecular Modeling Application on Merrimac

Mattan Erez† Jung Ho Ahn† Ankit Garg† William J. Dally† Eric Darve‡

†Department of Electrical Engineering
Stanford University
Stanford, CA, 94305

‡Department of Mechanical Engineering
Stanford University
Stanford, CA, 94305

Abstract

The Merrimac supercomputer uses stream processors and a high-radix network to achieve high performance at low cost and low power. The stream architecture matches the capabilities of modern semiconductor technology with compute-intensive parallel applications. We present a detailed case study of porting the GROMACS molecular-dynamics force calculation to Merrimac. The characteristics of the architecture which stress locality, parallelism, and decoupling of memory operations and computation, allow for high performance of compiler optimized code. The rich set of hardware memory operations and the ample computation bandwidth of the Merrimac processor present a wide range of algorithmic trade-offs and optimizations which may be generalized to several scientific computing domains. We use a cycle-accurate hardware simulator to analyze the performance bottlenecks of the various implementations and to measure application run-time. A comparison with the highly optimized GROMACS code, tuned for an Intel Pentium 4, confirms Merrimac's potential to deliver high performance.

1. Introduction

Modern semiconductor technology allows us to place hundreds of functional units on a single chip but provides limited global on-chip and off-chip bandwidths[25, 9]. General purpose processor architectures have not adapted to this change in the capabilities and constraints of the underlying technology, still relying on global on-chip structures for operating a small number of functional units[4, 5, 15]. Stream processors[13, 3], on the other hand, have a large number of functional units, and utilize a deep register hierarchy with high local bandwidth to match the bandwidth demands of the functional units with the limited available off-chip bandwidth.

A stream program exposes the large amounts of data parallelism available in some application domains, as well as multiple levels of locality. The Merrimac stream processor then exploits the parallelism to operate a large number of functional units and to hide the long latencies of memory operations. The stream architecture also lowers the required memory and global bandwidth by capturing short term producer-consumer locality, such as the locality present within a function call, in *local register files* (LRF) and long term producer-consumer locality in a large *stream register file* (SRF), potentially raising the application's *arithmetic intensity* (the ratio of arithmetic to global bandwidth).

This work was supported, in part, by the Department of Energy ASCI Alliances Program, Contract LLNL-B523583, with Stanford University, the NVIDIA Fellowship Program, the MARCO Interconnect Focus Research Center, the Stanford Graduate Fellowship Program, Northrup Grummon Corporation, and the Stanford Mechanical Engineering Summer Undergraduate Research Institute.

SC'04, November 6-12, 2004, Pittsburg, Pennsylvania, USA
0-7695-2153-3/04 $20.00 (c)2004 IEEE

Merrimac [3] is a design for a fully-programmable streaming supercomputer tailored specifically to exploit the parallelism and locality of many scientific codes. The design is scalable up to a $16,384$ processor 2PFLOPS system. The core of Merrimac is the single chip Merrimac processor, aimed for 90nm CMOS technology at 1GHz, that directly connects to 2GB of fast commodity DRAM and to the global interconnection network. We estimate the parts cost of this 128GFLOPS (double-precision) node to be under $\$1,000$, and expect Merrimac to be more cost effective as both a capacity and capability machine than ones designed using conventional processors.

This paper deals with the development of the StreamMD molecular dynamics application on Merrimac and its performance. We describe the algorithm used, explore the different trade-offs and optimizations allowed by the hardware, draw generalizations of these techniques to scientific computing domains other than molecular dynamics, and provide feedback to the Merrimac hardware and software development teams. We also analyze the performance on the Merrimac cycle-accurate simulator and compare it to an implementation on a traditional processor, as well as present initial results of the scaling of the algorithm to larger configurations of the system.

Molecular dynamics is a technique to model chemical and biological systems at the molecular level. It has been successfully applied to a wide range of disciplines including designing new materials, modeling carbon nanotubes and protein folding. In this paper, we are interested in the modeling of bio-molecules, such as proteins or membranes, using the highly optimized code GROMACS. In the most accurate models, a large number of water molecules surround the protein. In this case, the most expensive step in the calculation is the evaluation of the long range Lennard-Jones and electrostatic forces. We implemented a version of the algorithm where a cutoff is applied to those forces such that two atoms interact only if they are within a certain distance r_c from one another. One of the difficulties in this calculation on a data-parallel architecture such as Merrimac, is that the number of neighbors of a given atom (*i.e.* all the atoms at a distance less than r_c) varies. We explore several methods to deal with this problem using a combination of replication, addition of dummy atoms, and Merrimac's *conditional streams* mechanism [11]. The approach which uses conditional streams achieves the shortest run-time. Based on our simulation results and estimates, StreamMD on Merrimac outperforms a Pentium 4, produced using the same 90nm semiconductor technology as Merrimac, by a factor of 13.2. It is important to understand that the techniques proposed in this paper are applicable whenever the data set is unstructured which occurs in many other scientific applications such as calculations involving unstructured grids, sparse matrix calculations, and tree codes among others.

The remainder of the paper is organized as follows. Section 2 presents details on the Merrimac architecture and highlights its capabilities and constraints. Section 3 presents the details of the Molecular Dynamics application and the various implementations and optimizations. Our experimental setup and results appear in Section 4. A discussion and future work are in Section 5, and concluding remarks are given in Section 6.

2. Merrimac architecture and organization

The Merrimac system exploits the capabilities and constraints of modern VLSI and signaling technologies and matches them with application characteristics to achieve high performance at lower power and cost than today's state of the art systems. In a 90nm modern semiconductor fabrication process a 64-bit floating-point functional unit (FPU) requires an area of roughly 0.5mm^2 and consumes less than 50pJ per computation. Hundreds of such units can be placed on an economically sized chip making arithmetic almost free. Device scaling of future fabrication processes will make the relative cost of arithmetic even lower. On the other hand, the number of input/output pins available on a chip does not scale with fabrication technology making bandwidth the critical resource. Thus, the problem faced by architects is supplying a large number of functional units with data to perform useful computation. Merrimac employs a stream processor architecture to achieve this goal.

In this section we present the design of the Merrimac system, which is scalable from a 2TFLOPS workstation up to a 2PFLOPS supercomputer. We first describe the Merrimac processor and the stream programming model. We then continue with details of the memory system, the interconnection network, and the packaging of an entire Merrimac system.

2.1 Merrimac processor

The Merrimac processor is a stream processor that is specifically designed to take advantage of the high arithmetic-intensity and parallelism of many scientific applications. It contains a scalar core for performing control code and issuing stream instructions to the stream processing unit, as well as the DRAM and network interfaces on a single chip. Most of the chip area is devoted to the stream execution unit whose main components are a collection of arithmetic clusters and a deep *register hierarchy*. The compute clusters contain 64 64-bit FPUs and provide high compute performance, relying on the applications' parallelism. The register hierarchy exploits the applications' locality in order to reduce the distance an operand must travel thus reducing global bandwidth demands. The register hierarchy also serves as a data staging area for memory in order to hide the long memory and interconnection network latencies.

As shown in Figure 1, Merrimac's stream architecture consists of an array of 16 clusters, each with a set of 4 64-bit multiply-accumulate (MADD) FPUs, a set of *local register files* (LRFs) totaling 768 words per cluster, and a bank of the *stream register file* (SRF) of 8KWords, or 1MB of SRF for the entire chip. At the planned operating frequency of 1GHz each Merrimac processor has a peak performance of 128GFLOPS, where the functional units have a throughput of one multiply-add per cycle.

Each FPU in a cluster reads its operands out of an adjacent LRF over very short and dense wires. Therefore the LRF can provide operands at a very high bandwidth and low latency, sustaining 3 reads per cycle to each FPU. FPU results are distributed to the other

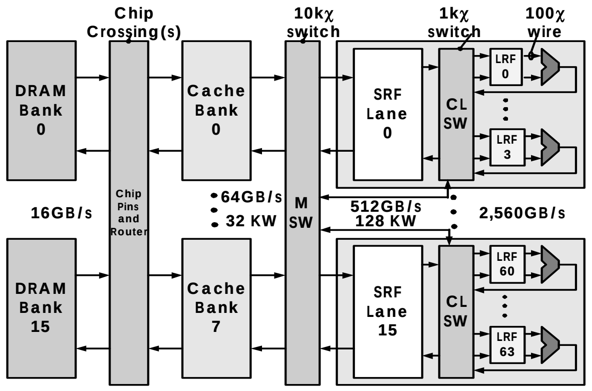


Figure 1: Architecture of Merrimac's stream core

LRFs in a cluster via the cluster switch over short wires, maintaining the high bandwidth required (1 operand per FPU every cycle). The combined LRFs of a single cluster, or possibly the entire chip, capture the *short term producer-consumer locality* of the application. This type of locality arises from the fact that the results of most operations are almost immediately consumed by subsequent operations, and can live entirely within the LRF. In order to provide instructions to the FPUs at a high rate, all clusters are run in *single instruction multiple data* (SIMD) fashion. Each cluster executes the same *very long instruction word* (VLIW) instruction, which supplies a unique instruction to each of the cluster's FPUs. Instructions are fetched from the on-chip micro-code store.

The second level of the register hierarchy is the stream register file (SRF), which is a software managed on-chip memory. The SRF provides higher capacity than the LRF, but at a reduced bandwidth of only 4 words per cycle for a cluster (compared to over 16 words per cycle on the LRFs). The SRF serves two purposes: capturing *long term producer-consumer locality* and serving as a data staging area for memory. Long term producer-consumer locality is similar to the short term locality but cannot be captured within the limited capacity LRF. The second, and perhaps more important, role of the SRF is to serve as a staging area for memory data transfers and allow the software to hide long memory latencies. An entire stream is transferred between the SRF and the memory with a single instruction. These stream memory operations generate a large number of memory references to fill the very deep pipeline between processor and memory, allowing memory bandwidth to be maintained in the presence of latency. FPUs are kept busy by overlapping the execution of arithmetic kernels with these stream memory operations. In addition, the SRF serves as a buffer between the unpredictable latencies of the memory system and interconnect, and the deterministic scheduling of the execution clusters. While the SRF is similar in size to a cache, SRF accesses are much less expensive than cache accesses because they are aligned and do not require a tag lookup. Each cluster accesses its own bank of the SRF over the short wires of the cluster switch. In contrast, accessing a cache requires a global communication over long wires that span the entire chip.

The final level of the register hierarchy is the inter-cluster switch which provides a mechanism for communication between the clusters, and interfaces with the memory system which is described in the following subsection.

In order to take advantage of the stream architecture, the stream programming model is used to allow the user to express both the locality and parallelism available in the application. This is achieved by casting the computation as a collection of *streams* (i.e. sequences of identical data records) passing through a series of computational *kernels*. The semantics of applying a kernel to a stream are completely parallel, so that the computation of the kernel can be performed independently on all of its stream input elements and in any order. Thus *data level parallelism* is expressed and can be utilized by the compiler and hardware to drive the 64 FPUs on a single processor and the 1 million FPUs of an entire system. An additional level of *task parallelism* can be discerned from the pipelining of kernels. Streams and kernels also capture multiple levels and types of locality. Kernels encapsulate short term producer-consumer locality, or *kernel locality*, and allow efficient use of the LRFs. Streams capture long term producer-consumer locality in the transfer of data from one kernel to another through the SRF without requiring costly memory operations, as well as spatial locality by the nature of streams being a series of data records.

While the Merrimac processor has not been implemented we have sketched a design for it based on a currently available 90nm fabrication process and a clock frequency of 1GHz (37 FO4 inverters in 90nm[25]). Each MADD unit measures $0.9\text{mm} \times 0.6\text{mm}$ and the entire cluster measures $2.3\text{mm} \times 1.6\text{mm}$. The floor-plan of a Merrimac stream processor chip is shown in Figure 2. The bulk of the chip is occupied by the 16 clusters while left edge of the chip holds the remainder of the node. There are two MIPS64 20kc [21] scalar processors where one processor shadows the other for reliability purposes. The node memory system consists of a set of *address generators*, a line-interleaved eight-bank 64KWords (512KB) cache, and interfaces for 16 external Rambus DRDRAM chips. A network interface directs off-node memory references to the routers. We estimate that each Merrimac processor will cost about $200 to manufacture[1] and will dissipate a maximum of 25W of power. Area and power estimates are for a standard cell process in 90nm technology and are derived from models based on a previous implementation of a stream processor [14].

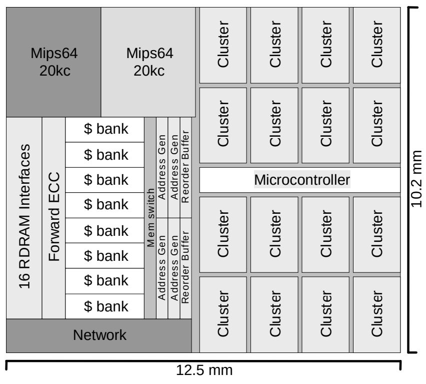


Figure 2: Floor-plan of a Merrimac stream processor chip.

2.2 Memory system

The memory system of Merrimac is designed to support efficient stream memory operations. As mentioned in Subsection 2.1 a single stream memory operation transfers an entire stream, which is typically many thousands of words long, between memory and the SRF. Merrimac supports both strided access patterns and gathers/scatters through the use of the stream address generators. Each processor chip has 2 address generators, which together produce up to 8 single-word addresses every cycle. The address generators take a base address in memory for the stream, and either a fixed stride and record-length or a pointer to a stream of indices in the SRF. The memory system provides high-bandwidth access to a single global address space for up to 16,384 nodes including all scalar and stream execution units. Each Merrimac chip has a 128KWords cache with a bandwidth of 8 words per cycle (64GB/s), and directly interfaces with the node's external DRAM and network. The 2GB of external DRAM is composed of 16 Rambus DRDRAM chips providing a peak bandwidth of 38.4GB/s and roughly 16GB/s, or 2 words per cycle, of random access bandwidth. Remote addresses are translated in hardware to network requests, and single word accesses are made via the interconnection network. The flexibility of the addressing modes, and the single-word remote memory access capability simplifies the software and eliminates the costly pack/unpack routines common to many parallel architecture. The Merrimac memory system also supports floating-point and integer streaming add-and-store operations across multiple nodes at full cache bandwidth. This *scatter-add* operation performs an atomic summation of data addressed to a particular location instead of simply replacing the current value as new values arrive. This operation is used in the molecular dynamics application described, and is also common in finite element codes.

2.3 Interconnection network

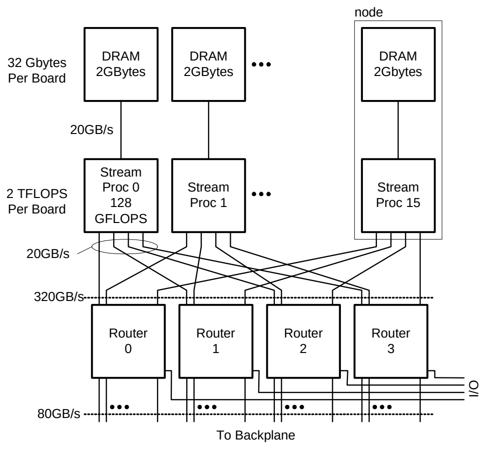


Figure 3: Sixteen 128GFLOPS stream processors each with 2GB of DRAM memory can be packaged on a single board. The board has a total of 2TFLOPS of arithmetic and 32GB of memory. Such a board is useful as a stand-alone scientific computer and as a building-block for larger systems.

Merrimac is scalable from a 16 node 2TFLOPS single board workstation to a 16,384 node 2PFLOPS supercomputer spanning 16 cabinets. Figure 3 illustrates a single Merrimac board containing 16 nodes totaling a peak performance of 2TFLOPS with a capacity of 32GB of memory, and four high-radix router chips. The router chips interconnect the 16 processors on the board, providing flat memory bandwidth on board of 20GB/s per node, as well as providing a gateway to the inter-board interconnection network. The network uses a five-stage folded-Clos [2] network[2] as illustrated in Figure 4. The first and last stage of the network are the on-board router chips, where every router provides two 2.5GB/s channels to each of the 16 processor chips as well as eight such channels to connect to the back-plane level of the network. At the backplane level, 32 routers connect one channel to each of the 32 boards in a

[1]Not accounting for development costs.

[2]This topology is sometimes called a Fat Tree [17].

cabinet and connect 16 channels to the system-level switch. A total of 512 2.5GB/s channels traverse optical links to the system-level switch where 512 routers connect the 16 backplanes in 16 cabinets. The maximal number of cabinets that can be connected by the network is 48 but we do not plan to build a system of that size. As with the rest of the Merrimac design, the network does not rely on future technology and can be built using current components and custom chips.

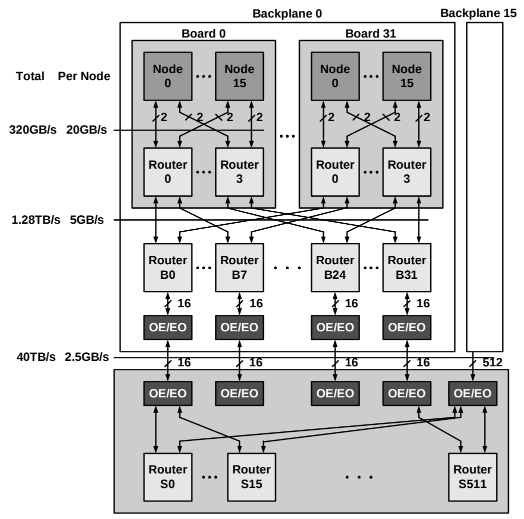


Figure 4: A 2PFLOPS Merrimac system uses a high-radix interconnection network.

3. StreamMD

Molecular Dynamics is the technique of simulating detailed atomic models of molecular systems in order to determine the kinetic and thermodynamic properties of such systems. GROMACS [28] is an engine that performs molecular dynamics simulations by repeatedly solving Newton's equations of motion for all the atoms in a particular system. The simulations follow a discrete timescale, and are used to compute various observables of interest. Often, such simulations are used to design new materials or study biological molecules such as proteins, DNA. When simulating the latter, it is common to surround those molecules with a large number of water molecules. In that case a significant portion of the calculation is spent calculating long ranged interactions between water molecules and between the protein and the surrounding water molecules.

GROMACS is highly optimized to take advantage of mechanisms found in commercial high-end processors including hand optimized loops using SSE, 3DNow!, and Altivec instructions [27, 1, 22]. One of the numerical method employed by GROMACS for long ranged forces uses a cut-off distance approximation. All interactions between particles which are at a distance greater than r_c are approximated as exerting no force. This approximation limits the number of interactions calculated for the system from $O(n^2)$ to $O(n)$, where each particle (we will refer to this as the *central particle*) only interacts with a small number of neighboring particles. The list of neighbors for each particle is calculated in scalar-code and passed to the stream program through memory along with the particle positions. The overhead of the neighbor list is kept to a minimum by only generating it once every several time-steps. The accuracy of the calculation is maintained by artificially increasing the cutoff distance beyond what is strictly required by the physics. We have chosen this phase of the GROMACS algorithm to test a streaming implementation on Merrimac.

3.1 Basic implementation

Since the Merrimac system integrates a conventional scalar processor with a stream unit it offers a simple path for porting an application. Most of the application can initially be run on the scalar processor and only the time consuming computations are streamed. We are developing *StreamMD* to perform the force calculation of GROMACS using Merrimac's highly parallel hardware. We are currently concentrating on the force interaction of water molecules and interface with the rest of GROMACS directly through Merrimac's shared memory system.

The stream program itself consists of a single kernel performing the interactions. As will be explained later, the interacting particles are not single atoms but entire water molecules. Each kernel iteration processes a molecule and one of its neighbors, and computes the non-bonded interaction force between all atom pairs (Equation 1). The first term contributes to Coulomb interaction, where $\frac{1}{4\pi\epsilon_0}$ is the electric conversion factor. The second term contributes to Lennard-Jones interaction, where C_{12} and C_6 depend on which particle types constitute the pair. After accounting for various computations required to maintain periodic boundary conditions, each interaction requires 234 floating-point operations including 9 divides and 9 square roots.

$$V_{nb} = \sum_{i,j} \left[\frac{1}{4\pi\epsilon_0} \frac{q_i q_j}{r_{ij}} + \left(\frac{C_{12}}{r_{ij}^{12}} - \frac{C_6}{r_{ij}^6} \right) \right] \qquad (1)$$

GROMACS provides the configuration data consisting of a position array containing nine coordinates for each molecule (because we are using water molecules these are the coordinates for each of the three atoms), the central molecule indices stream `i_central` (one element per molecule), and the neighboring molecules indices stream `i_neighbor` (one element per interaction). These streams index into the position input array and the force output array. These data are loaded through memory, so that the stream unit can be used as a coprocessor for the time-intensive force computation portion of the program.

The basic flow of StreamMD is as follows:

1. Gather the positions of the interacting molecules into the SRF using Merrimac's hardware gather mechanism.

2. Run the force computation kernel over all interactions, reading the required data from the predictable SRF. The kernel's output is a stream of partial-forces and a stream of indices which maps each force to a molecule. Those partial forces are subsequently reduced to form the complete force acting on each atom.

3. Reduce the partial forces. This is achieved using Merrimac's scatter-add feature.

This can be summarized in the pseudo-code below:

```
c_positions = gather(positions,i_central);
n_positions = gather(positions,i_neighbor);

partial_forces =
     compute_force(c_positions,n_positions);

forces =
       scatter_add(partial_forces,i_forces);
```

As explained in Section 2, to attain high performance on Merrimac we must ensure our program has latency tolerance, parallelism, and locality. The following subsections explain how each of these is achieved, and what optimizations and trade-offs are available on Merrimac.

3.2 Latency tolerance

Merrimac can tolerate the high latency of memory operations which results both from the slow access times of DRAM and the low off-chip bandwidth. This is done in two different ways. First, by issuing a memory operation consisting of a long stream of words, the requests may be pipelined to amortize the initial long latency. Second, by concurrently executing memory operations and kernel computations, thus hiding memory accesses with useful computation. The SRF serves as a staging area for memory operations and decouples them from the execution units, permitting memory and computation to proceed concurrently. It allows the software-system or programmer to *pipeline* the stream execution as illustrated in Figure 5: while a kernel is running, the memory system is busy gathering data for the next invocation of the kernel into the SRF, and writing the results of the previous kernel out of the SRF into memory.

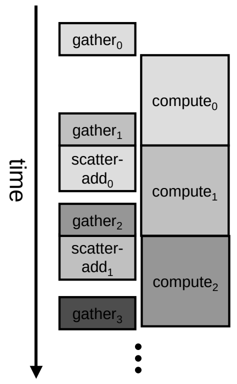


Figure 5: Software pipelining overlaps computation and memory operations

Since StreamMD consists of a single kernel, it must first be *strip-mined* before pipelining can be employed. Strip-mining refers to a software technique of breaking up a large data set into smaller strips which are more easily handled by the hardware [18]. In the case of StreamMD we strip-mine the computation by including an outer loop such that each kernel invocation is only performed on a subset of the input stream:

```
for i=1..strips {
  c_positions =
      gather(positions,i_central[strip_i]);
  n_positions =
      gather(positions,i_neighbor[strip_i]);

  partial_forces =
     compute_force(c_positions,n_positions);

  forces = scatter_add(partial_forces,
                       i_forces[strip_i]);
}
```

The strip-mine process was done manually, but the next version of our compiler will handle this case automatically.

3.3 Parallelism

StreamMD has abundant parallelism as each interaction can be calculated independently of all other interactions, and the forces reduced in a separate step. This parallelism is used to operate Merrimac's 16 clusters in SIMD fashion. This requires that all compute clusters perform the same operations every cycle.

The simplest implementation (*expanded* in Table 3) is to fully expand the interaction lists, so that the kernel reads 16 pairs of interacting molecules each iteration – one pair per cluster. Thus, each cluster reads in two interacting molecules and produces two partial forces. This requires 27 input words (18 words for the coordinates of the 6 atoms, and 9 words for periodic boundary conditions) and 18 output words (for the two partial forces). In addition three index streams must also be read, for a total of 48 words for the 234 operations of each interaction. The partial forces are reduced using the hardware scatter-add mechanism (Section 2.2).

A different implementation, which is still pure SIMD, is to use a fixed-length neighbor list of length L for each central molecule (*fixed* in Table 3). Now each cluster reads a central molecule once every L iterations and a neighbor molecule each iteration. The partial forces computed for the L interactions of the central molecule are reduced within the cluster to save on output bandwidth. This configuration requires 9 words of input and 9 words of output every iteration (for the position and partial forces of the neighboring molecule), and 18 words of input and 9 words of output for every L iterations (for the central molecules position, boundary condition, and partial forces). Including the index streams we end up with $18 + 1 + \frac{27+2}{L}$ words per iteration per cluster. If we choose a fixed length of $L = 8$ the result is 22.6 words per iteration instead of 48 of the naive implementation. Since the number of neighbors for each molecule is typically larger than L, central molecules are repeated in `i_central` in order to accommodate all their neighbors. Furthermore, whenever a central molecule does not have a multiple of L neighbor molecules, some dummy neighbor molecules are added to `i_neighbor` to keep the fixed $L : 1$ correspondence (Figure 6).

p(0)	p(1)	p(2)	p(3)	$\cdots$	p(N)

Original position array `position`

0^1	0^2	1^1	1^2	1^3	2^1	$\cdots$	$\cdots$	N^1

Central molecule index stream `i_central`

p(0^1)	p(0^2)	p(1^1)	p(1^2)	p(1^3)	p(2^1)	$\cdots$	$\cdots$	p(N^1)

Gathered central position stream `c_positions`

p(0^1_1)	$\cdot$	p(0^1_L)	p(0^2_1)	$\cdot$	p(0^2_L)	p(1^1_1)	$\cdot$	p(1^1_L)	$\cdots$

Gathered neighbor position stream `n_positions`

Figure 6: Central molecules are repeated and neighbors are padded for fixed length neighbor lists

The disadvantages of this scheme are the extra bandwidth required for reading the repeated molecules and the dummy values, as well as the extra computation performed on the dummy neighbors. This computation does not contribute to the solution but consumes resources. The overheads are typically not large for a reasonable value of L (between 8 and 32).

A third implementation option relies on Merrimac's inter-cluster communication, *conditional streams* mechanism [11], and the *indexable SRF* [10] (*variable* in Table 3). These mechanisms allow a SIMD architecture to process inputs and produce outputs at a different rate for each cluster. We use this to implement variable-length neighbor lists for each central molecule. Instead of reading a new central molecule once every L iterations, each cluster checks on each iteration whether a new molecule is required or not. To adhere to SIMD restrictions, the instructions for reading a new central position and the writing of the partial forces for the central molecule are issued on every iteration with a condition. Unless a new molecule is required the instructions have no effect. This leads to a slight overhead of unexecuted instructions within the kernel, but since they are not floating-point arithmetic instructions they have little detrimental effect on the overall kernel efficiency. The exact number of words required for an iteration depends on the number of neighbors per central molecule, but as a minimum 10 words of input are consumed and 9 words are produced for every iteration in order to read the neighbor coordinates and index, and write the partial force acting on that neighbor.

3.4 Locality and computation/bandwidth trade-offs

The locality in StreamMD is a short term producer consumer locality (*i.e.* within a single kernel). It is the result of computing 9 full atom-atom forces while reading the positions of just two molecules in every kernel iteration, capturing intermediate results used for these force calculations, and the internal reduction of the partial forces of a central molecule within a cluster.

This locality is entirely captured within the LRFs. The amount of arithmetic intensity changes with the implementation chosen as shown in the preceding subsection. A key insight is that we can trade-off extra computation for reduced memory bandwidth requirements. Since Merrimac has abundant arithmetic resources, this might lead to an overall performance improvement. One example of such a trade-off was presented in the previous subsection regarding extra computation performed for interactions with dummy molecules.

A more extreme case is to duplicate all interaction calculations and save on the bandwidth required for the partial forces (*duplicated* in Table 3). In this scheme, the complete force for each center molecule is reduced within the clusters and no partial force for neighbor molecules is written out. More sophisticated schemes are briefly described as future work in Section 5.

4. Evaluation

In this section we evaluate the effectiveness of Merrimac and its automated compilation system for running the molecular dynamics force computation. We begin by describing our experimental setup, continue by evaluating our various implementation versions according to locality, computation, and total run-time, suggest improvements to the Merrimac system, and finally compare the results with the fully hand optimized GROMACS on a conventional processor.

4.1 Experimental setup

For this paper we conducted single-node experiments for the run of one time-step of a 900 water-molecule system. The code was run on a cycle-accurate simulator of a single Merrimac streaming node, which accurately models all aspects of stream execution and the stream memory system. The parameters of the Merrimac system are summarized in Table 1, and the characteristics of the dataset appear in Table 2.

For comparison purposes the same data-set was run on the latest version of GROMACS 3.2 with its hand optimized assembly loops using single-precision SSE instructions. We used GCC 3.3.4 on a 3.4GHz Intel Pentium 4 processor fabricated in a 90nm process.

We currently programmed 4 variants of StreamMD, which are summarized in Table 3.

[3] 64 fused multiply-add operations per cycle.

Parameter	Value
Number of stream cache banks	8
Number of scatter-add units per bank	1
Latency of scatter-add functional unit	4
Number of combining store entries	8
Number of DRAM interface channels	16
Number of address generators	2
Operating frequency	1GHz
Peak DRAM bandwidth	38.4GB/s
Stream cache bandwidth	64GB/s
Number of clusters	16
Peak floating point operations per cycle	128[3]
SRF bandwidth	512GB/s
SRF size	1MB
Stream cache size	1MB

Table 1: Merrimac parameters

Parameter	Value
molecules	900
interactions	$61,770$
repeated molecules for *fixed*	9150
total neighbors for *fixed*	$71,160$

Table 2: Dataset properties

Name	Description
expanded	fully expanded interaction list
fixed	fixed length neighbor list of 8 neighbors
variable	reduction with variable length list
duplicated	fixed length lists with duplicated computation
Pentium 4	fully hand-optimized GROMACS on a Pentium 4 with single-precision SSE (water-water only)

Table 3: Variants of StreamMD

4.2 Latency tolerance

Merrimac's hardware and software systems automatically allocate space in the SRF, and schedule memory and kernel operations to execute concurrently [12]. This can be graphically seen in a snippet of the execution of *duplicated*. Figure 7a shows that even though the program spends more time executing kernels than memory operations, memory latency is not completely hidden. After analyzing the reasons, we found a flaw in Merrimac's allocation of a low-level hardware register which holds a mapping between an active stream in the SRF and its corresponding memory address. After modifying the system to better handle this case, Figure 7b shows a perfect overlap of memory and computation.

4.3 Locality

The arithmetic intensity and locality are tightly related and strongly dependent on the specific implementation. Table 4 summarizes the arithmetic intensity as calculated in Section 3 based on the algorithm properties, as well as the measured operations per input/output word for the actual data set. For *fixed* and *variable*, the arithmetic intensity also depends on the data set. We also calculate the arithmetic intensity more accurately based on the actual data set used (without actually running the code), and present it in Table 4 as well (these values appear in parentheses). The small differences between the theoretical and measured numbers attest to the ability of the compilation tools to efficiently utilize the register hierarchy of Merrimac.

A more direct measure of locality is shown in Figure 8, where

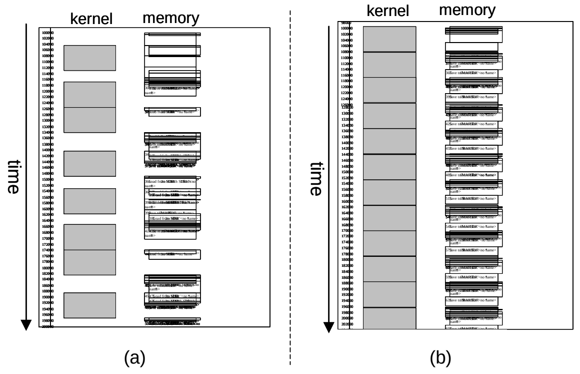


Figure 7: Snippet of execution of "*duplicated*" (see Table 3) showing the improvement in overlap of memory and kernel operations. Left column of both (a) and (b) represents a kernel being executed, and the right columns are for memory operations.

Variant	Calculated	Measured
expanded	4.9	4.9
fixed	10.3 (8.6)	8.6
variable	12.0 (9.9)	9.7
duplicated	8.0	7.2

Table 4: Arithmetic intensity of the various StreamMD implementations. Values in parenthesis are more accurate calculations accounting for the actual data set properties.

each bar represents the percentage of references made to each level of the register hierarchy. Nearly all references are made to the LRFs due the large amounts of kernel locality available in this application. The relatively small difference between the number of references made to the SRF and to memory indicates the use of the SRF as a staging area for memory, and not for capturing long term producer-consumer locality.

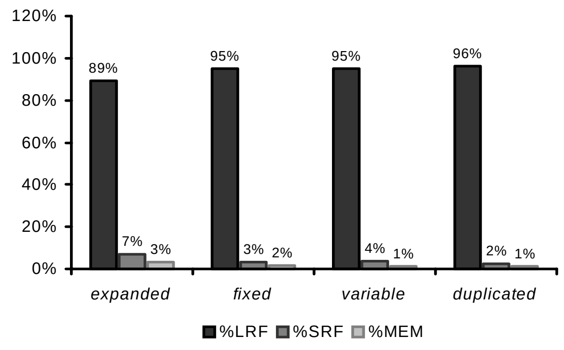
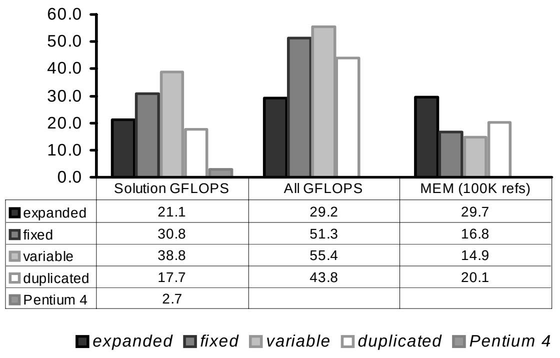


Figure 8: Locality of the various implementations

4.4 Performance

Figure 9 shows the performance results for the various implementations. The left-most bar group represents the rate of computation for floating-point operations directly contributing to the water-water interaction force calculation, and corresponds to the time-to-solution – a higher value indicates faster execution. The central group is the execution rate of all floating-point operations on the hardware. These include operations on dummy neighbors, kernel initialization, and other overheads. The right-most group shows the total number of memory accesses performed. While a higher total execution rate or lower memory reference count do not correlate with better algorithm performance we can clearly see the various trade-offs made between increased amount of computation and memory bandwidth requirements of the difference implementation choices. At this point we only estimate the performance on a conventional processor based on the wall-clock time of simulating the same data set for 1000 time-steps. We are assuming the actual force calculation accounts for 75% of the time.



	Solution GFLOPS	All GFLOPS	MEM (100K refs)
expanded	21.1	29.2	29.7
fixed	30.8	51.3	16.8
variable	38.8	55.4	14.9
duplicated	17.7	43.8	20.1
Pentium 4	2.7		

Figure 9: Performance of the different StreamMD implementations

The highest performance implementation is *variable* which outperforms *expanded* by 84%, *fixed* by 26%, *duplicated* by 119%, and GROMACS on a 3.4GHz Pentium 4 by a factor of 13.2.

Not only does Merrimac outperform a conventional processor fabricated with the same semiconductor process by a factor of over 13, Merrimac also performs 64-bit double-precision computations at full bandwidth as opposed to single-precision operations on the Pentium 4.

5. Discussion and future work

5.1 Merrimac allows for effective automatic optimizations

The Merrimac hardware is designed to allow effective automatic compilation. The hardware presents a simple target which is mostly software controlled, so that the compiler has access to all execution information and can statically decide on optimizations. The main enabling hardware feature is the SRF which decouples memory from execution. It serves as a buffer for the unpredictable nature of DRAM and interconnection network traffic, and presents a deterministic view of data supply to the execution clusters. Each execution cluster is a VLIW execution core with a distributed register file, an inter-cluster result switch, and has predictable latency and bandwidth access to its SRF bank.

Communication scheduling [20] takes advantage of these features, and automatically employs sophisticated compilation techniques to achieve near optimal schedules. Figures 10a and 10b show the effect of applying loop unrolling and software pipelining to the interaction kernel of *variable*. In each figure, the 4 columns represent the cluster's 4 arithmetic units, and a box represents an instruction that started execution at a particular cycle. The optimized kernel is unrolled twice and software pipelined to achieve an execution rate that is 83% better than the original kernel. It takes 120 cycles for each iteration of the loop, which performs two distinct interactions (unrolled loop). While executing this inner loop of the kernel a new instruction is issued on 90% of the cycles.

In contrast to this automated approach highly optimized code for conventional processors is often written directly in assembly, as was done for GROMACS.

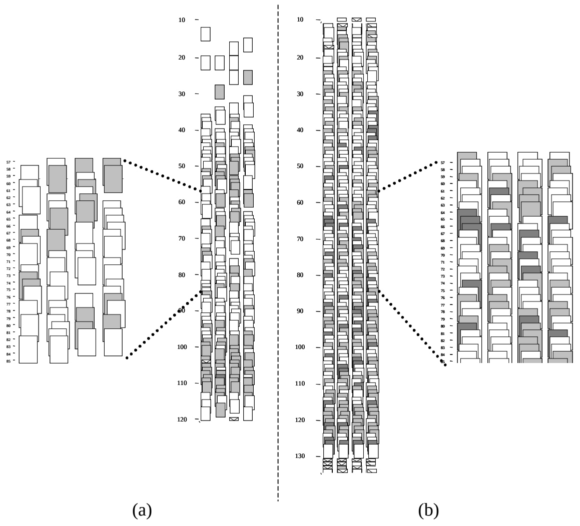


Figure 10: Schedules of the *variable* interaction kernel (see Table 3), before, and after optimization is applied – (a) and (b) respectively

In addition the SRF is software managed using a technique called *stream scheduling*, allowing the compiler to capture long term producer-consumer locality, and manage eviction from the SRF in an intelligent manner which cannot be mimicked by a conventional data cache. Kapasi *et al.* discuss this point in depth [12].

The effectiveness of the automated tools is evident in the performance of StreamMD. While the peak performance of Merrimac is specified at 128GFLOPS, the optimal we can achieve for StreamMD is 77.6 *solution GFLOPS* – floating point operations that are specified by the programmer. Optimal performance is lower than peak due to the facts that peak performance can only be achieved if all operations are fused multiply-add, and that divides and square-roots are computed iteratively and require several operations. *Variable* sustains a respectable 50% of this optimal at 38.8 solution GFLOPS and 55.5GFLOPS overall (43% of peak). The reasons for not achieving optimal performance are mainly constraints on communication scheduling, overheads associated with running a kernel and priming its software-pipelined loop, and small inefficiencies in stream scheduling.

5.2 Merrimac is tuned for scientific applications and modern semiconductor technology

While most conventional processors are designed to execute a large set of applications, ranging from productivity applications to scientific simulation and games, Merrimac is specifically tuned for scientific applications. Merrimac is not a special purpose processor and is fully programmable. It takes advantage of the high arithmetic intensity and large amount of parallelism found in many scientific codes and matches them with the capabilities and limitations of modern semiconductor technology.

This allows the Merrimac hardware to provide a large number of execution units with data and instructions leading to much higher performance, both in terms of computation and memory bandwidth. We demonstrated this performance advantage by the StreamMD application that, based on our simulation results, will run 13.2 times faster on a Merrimac processor than a conventional Pentium 4 fabricated in the same 90nm semiconductor process.

Moreover, Merrimac allows full bandwidth double-precision operations which are important to the scientific computing community. Merrimac also simplifies programming in a multi-node setting by providing ample global bandwidth and a single name-space for memory.

We did not perform a direct comparison to processors other than the Pentium 4. However the GROMACS benchmark results [6] indicate that leading conventional CPUs from different companies, belonging to the same generation as the Pentium 4 such as the AMD Opteron and IBM PPC970, perform within 10% of one another.

We also would have liked to compare Merrimac to a vector machine or a special purpose design such as MDGRAPE-3 [26], but the lack of standardized benchmarks puts it beyond the scope of this paper. Simply comparing operations per second, or even the computation rate of atom interactions, is not enough to draw meaningful conclusions due to the wide range of algorithms used. For example, MDGRAPE-3 boasts impressive numbers of 165GFLOPS, or 5 billion atom-atom interactions every second, using conservative design in a 0.13μm process. However, the algorithm used on MDGRAPE computes many more interactions than GROMACS since it does not maintain interaction lists. Instead it performs a full $O(n^2)$ interactions for long range forces, and $O(n'^2)$ for limited range forces where n' is the number of molecules in two cells whose dimension is ${r_{cutoff}}^3$.

5.3 Implications of StreamMD on other applications

We chose StreamMD as a case study of application development on Merrimac for three main reasons. First, interaction force calculation in molecular dynamics shares many characteristics with other scientific codes. Second, the algorithm for water-water interaction is fairly straightforward to understand and implement. And finally, since a very highly optimized version exists for conventional processors for comparison purposes.

While developing StreamMD we are encountering many general problems such as variable-length work-lists. These appear as neighbor lists in our case, but are also common in finite-element methods with unstructured grids, sparse linear algebra, graph algorithms, and adaptive mesh applications. The techniques developed for StreamMD are generally applicable to scientific codes on Merrimac and other data-parallel machines, and not limited to molecular dynamics. Just as importantly, StreamMD serves as a test case for the Merrimac hardware design and software systems, and we are able to provide feedback to the Merrimac architecture and compilation teams. The improved hardware features and compiler algorithms again benefit other scientific domains as well.

5.4 Computation and memory bandwidth trade-offs and future work

As semiconductor technology continues to scale on-chip devices at a higher rate than off-chip bandwidth, recomputing values instead of fetching them from memory becomes a meaningful optimization. Performance can potentially be improved by increasing the arithmetic intensity, either trading off some of the computation for memory bandwidth as was done in the *duplicated* variant, or by taking advantage of the ample compute resources to implement more accurate chemical models. We explore the two options in this section.

Computation and memory bandwidth can be traded off by grouping molecules into cubic clusters of size r^3. Since the GROMACS algorithm requires an interaction sphere of radius r_c (cut-off radius), we will pave this sphere with cubic clusters. This approach, which we call the *blocking* scheme, is similar to the technique of blocking used in many applications such as matrix vector products, and fast Fourier transforms. It is possible to show that the total memory bandwidth required with this approach scales as $O(r^{-3})$. However, the computation has to go up since the calculation now includes extra pairs of molecules which are at a distance between r_c and $r_c + 2\sqrt{3}r$. We wrote MATLAB code to estimate this trade-off more precisely. This estimate is less accurate than the Merrimac simulator but it points out the trends and motivates further work. Figure 11 shows the increase in computation and the reduction in memory operations relative to the *variable* scheme. This shows that

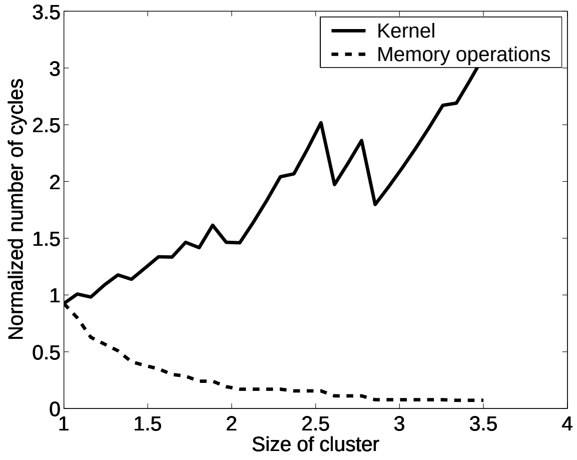


Figure 11: Estimate of computation and memory operations with the *blocking* scheme. Results are relative to the *variable* scheme and were obtained with MATLAB. The size of the cluster has been normalized so that a cluster of size 1 contains exactly one molecule.

some improvement can be expected. Using data from the simulation of the *variable* scheme, we further estimate the run-time of the *blocking* scheme and show that there is a minimum around a cluster size of 1.4^3, which corresponds to a cluster containing about 3 molecules. These results will be confirmed and validated in the future using the more accurate Merrimac simulator.

The second topic of future research is to consider more complex water models. Since Merrimac can provide high performance for applications that exhibit good data locality, complex water models are attractive as they can significantly increase the amount of arithmetic intensity. Water models are developed in order to improve the fit with one particular physical structure (protein folding) or a set of parameters such as density anomaly, radial distribution function, and other critical parameters. By and large, the more parameters are used, the better the fit. Some examples of physical properties computed with various models, and their corresponding real experimental values, are given in Table 5. In our simulation, we considered a model where 3 partial charges are located at the hydrogen and oxygen atoms. More advanced models use up to 6 charges [23] at various locations. In all those models the location of the charges is considered to be fixed relative to the molecule and thus does not require any additional memory bandwidth. In real physical systems however, molecules exhibit significant polarization. Polarizable force fields [8] seek to account for appropriate variations in charge distribution with dielectric environment. Examples are PPC [16], SWFLEX-AI [29], POL5/TZ [7]. All of those schemes give better fit. They also lead to a significant increase in

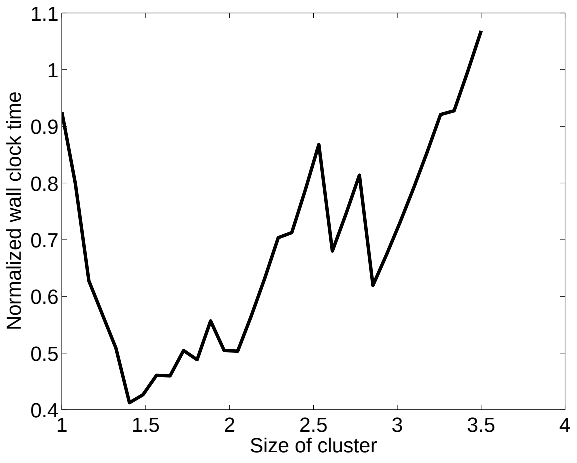


Figure 12: Estimate of execution time with the *blocking* scheme, relative to the *variable* scheme. The minimum occurs at about 3 molecules per cluster (cluster size of 1.4^3).

arithmetic intensity. Consequently, Merrimac will provide better performance for those more accurate models.

Model	Dipole moment	Dielectric constant	Self diffusion 10^{-5} cm^2/s
SPC [24]	2.27	65	3.85
TIP5P [19]	2.29	81.5	2.62
PPC [16]	2.52	77	2.6
Experimental	2.65, 3.0	78.4	2.30

Table 5: SPC is similar to the model used for our GROMACS tests; TIP5P uses five fixed partial charges; PPC is a polarizable model.

6. Conclusions

In this paper, we presented StreamMD – a molecular dynamic application developed for the fully programmable Merrimac streaming supercomputer. StreamMD performs the water-water interaction force calculation of the GROMACS molecular dynamic simulation package. It integrates with GROMACS through memory, and the interface is simply the molecules position array, neighbor-list stream, and the force array. The force computation presents implementation challenges on a high-performance, data-parallel SIMD architecture such as Merrimac, due to the variable nature of the neighbor interaction lists and the only moderate arithmetic intensity of the algorithm.

We developed several techniques for dealing with the variable number of interactions of each molecule. The simplest technique expands the interaction list by replicating the interacting molecule to form a molecule-pair for each interaction. This method suffers from poor locality leading to relatively low performance. A second variant of the algorithm reduces the bandwidth demands, while adhering to strict SIMD semantics, by forming fixed-length neighbor lists. To achieve this, a fixed-length list of 8 neighbors was chosen, central molecules were replicated, and dummy neighbors were introduced. The dummy neighbors lead to unnecessary computation but still performance improved by 46% compared to the first implementation. Finally, conditional streams were used to relax SIMD constraints on stream inputs, allowing for variable-length neighbor lists. The result is a fairly fast algorithm achieves 84% speedup over the original implementation, and performs at 50% of optimal. Our last implementation attempted to trade-off extra computation for reduced bandwidth demands in order to improve performance. The simplistic approach we implemented did not achieve higher performance with Merrimac's ratio of computational to memory bandwidths. However, we believe that this type of optimization is becoming increasingly important and plan to explore more sophisticated techniques.

Finally, we compared Merrimac's suitability for molecular dynamic applications against a conventional Pentium 4 processor. The unique architecture of Merrimac, which decouples memory and kernel execution, allows for highly effective static scheduling of kernel operations. This leads to very high performance on the critical inner loops of the computation without resorting to assembly coding, as was the case with GROMACS on the Pentium 4. Merrimac also provides much higher memory and computational bandwidths than a Pentium 4 built with equivalent semiconductor process technologies, and the software system is able to take advantage of these resources. Our automatically optimized code on Merrimac outperforms the highly hand-tuned Pentium implementation by a factor of 13.2.

7. REFERENCES

[1] Advanced Micro Devices, Inc., One AMD Place, P.O. Box 3453, Sunnyvale, California, USA. *3DNow! Technology Manual*, Mar. 2000. Order number 21928G/0.

[2] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32:406–424, 1953.

[3] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonté, J.-H. A., N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *SC'03*, Phoenix, Arizona, November 2003.

[4] P. N. Glaskowsky. Pentium 4 (partially) previewed. *Microprocessor Report*, August 28, 2000.

[5] P. N. Glaskowsky. IBM's PPC970 becomes Apple's G5. *Microprocessor Report*, July 7, 2003.

[6] GROMACS. GROMACS single processor benchmarks. http://http://www.gromacs.org/benchmarks/single.php.

[7] B. J. B. H. A. Stern, F. Rittner and R. A. Friesner. Combined fluctuating charge and polarizable dipole models: Application to a five-site water potential function. *J. Chem. Phys.*, 115:2237–2251, 2001.

[8] T. Halgren and W. Damm. Polarizable force fields. *Current opinion in structural biology*, 11(2):236–242, 2001.

[9] R. Ho, K. W. Mai, and M. A. Horowitz. The future of wires. *Proc. of the IEEE*, 89(4):14–25, April 2001.

[10] N. Jayasena, M. Erez, J. H. Ahn, and W. J. Dally. Stream register files with indexed access. In *Proceedings of the Tenth International Symposium on High Performance Computer Architecture*, Madrid, Spain, February 2004.

[11] U. J. Kapasi, W. J. Dally, S. Rixner, P. R. Mattson, J. D. Owens, and B. Khailany. Efficient Conditional Operations for Data-parallel Architectures. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 159–170, December 2000.

[12] U. J. Kapasi, P. Mattson, W. J. Dally, J. D. Owens, and B. Towles. Stream scheduling. In *Proceedings of the 3rd Workshop on Media and Streaming Processors*, pages 101–106, 2001.

[13] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *IEEE Computer*, August 2003.

[14] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, J. D. Owen, and B. Towles. Exploring the VLSI scalability of stream processors. In *Proceedings of the Ninth Symposium on High Performance Computer Architecture*, pages 153–164, Anaheim, California, USA, February 2003.

[15] K. Krewell. AMD serves up Opteron. *Microprocessor Report*, April 28, 2003.

[16] P. G. Kusalik and I. M. Svishchev. The spatial structure in liquid water. *Science*, 265:1219–1221, 1994.

[17] C. E. Leiserson. Fat-trees: Universal networks for hardware efficient supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, October 1985.

[18] D. B. Loveman. Program improvement by source to source transformation. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, pages 140–152. ACM Press, 1976.

[19] M. W. Mahoney and W. L. Jorgensen. A five-site model for liquid water and the reproduction of the density anomaly by rigid, nonpolarizable potential functions. *J. Chem. Phys.*, 112:8910–922, 2000.

[20] P. Mattson, W. J. Dally, S. Rixner, U. J. Kapasi, and J. D. Owens. Communication scheduling. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 82–92. ACM Press, 2000.

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

[22] Motorola, Inc. *AltiVec Technology Programming Interface Manual*. Motorola, Inc, 1999.

[23] H. Nada and J. P. J. M. van der Eerden. An intermolecular potential model for the simulation of ice and water near the melting point: A six-site model of H2O. *J. Chem. Phys.*, 118:7401–7413, 2003.

[24] G. W. Robinson, S. B. Zhu, S. Singh, and M. W. Evans. *Water in Biology, Chemistry and Physics: Experimental Overviews and Computational Methodologies*. World Scientific, Singapore, 1996.

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

[26] M. Taiji, T. Narumi, Y. Ohno, N. Futatsugi, A. Suenaga, N. Takada, and A. Konagaya. Protein explorer: A petaflops special-purpose computer system for molecular dynamics simulations. In *SC'03*, Phoenix, Arizona, November 2003.

[27] S. T. Thakkar and T. Huff. The Internet Streaming SIMD Extensions. *Intel Technology Journal*, (Q2):8, May 1999.

[28] D. van der Spoel, A. R. van Buuren, E. Apol, P. J. Meulenhoff, D. P. Tieleman, A. L. T. M. Sijbers, B. Hess, K. A. Feenstra, E. Lindahl, R. van Drunen, and H. J. C. Berendsen. *Gromacs User Manual version 3.1*. Nijenborgh 4, 9747 AG Groningen, The Netherlands. Internet: http://www.gromacs.org, 2001.

[29] P. J. van Maaren and D. van der Spoel. Molecular dynamics of water with novel shell-model potentials. *J. Phys. Chem. B*, 105:2618–2626, 2001.