

Scatter-Add in Data Parallel Architectures

Jung Ho Ahn, Mattan Erez and William J. Dally *
Computer Systems Laboratory
Stanford University, Stanford, CA 94305, USA
{gajh,merez,billd}@cva.stanford.edu

Abstract

Many important applications exhibit large amounts of data parallelism, and modern computer systems are designed to take advantage of it. While much of the computation in the multimedia and scientific application domains is data parallel, certain operations require costly serialization that increase the run time. Examples include superposition type updates in scientific computing and histogram computations in media processing. We introduce scatter-add, which is the data-parallel form of the well-known scalar fetch-and-op, specifically tuned for SIMD/vector/stream style memory systems. The scatter-add mechanism scatters a set of data values to a set of memory addresses and adds each data value to each referenced memory location instead of overwriting it. This novel architecture extension allows us to efficiently support data-parallel atomic update computations found in parallel programming languages such as HPF, and applies both to single-processor and multi-processor SIMD data-parallel systems. We detail the micro-architecture of a scatter-add implementation on a stream architecture, which requires less than 2% increase in die area yet shows performance speedups ranging from 1.45 to over 11 on a set of applications that require a scatter-add computation.

1. Introduction

The relative importance of multimedia and scientific computing applications continues to increase. These application domains exhibit large amounts of parallelism, and specifically data parallelism. As a result many modern computer systems are designed to exploit data level parallelism to achieve high performance. These data parallel architectures (DPAs) execute similar or identical operations concurrently on multiple data elements, allowing them to sustain high execution rates while tolerating long memory la-

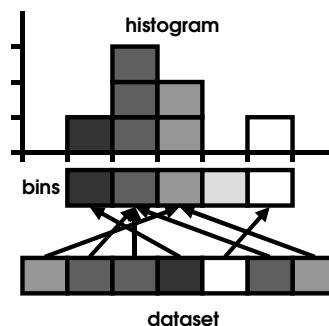


Figure 1: Parallel histogram computation leads to memory collision, when multiple elements of the dataset update the same histogram bin.

tencies. In this paper we will concentrate on the *single instruction multiple data* (SIMD) class of DPAs, exemplified by vector [9, 6], and stream processors [21, 10, 37].

While much of the computation of a typical multimedia or scientific application is indeed data parallel, some sections of the code require serialization which significantly limits overall performance. One commonly used algorithm that exemplifies this is a *histogram* or binning operation. Given a dataset, a histogram is simply the count of how many elements of the dataset map to each bin as shown in the pseudo-code below ¹.

```
for i=1..n
    histogram[data[i]] += 1;
```

Histograms are commonly used in signal and image processing applications to perform *equalization* and *active thresholding* for example. The inherent problem with parallelizing the histogram computation is *memory collisions* where multiple computations performed on the dataset must update the same element of the result in memory (Figure 1). One conventional way of dealing with this problem is to introduce expensive synchronization. Before a hardware processing element (PE) updates a location in mem-

* 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, and Northrup Grumman Corporation.

¹ In the example we refer to the case where the dataset is integer and corresponds directly to the histogram bins. In the general case, an arbitrary mapping function can be defined from the values in the dataset onto the integer bins.

ory which holds a histogram bin it acquires a lock on the location. This ensures that no other PE will interfere with the update and that the result is correct. Once the lock is acquired, the PE updates the value by first reading it and then writing back the result. Finally the lock must be released so that future updates can occur:

```
for i=1..n {
  j = data[i];
  lock(histogram[j]);
  histogram[j] += 1;
  unlock(histogram[j]);
}
```

This straightforward approach is complicated by the SIMD nature of the architectures under discussion that require very fine-grained synchronization, as no useful work is performed until all PEs have acquired and released their lock. To overcome this limitation, parallel software constructs, such as segmented scan[8], have been developed. However, the hardware *scatter-add* mechanism presented in this paper provides up to an order of magnitude higher performance in the case of a histogram operation and a 76% speedup on a molecular-dynamics application over a software-only approach. Moreover, these performance gains are achieved with a minimal increase in the chip area – only 2% of a 10mm×10mm chip in 90nm technology based on a standard-cell design. Scatter-add is a special type of memory operation that performs a data-parallel atomic read-modify-write entirely within the memory system, and is defined in the pseudo-code below. The first version of the operation atomically adds each value of an input data array to the memory location it accesses, while the second version performs a constant increment each time a memory location is accessed:

```
scatterAdd(<T> a[m], int b[n], <T> c[n]) {
  forall i = 1..n
    ATOMIC{a[b[i]] = (a[b[i]] + c[i])};
}

scatterAdd(<T> a[m], int b[n], <T> c) {
  forall i = 1..n
    ATOMIC{a[b[i]] = (a[b[i]] + c)};
}
```

where *a* is an *array* (contiguous memory locations) of length *m* and an integral or floating-point type; *b* is an integral array of length *n* whose elements are in the range [1..m] and defines the mapping of each element of *c* onto *a* (an index array); *c* is an array of length *n* and the same type as *a* or a scalar of the same type. Many, and potentially all, the updates can be issued concurrently and the hardware guarantees the atomicity of each update. The scatter-add is essentially a hardware implementation of the *array combining scatter* operation defined in High Performance Fortran (HPF) [17].

Applying the scatter-add to computing a histogram in a data-parallel way is straightforward, where *a* is

the histogram value, *b* is the mapping of each data element – simply the data itself, and *c* is simply 1:

```
scatterAdd(histogram, data, 1);
```

Relying on the memory system to atomically perform the addition, the histogram is computed without requiring multiple round-trips to memory for each bin update and without the need for explicit and costly synchronization of the conventional implementation. Also, the processor’s main execution unit can continue running the program, while the sums are being updated in memory using the dedicated scatter-add functional units. While these observations are true for a conventional scalar fetch-and-add, our innovative data parallel scatter-add extends these benefits to vector, stream, and other SIMD processors. This new approach is enabled by the ever-increasing chip gate count, which allows us to add floating point computation capabilities to the on-chip memory system at little cost.

Histogram is a simple illustrative example, but the scatter-add operation can also be used to efficiently express other operators as well. One such important operator is the superposition operator which arises naturally in many physical scientific applications. As explained in [33], due to the inherent linearity in the physical objects simulated and due to the linearization and simplification of nonlinear problems, superposition is a prevalent operation in scientific codes. Examples include particle-in-cell methods to solve for plasma behavior within the self-consistent electromagnetic field [42], molecular dynamics to simulate the movement of interacting molecules [11], finite element methods [7] and linear algebra problems [36].

The contributions of this paper include a design of an innovative hardware mechanism – scatter-add – that efficiently and effectively supports commonly used software constructs such as binning and superposition on data-parallel SIMD architectures; evaluating and contrasting the data-parallel performance of previous software only techniques with hardware scatter-add; showing the feasibility of scatter-add by detailing its micro-architecture, estimating its small die area requirements, and analyzing sensitivity to key system parameters and multi-node scalability.

The remainder of this paper is organized as follows: we discuss related work in Section 2, detail the scatter-add architecture in Section 3, present our evaluation results in Section 4, and conclude in Section 5.

2. Related Work

Previous work related to scatter-add falls into two main categories: software only techniques, and hardware mechanisms related mostly to control code and not for computation. In this paper we concentrate on data-parallel (SIMD, vector, or stream) architectures and do not discuss SPMD or MIMD techniques.

2.1. Software Methods

Implementing scatter-add without hardware support on a data parallel architecture requires the software to handle all of the address collisions. Three common ways in which this can be done are *sorting*, *privatization*, and *coloring*. The first option is to sort the data by its target address and then compute the sum for each address before writing it to memory. Many algorithms exist for data-parallel sort such as bitonic-sort and merge-sort, and the per-address sums can be computed in parallel using a *segmented scan* [8]. Both the segmented scan and the sort can be performed in $O(n)$ time complexity. While sorting is an $O(n \log n)$ operation in general, we only use the sort to avoid memory collisions, and process the scatter-add data in constant-sized batches. It is also important to note that the complexity constant for the sort is relatively large on a DPA. The second software option for scatter-add implementation is *privatization* [14]. In this scheme, the data is iterated over multiple times where each iteration computes the sum for a particular target address. Since the addresses are treated individually and the sums stored in registers, or other named state, memory collisions are avoided. This technique is useful when the range of target addresses is small, and its complexity is $O(mn)$, where m is the size of the target address range. The final software technique relies on *coloring* of the dataset, such that in each color only contains non-colliding elements [5]. Then each iteration updates the sums in memory for a single color and the total run-time complexity is $O(n)$. The problem is in finding a partition of the dataset that satisfies the coloring constraint, which often has to be done off-line, and the fact that in the worst case a large number of necessary colors will yield a serial schedule of memory updates.

Software algorithms have also been developed for scatter-add implementation in the context of coarse-grained multi-processor systems and HPF's array combining scatter primitive [2]. One such obvious technique is to equally partition the data across multiple processors, and perform a global reduction once the local computations are complete.

2.2. Hardware Methods

The NYU Ultracomputer [15] suggested a hardware fetch-and-add mechanism for atomically updating a memory location based on multiple concurrent requests from different processors. An integer-only adder was placed in each network switch which also served as a gateway to the distributed shared memory. While fetch-and-add could be used to perform general integer operations [4], its main purpose was to provide an efficient mechanism for implementing various synchronization primitives. For this reason it has been implemented in a variety of ways and is a standard hardware primitive in large scale multi-processor systems [22, 38, 26, 18, 43]. Our hardware scatter-add is a data-parallel, or vector, version of the simple fetch-and-add, which supports floating-point operations and is specifically designed for performing data

computation and handling the fine grained synchronization required by advanced SIMD memory systems.

Several designs for aggregate and combining networks have also been suggested. The suggested implementation for the NYU Ultracomputer fetch-and-add mechanism included a combining operation at each network switch, and not just at the target network interface. The CM-5 control network could perform reductions and scans on integral data from the different processors in the system, and not based on memory location [27]. Hoarse and Dietz suggest a more general version the above designs [19].

The alpha-blending mechanism [30] of application specific graphic processors can be thought of as a limited form of scatter-add. However, this is not a standard interpretation and even as graphic processors are implementing programmable features [39, 1], alpha-blending is not exposed to the programmer.

Finally, processor-in-memory architectures provide functional units within the DRAM chips and can potentially be used for scatter-add. However, the designs suggested so far do not provide the right granularity of functional units and control for an efficient scatter-add implementation, nor are they capable of floating-point operations. Active Pages [31] and FlexRAM [20] are coarse-grained and only provide functional units per DRAM page or more, and *C • RAM* [13] places a functional unit at each sense-amp. Our hardware scatter-add mechanism presented in Section 3 places the functional units at the on-chip cache or DRAM interfaces, providing a high-performance mechanism at a small hardware cost while using commodity DRAM.

3. Scatter-Add Architecture

In this section we present data-parallel architectures in greater detail, describe the integration of the scatter-add unit with a typical vector/stream processor and its micro-architecture, and explain the benefits and implications of the scatter-add operation.

3.1. Data Parallel Architectures

SIMD data parallel architectures exploit the abundance of data-level parallelism available in many important application classes. They offer significant cost/performance advantages over traditional systems which take advantage of instruction-level parallelism, but rely more on the programmer and software system.

The canonical SIMD DPA is composed of a set of processing elements (PEs), each with a local high bandwidth and low-latency data store, and a global memory space. The PEs in modern DPA systems are usually clustered into nodes (Figure 2), such that each node contains several PEs placed on a single chip, a part of the global memory, and an interface into the system-wide interconnection network for sharing data and implement-

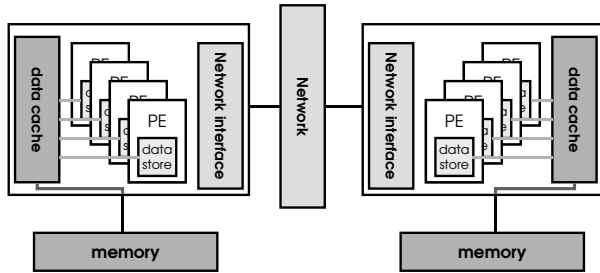


Figure 2: Clustered (SIMD) Data Parallel Architecture

ing the global memory². A cache is typically placed on the chip along with the PEs to act as a bandwidth amplifier and alleviate the restrictions due to off-chip signaling. Examples of classic DPAs are the fully SIMD ILLIAC IV [3] and the *vector* Cray-1 [35]. While these architectures (and vector processors in particular) have continued to evolve and play an important role in high performance computing [9, 6], the recent surge in interest in multi-media and scientific simulations has brought DPAs to the desktop. Most general-purpose processors now contain SIMD extensions [32, 28, 29, 40], and new *streaming* processors [21, 10, 16] and *programmable graphics* processors [39, 1] are starting to make an impact as well. There has also been interest in targeting vector processors for media application [25, 12].

In order to use the hardware resources of the DPA effectively, the computation is broken down into three phases:

1. In the *gather* phase data is collected from various locations in global memory, partitioned across the PEs, and each part is packed into a PEs local data store. This is done by a complex DMA operation or a vector-load.
2. During the *compute* phase all PEs concurrently operate on their share of the data and computation, consuming data and writing results from and into the local data store. Since data is provided by the low-latency and high-bandwidth local store, very high execution rates can be achieved.
3. Final results are written back into global memory in the *scatter* phase. The memory system again utilizes a DMA or vector-operation to perform the parallel memory write.

Parallelism is utilized not just for executing concurrent instructions across the PEs, but also to pipeline instruction execution on a single PE in order to amortize high-latency operations. In particular, the fact that memory operations are also expressed in parallel manner allows the memory system to pipeline these multi-word accesses. By pipelining these memory streams maximal bandwidth can be achieved from DRAM [34], and the long unpredictable latencies can be tolerated.

² In most systems of this type, strict SIMD is only enforced within a single processor.

3.2. Scatter-add Micro-architecture

As described earlier, the scatter-add operation is a parallel atomic read-modify-write operation. Another way of thinking of this operation is as an extension to the memory scatter operation where each value being written is summed with the value already in memory, instead of replacing it.

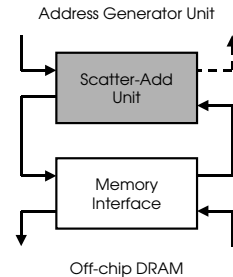


Figure 3: Scatter-add unit in the memory interface

The key technological factor allowing us to provide the scatter-add functionality in hardware is the rapid rate of VLSI device scaling. While a 64-bit floating-point functional unit consumed a large fraction of a chip in the past [24], corresponding area for such a unit in today's 90nm technology requires only 0.3mm². As a result we can dedicate several floating-point/integer adders to the memory system and allow it to perform an atomic read-modify-write, enabling the hardware scatter-add. A natural location for the scatter-add unit is at the memory interface of the DPA processor chip since all memory requests pass through this point (Figure 3). This configuration allows an easy implementation of atomic operations since the access to each part of global memory is limited to on-chip memory controller of that node (Figure 2). A further advantage of placing the scatter-add unit in front of the memory controller is that it can combine scatter-add requests and reduce memory traffic as will be explained shortly. Another possible configuration is to associate a scatter-add unit with each cache-bank of the on-chip cache (if it exists) as depicted in Figure 4a. The reasoning is similar to the one presented above as the on-chip cache also processes every memory request.

The Scatter-add unit itself consists of a simple controller with multiplexing wires, the functional unit that performs the integer and floating-point additions, and a *combining store* that is used to ensure atomicity as explained below (Figure 4b). The combining store is analogous to the *miss status handling register* (MSHR) and *write combining buffer* of memory data caches, and serves two purposes. First, it acts as an MSHR and buffers scatter-add requests until the original value is fetched from memory. Second, it buffers scatter-add requests while an addition, which takes multiple cycles, is performed. The physical implementation of the scatter-add unit is simple and our estimates for the area required are 0.2mm², and 8 scatter-add units require only 2% of a 10mm×10mm chip in 90nm technology based on a standard-cell design. The overheads of

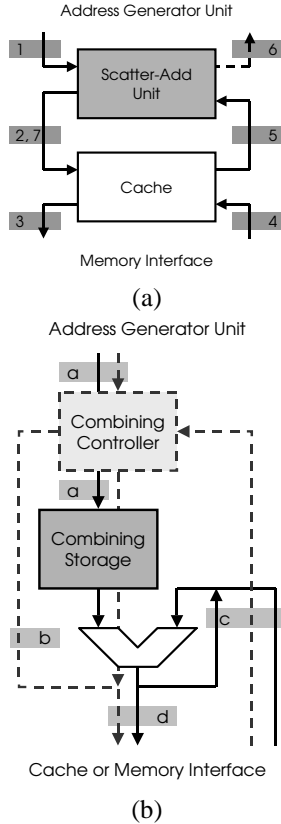


Figure 4: Scatter-add unit in a stream cache bank (a) along with the internal micro architecture (b). Solid lines represent data movement, and dotted lines address paths. The numbers correspond to the explanation in the text.

the additional wire tracks necessary for delivering scatter-add requests within the memory system are negligible. Our area analysis is based on the ALU implementation of Imagine [23] designed in a standard-cell methodology and targeting a latency of 4 1ns cycles. We also include the area required for the combining store and control logic.

We will explain how atomicity and high throughput are achieved by walking through the histogram example in detail. Programming the histogram application on a scatter-add enabled DPA involves a gather on the data to be processed, followed by computation of the mapping and a scatter-add into the bins:

```
gather(data_part, data);

forall i = 1..data_part_len
  bin_part[i] = map(data_part[i]);

scatterAdd(bins, bin_part, 1);
```

We will concentrate on the histogram computation performed by the scatter-add unit as depicted in Figure 4, where the number annotations will be referred to in the order of operations performed, solid lines represent data

movement, and dotted lines represent address paths. The process is also showed using a flow-diagram in Figure 5. The memory-system address generator of the DPA produces a vector (referred to as a stream in some architectures) of memory addresses corresponding to the histogram bins, along with a vector of values to be summed (simply a vector of 1s in the case of histogram). If an individual memory request that arrives at the input of the scatter-add unit (1) is a regular memory-write, it bypasses the scatter-add and proceeds directly to the cache and DRAM interface (2,3). Scatter-add requests must be performed atomically, and to guarantee this the scatter-add unit uses the combining store structure. Any scatter-add request is placed in a free entry of the combining store, if no such entry exists, the scatter-add operation stalls until an entry is freed. At the same time, the request address is compared to the addresses already buffered in the unit using a *content addressed memory* (CAM) (a). If the address does not match any active addition operations a request for the current memory value of this bin is sent to the memory system (b, 2, 3). If an entry matching the address is found in the combining store no new memory accesses are issued. When a current value returns from memory (4,5) it is summed with the combining store entry of a matching address (again using a CAM search) (c). Once the sum is computed by the integer/floating point functional unit an acknowledgment signal is sent to the address generator unit (6), and the combining store is checked once more for the address belonging to the computed sum (d)³. If a match is found, the newly computed sum acts as a returned memory value and repeats step c. If there are no more pending additions to this value, it is written out to memory (7). We are assuming that the acknowledgment signal is sent once the request is handled within the scatter-add unit. Since atomicity is achieved by the combining register, no further synchronization need take place. Using the combining register we are able to pipeline the addition operations achieving high computational throughput.

Multi-node Scatter-add

The procedure described above illustrates the use and details of the scatter-add operation within a single node of a DPA. When multiple nodes perform a scatter-add concurrently, the atomicity of each individual addition is guaranteed by the fact that a node can only directly access its own part of the global memory. The network interface directs memory requests to pass through the remote scatter-add unit where they are merged with local requests⁴. For multi-node configurations with local data-caches an optimization of this mechanism is to perform the scatter-add in two logical phases:

- A *local* phase in which a node performs a scatter-add

3 Only a single CAM operation is necessary if a simple ordering mechanism is implemented in the combining store
 4 The combining store will send an acknowledgment to the requesting node once an addition is complete

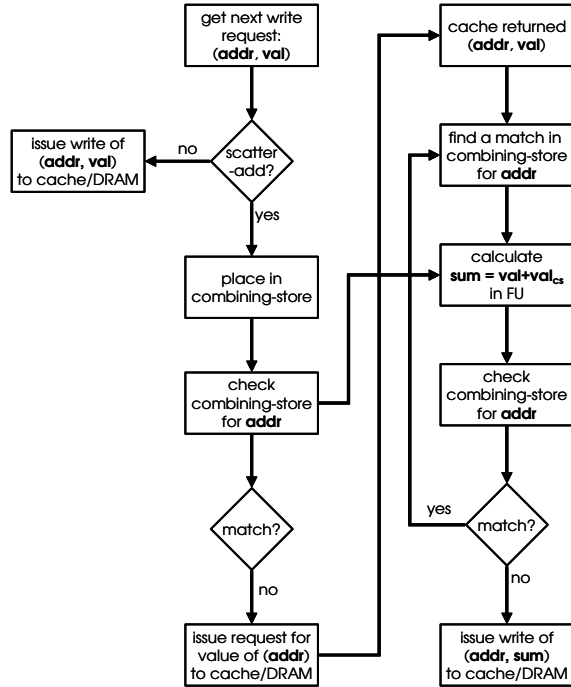


Figure 5: Flow diagram of a scatter-add request

on local and remote data within its cache. If a remote memory value has to be brought into the cache, it is simply allocated with a value of 0 instead of being read from the remote node.

- In the *global* phase the global scatter-add is computed by performing a *sum-back* of the cached values. A *sum-back* is similar to a cache write-back except that the remote write-request appears as a scatter-add on the node owning the memory address.

The global sum is continuously updated as lines are evicted from the different caches (via *sum-back*), and to ensure the correct final result a *flush-with-sum-back* is performed as a synchronization step once all nodes complete.

3.3. Scatter-add Usage and Implications

The main use of scatter-add is to allow the programmer the freedom to choose algorithms that were previously prohibitively expensive due to sorting, privatization complexity, and the additional synchronization steps required. Shifting computation to the scatter-add unit from the main DPA execution core allows the core to proceed with running the application, while the scatter-add hardware performs the summing operations. Also, the combining ability of the combining store may reduce the memory traffic required to perform the computation. The performance implications are analyzed in detail in Section 4.

A subtle implication of using a hardware scatter-add has to do with the ordering of operations. While the user coded the application using a specific order of data elements, the hardware reorders the actual sum computation due to the

pipelining of the addition operations and the unpredictable memory latencies when fetching the original value. It is important to note that while the ordering of computation does not reflect program order, it is consistent in the hardware and repeatable for each run of the program⁵. The user must be aware of the potential ramifications of this reordering when dealing with floating-point rounding errors and memory exceptions.

The scatter-add architecture is versatile and with simple extensions can be used to perform more complex operations. We will not describe these in detail, but will mention them below. A simple extension is to expand the set of operations handled by the scatter-add functional unit to include other commutative and associative operations such as min/max and multiplication. A more interesting modification is to allow a return path for the original data before the addition is performed and implement a parallel fetch-add operation similar to the scalar Fetch&Op primitive[15]. This data-parallel version can be used to perform parallel queue allocation on SIMD vector and stream systems.

4. Evaluation

To evaluate the hardware scatter-add mechanism we measure the performance of several applications on a simulated SIMD streaming system, with and without hardware scatter-add. We also modify the machine configuration parameters of the baseline system to test the sensitivity of scatter-add performance to the parameters of the scatter-add unit. Finally, we show that scatter-add is valuable in a multi-node environment as well.

4.1. Test Applications

The three applications we use were chosen because they require a scatter-add operation. Codes that do not have a scatter-add will run unaffected on an architecture with a hardware scatter-add capability. The inputs used for the applications below are representative of real datasets, and the dataset sizes were chosen to keep simulation time reasonable on our cycle accurate simulator.

Histogram – The histogram application was presented in the introduction. The input is a set of random integers chosen uniformly from a certain range which we vary in the experiments. The output is an array of bins, where each bin holds the count of the number of elements from the dataset that mapped into it. The number of bins in our experiments matches the input range. Scatter-add is used to produce the histogram as explained in Section 1.

Sparse Matrix-Vector Multiply – The sparse matrix-vector multiply is a key computation in many scientific applications. We implement both a *compressed sparse row* (CSR) and an *element by element* [36] (EBE) algorithm. The two algorithms provide different trade-offs between

5 This is not the case when dealing with multi-node scatter-add, but the lack of repeatability in multi-node memory accesses is a common feature of high performance computer systems.

amount of computation and memory accesses required, where EBE performs more operations at reduced memory demand. The dataset was extracted from cubic element discretization with 20 degrees of freedom using C^0 continuous Lagrange finite elements of a 1916 tetrahedra finite-element model. The matrix size is $9,978 \times 9,978$ and it contains an average of 44.26 non-zeros per row. In CSR all matrix elements are stored in a dense array, and additional information is kept on the position of each element in a row and where each row begins. The CSR algorithm is gather based and does not use the scatter-add functionality. EBE utilizes the inherent structure in the sparse matrix due to the finite-element model, and is able to reduce the number of memory accesses required by increasing the amount of computation and performing a scatter-add to compute the final multiplication result. Essentially in the EBE algorithm instead of performing the multiplication on one large sparse-matrix, the calculation is performed by computing many small dense matrix multiplications where each dense matrix corresponds to an element.

Molecular Dynamics – We use the non-bonded force calculation kernel of GROMACS [41]. This kernel calculates the interaction forces of water, and our simulation was performed on a sample of 903 water molecules for a single time-step. Experiments on the full GROMACS implementation on a Pentium 4 show that roughly 60–75% of the entire execution is spent running this kernel.

We implemented two versions of each application, one that uses the hardware scatter-add, and a second that performs a sort followed by a segmented scan. It is important to note that it is not necessary to sort the entire stream that is to be scatter-added, and that the scatter-add can be performed in batches. This reduces the run-time significantly, and on our simulated architecture a batch size of 256 elements achieved the highest performance. Longer batches suffer from the $O(n \log n)$ scaling of sort, while smaller batches do not amortize the latency of starting a stream operations and priming the memory pipeline. The sort was implemented using a combination of a bitonic and merge sorting phases. The histogram computation was also implemented using the privatization method.

4.2. Experimental Setup

Single node simulations were performed using a cycle-accurate simulator for a single node of the Merimac streaming supercomputer architecture [10]. Merimac is a stream processor with 16 data-parallel execution clusters, where each cluster can perform up to 4 floating point multiply-add operations per cycle. Merimac executes stream programs which are composed of computational kernels and streams of data. Data is transferred between memory and an on-chip software managed store, the stream register file (SRF), in granularity of entire streams which are typically hundreds to thousands of words long. An address partitioned on-chip data cache serves as a bandwidth amplifier for memory and also con-

Parameter	
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	1 GHz
Peak DRAM bandwidth	38.4 GB/s
Stream cache bandwidth	64 GB/s
Number of clusters	16
Peak floating point operations per cycle	128
SRF bandwidth	512 GB/s
SRF size	1 MB
Stream cache size	1 MB

Table 1: Machine parameters

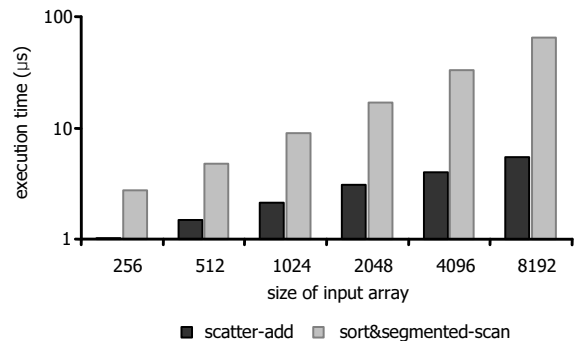


Figure 6: Performance of histogram computation for inputs of varying lengths and an input range of 2,048.

tains one scatter-add unit per cache bank. The exact details of the architecture are not described in this paper, and Table 1 summarizes the parameters of the base configuration. Our multi-node simulator was derived from this single-node simulator as explained in Section 4.5.

4.3. Scatter-add Comparison

The experiments below compare the performance of the hardware scatter-add to that of the software alternatives. We will show that hardware scatter-add not only outperforms our best software implementations, but that it also allows the programmer to choose a more efficient algorithm for the computation that would have been prohibitively expensive without hardware support.

Figure 6 shows the performance of the histogram computation using the hardware and software (sort followed by segmented scan) implementations of scatter-add. Both mechanisms show the expected $O(n)$ scaling, and the hardware scatter-add consistently outperforms software by ratios of 3-to-1 and up to 11-to-1.

While the software method using sort does not depend much on the distribution of the data, the hardware scatter-add depends on the range of indices. Figure 7 shows the ex-

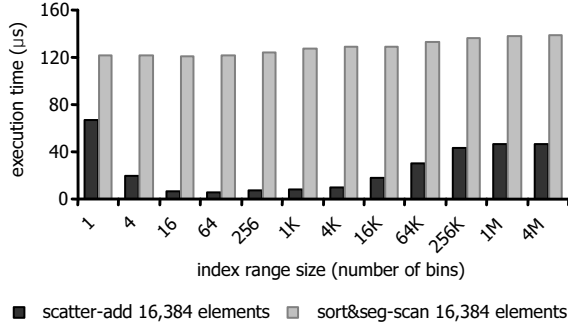


Figure 7: Performance of histogram computation for inputs of length 32,768 and varying index ranges.

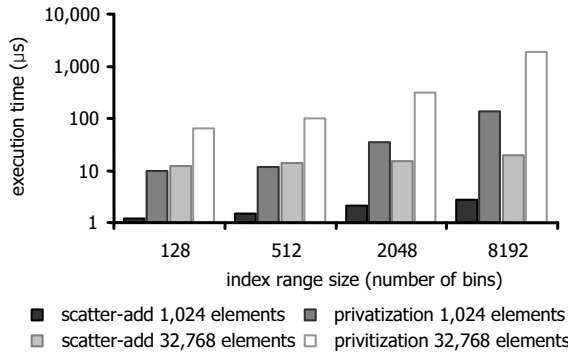


Figure 8: Performance of histogram computation with privatization for inputs of constant lengths and varying index ranges.

ecution time of hardware scatter-add and the sort&scan operations for a fixed dataset size of 32,768 elements and increasing index range sizes. The indices are uniformly distributed over the range. When the range of indices is small hardware performance is reduced due to the *hot bank effect*. The small index range causes successive scatter-add requests to map to the same cache bank, leaving some of the scatter-add units idle. As the range of indices increases, execution time starts to decrease as more of the units are kept busy. But when the index range becomes too large to fit in the cache, performance decreases significantly and reaches a steady-state. The sort&scan method performance is also lower for the large index ranges as a result of the larger number of memory references required to write out the final result.

Unlike sort, the runtime complexity of the privatization algorithm depends on the number of histogram bins. Figure 8 compares the performance of histogram using the hardware scatter-add to the privatization computation method, where the input data length is held constant at 1,024 or 32,768, and the range varies from 128 to 8,192. As the range increases, so does the performance advantage of the hardware scatter-add enabled algorithm, where speedups greater than an order of magnitude are observed for the large input ranges.

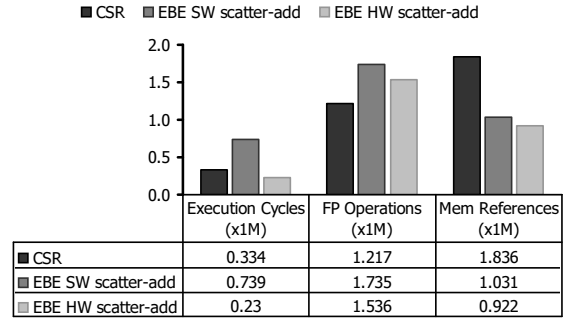


Figure 9: Performance of sparse matrix-vector multiplication.

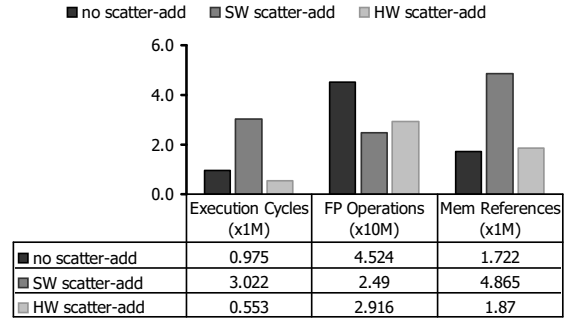


Figure 10: Performance of GROMACS.

Figure 9 shows the performance of the sparse matrix-vector multiplication. It is clearly seen that the hardware scatter-add enables the use of the EBE method, as without it CSR outperforms EBE by a factor of 2.2. With hardware scatter-add, EBE can provide a 45% speedup over CSR.

In Figure 10 we see how the hardware scatter-add influenced the algorithm chosen for GROMACS. The software only implementation with a sort&scan performed very poorly. As a result, the programmer chose to modify the algorithm and avoid the scatter-add. This was done by doubling the amount of computation, and not taking advantage of the fact that the force exerted by one atom on a second atom is equal in magnitude and opposite in direction to the force exerted by the second atom on the first. Using this technique a performance improvement of a factor of 3.1 was achieved. With the availability of a high-performance hardware scatter-add, the algorithm without computation duplication showed better performance, and a speedup of 76% over our best software-only code. Although we did not implement the entire GROMACS application, this kernel alone is responsible for 60–75% of the execution time on a Pentium 4. A similar speed-up on the Pentium 4 implementation would translate to a 35% minimum performance gain for the entire application.

4.4. Sensitivity Analysis

The next set of experiments evaluates the performance sensitivity to the amount of buffering in the combining store

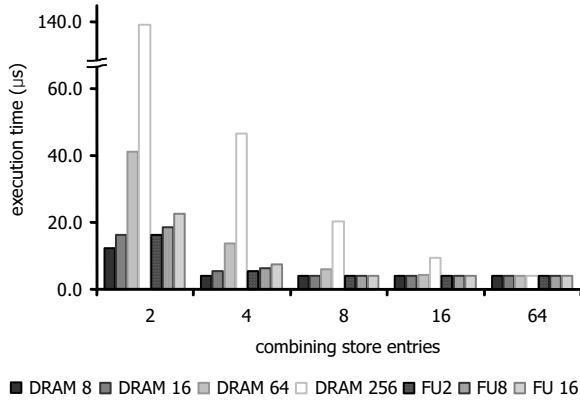


Figure 11: Histogram runtime sensitivity to combining store size and varying latencies.

(CS). In order to isolate and emphasize the sensitivity, we modify the baseline machine model and provide a simpler memory system. We run the experiments without a cache, and implement memory as a uniform bandwidth and latency structure. Throughput is modeled by a fixed cycle interval between successive memory word accesses, and latency by a fixed value which corresponds to the average expected memory delay. On an actual system, DRAM throughput and delays vary depending on the specific access patterns, but with memory access scheduling [34] this variance is kept small.

Figure 11 shows the dependence of the execution time of the histogram application using hardware scatter-add on the number of entries in the combining store, and on the latencies of the functional unit memory. Each group in the figure corresponds to a specific number of combining store entries, ranging from 2 to 64, and contains seven bars. The four leftmost bars in each group are for increasing memory latencies of 8–256 cycles and a functional unit latency of 4 cycles, and the three rightmost bars show the performance for different functional unit latencies with a fixed memory latency of 16 cycles. Memory throughput is held constant at 1 word every 2 cycles. The execution time is for an input set of 512 elements ranging over 65,536 possible bins. It can be clearly seen that even with only 16 entries in the combining store performance does not depend on ALU latency and is almost independent of memory latencies. With 64 entries, even the maximal memory latency can be tolerated without affecting performance.

The last sensitivity experiment gauges the scatter-add sensitivity to memory throughput. In Figure 12 we plot the histogram runtime vs. the combining store size and decreasing memory throughput. Memory throughput decreases as the number of cycles between successive memory accesses increases. Each group of bars is for a different number of combining store entries. Dark bars represent the performance of a histogram where the index range is 16, and light bars the case where the index range is 65,536. Within each group, the dark-light pairs are for a certain memory

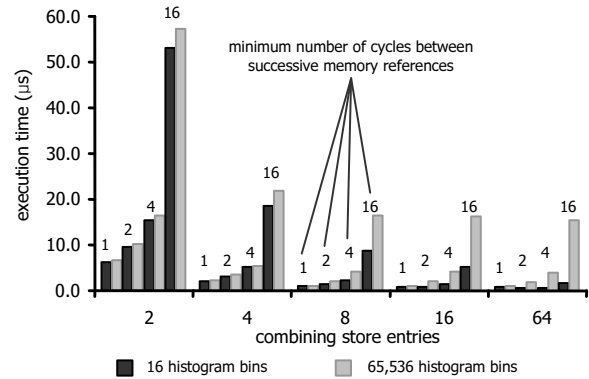


Figure 12: Histogram runtime sensitivity to combining store size and varying memory throughput.

throughput number. Even when a combining store of 64 entries cannot overcome the performance limitations of very low memory bandwidth. However, the effects of combining become apparent when the number of bins being scattered to is small. In this case, many of the scatter-add requests are captured within the combining store and do not require reads and writes to memory.

4.5. Multi-Node Results

Finally, we take a look at the multi-node performance of scatter-add with and without the optimizations described in Section 3.2. The multi-node system is comprised of 2–8 nodes, where each node contains a stream processor with the same parameters as used for the previous experiments, along with its cache and network interface as described in Section 3.1. The network we model is an input-queued crossbar with back-pressure. We ran experiments limiting the maximum per-node bandwidth to 1word/cycle (*low* configuration in the results below), and also 8words/cycle (*high* configuration) that are enough to satisfy scatter-add requests at full bandwidth. In addition to varying the network bandwidth we also turn the combining operation (Section refsec:multi-node) on and off. We concentrate on scatter-add alone and report the scalability numbers for only the scatter-add portion of the three test applications. For Histogram we ran two separate data-sets, each with a total of 64K scatter-add references: *narrow* which has an index range of 256, and *wide* with a range of 1M. GROMACS uses the first 590K references which span 8,192 unique indices, and SPAS uses the full set of 38K references over 10,240 indices of the EBE method (Section 4.1).

Figure 13 presents the scatter-add throughput (additions per cycle) achieved for scaling of 1–8 nodes, where each of the four sets of lines corresponds to a different application. The *wide* version of Histogram is limited by the memory bandwidth in both the single node case and for any number of nodes when using the high bandwidth network achieving perfect scaling. When the network bandwidth is low, performance is limited by the network and the application does not scale as well. The optimization of combining in the

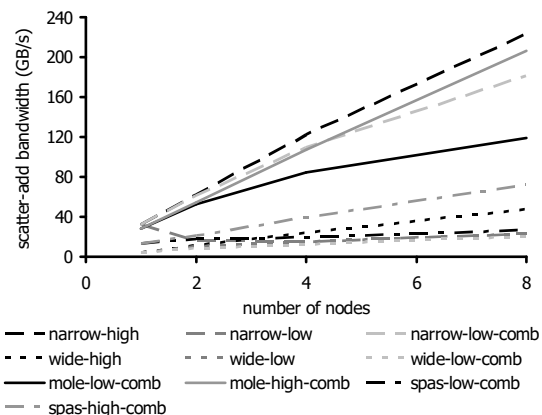


Figure 13: Multi-node scalability of scatter-add

caches does not increase performance due to the large range of addresses accessed, which lead to an extremely low cache hit rate. On the contrary, the added overhead of cache warm-up, cache combining, and the final synchronization step actually reduce performance and limit scalability.

In the *narrow* case of Histogram, the results are quite different as the high locality makes both the combining within the scatter-add unit itself and in the cache very effective. As a result, no scaling is achieved in the case of the low-bandwidth network, although with high bandwidth we see excellent speedups of up to 7.1 for 8 nodes. Employing the multi-node optimization of local combining in the caches followed by global combining as cache lines are evicted provided a significant speedup as well. The reduction in network traffic allowed even the low bandwidth configuration to scale performance by 5.7 for 8 nodes. Again, the overhead of warming up the cache, flushing the cache, and synchronization prevents optimal scaling.

The molecular dynamics application (GROMACS) exhibits similar trends to *narrow*, as the locality in the neighbor lists is high. Scaling is not quite as good due to the fact that the overall range of addresses is larger, and the address distribution is not uniform. SPAS experiences similar problems but its scaling trend is closer to that of *wide*. The large index range of both these applications reduces the effectiveness of cache combining due to increased overheads incurred on cache misses and line evictions. Increasing the network bandwidth limits the effect of this overhead and scaling is improved (*GROMACS-high-comb* and *SPAS-high-comb* in Figure 13).

5. Conclusion

In this paper we introduced the hardware scatter-add operation for data parallel SIMD architectures. Scatter-add allows global accumulation operations to be executed efficiently and in parallel, while ensuring the atomicity of each addition and the correctness of the final result. This type of operation is a primitive in High Performance Fortran, and is common in many application domains such as histogram

computations in signal and image processing, and expressing superposition in scientific applications for calculating interactions and performing algebraic computations.

We described the general scatter-add micro-architecture and estimated its area overhead as only 2% of a standard size die in 90nm technology. We also showed that the availability of this cheap hardware mechanism allows the programmer to choose algorithms that are prohibitively expensive to implement entirely in software. For example, in the molecular-dynamics application, use of the hardware scatter-add improved performance by 76% over our best data-parallel software version which was 3.1 times faster than the software implementation of the scatter-add operation. In addition when performing certain computations such as computing a histogram, the hardware approach provides an order of magnitude better performance than software alone.

This paper analyzed the performance implications of a single-node SIMD data parallel processor. We also described multi-node implementation and explored the scalability of scatter-add as well using on-chip caching to optimize multi-node performance. In future work we plan enhancements that will allow efficient computation of scans (parallel prefix operations) in hardware and implement system wide synchronization primitives for SIMD architectures. We are also considering an optimization to our multi-node cached algorithm that will arrange the nodes in a logical hierarchy and allow the combining across node to occur in logarithmic instead of linear complexity.

References

- [1] ATI. RADEON X800 Technology Specifications, 2004. <http://www.ati.com/products/radeonx800/specs.html>.
- [2] S. Bae, K. A. Alsabti, and S. Ranka. Array Combining Scatter Functions on Coarse-Grained, Distributed-Memory Parallel Machines, 1997.
- [3] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. The Illiac IV computer. *IEEE Transactions on Computers*, C-17:746–757, 1968.
- [4] J. Boisseau, L. Carter, K. Gatlin, A. Majumdar, and A. Snively. NAS benchmarks on the Tera MTA. In *Proceedings of the Multithreaded Execution Architecture and Compilation Workshop*, 1998.
- [5] L. Catabriga, M. Martins, A. Coutinho, and J. Alves. Clustered Edge-By-Edge Preconditioners for Non-Symmetric Finite Element Equations, 1998.
- [6] T. E. S. Center. Earth simulator. <http://www.es.jamstec.go.jp/esc/eng/ES/hardware.html>.
- [7] T. R. Chandrupatla and A. D. Belegundu. *Introduction to Finite Elements in Engineering*. Prentice Hall, 2002.
- [8] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan Primitives for Vector Computers. In *Supercomputing*, pages 666–675, 1990.
- [9] Cray Inc. Cray X1 System. <http://www.cray.com/products/systems/x1/>.

- [10] W. J. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonté, J. H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with Streams. In *SC2003*, Nov 2003.
- [11] E. Darve, M. Wilson, and A. Pohorille. Calculating Free Energies using a Scaled-Force Molecular Dynamics Algorithm. *Molecular Simulation*, 28(1–2):113–144, 2002.
- [12] K. Diefendorff. Sony’s Emotionally Charged Chip. *Microprocessor Report*, 13(5):6–11, Apr 1999.
- [13] D. Elliott, M. Snelgrove, and M. Stumm. Computational RAM: A memory-SIMD hybrid and its application to DSP. In *Custom Integrated Circuits Conference*, May 1992.
- [14] B. R. Gaeke, P. Husband, X. S. Li, L. Oliker, K. A. Yelick, and R. Biswas. Memory-Intensive Benchmarks: IRAM vs. Cache-Based Machines. In *International Parallel and Distributed Processing Symposium*, Apr 2002.
- [15] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer - designing an MIMD parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, Feb 1984.
- [16] T. R. Halfhill. Floating Point Buoys ClearSpeed. *Microprocessor Report*, Nov 2003.
- [17] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, High Performance Fortran Forum, Houston, TX, 1993.
- [18] W. D. Hillis and L. W. Tucker. The CM-5 Connection Machine: a scalable supercomputer. *Communications of the ACM*, 36(11):31–40, 1993.
- [19] Hoare and Dietz. A Case for Aggregate Networks. In *IPPS: 11th International Parallel Processing Symposium*, pages 162–166. IEEE Computer Society Press, 1998.
- [20] Y. Kang, W. Huang, S. M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. In *International Conference on Computer Design*, pages 192–201, Oct 1999.
- [21] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable Stream Processors. *IEEE Computer*, Aug 2003.
- [22] R. E. Kessler and J. L. Schwarzmeier. Cray T3D: a new dimension for Cray Research. In *COMPCON*, pages 176–182, Feb 1993.
- [23] B. Khailany. *The VLSI Implementation and Evaluation of Area- and Energy-Efficient Streaming Media Processors*. PhD thesis, Stanford University, Jun 2003.
- [24] L. Kohn and N. Margulis. Introducing the Intel i860 64-bit Microprocessor. *IEEE Micro*, 9(4):15–30, Aug 1989.
- [25] C. Kozyrakis. A Media-Enhanced Vector Architecture for Embedded Memory Systems. Technical report, University of California at Berkeley, 1999.
- [26] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, Jun 1997.
- [27] C. E. Leiserson, Z. S. Abuhmdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. S. Pierre, D. S. Wells, M. C. Wong-Chan, S.-W. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, 1996.
- [28] Motorola, Inc. *AltiVec Technology Programming Interface Manual*. Motorola, Inc, 1999.
- [29] S. Oberman, F. Weber, N. Juffa, and G. Favor. AMD 3DNow! Technology and the K6-2 Microprocessor. In *Hot Chips 10*, Aug 1998.
- [30] C. OpenGL Architecture Review Board. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.4*. Addison Wesley Professional, 2004.
- [31] M. Oskin, F. T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 192–203, 1998.
- [32] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, Aug 1996.
- [33] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in Fortran 90*. Cambridge University Press, 1996.
- [34] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 128–138, Jun 2000.
- [35] R. M. Russell. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, Jan. 1978.
- [36] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial & Applied Mathematics, second edition, 2003.
- [37] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, 2003.
- [38] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–36, 1996.
- [39] nVIDIA. nVIDIA GeForce 6 Technology Specifications, 2004. http://www.nvidia.com/object/geforce6_techspecs.html.
- [40] M. Tremblay, J. M. O’Connor, V. Narayanan, and L. He. VIS Speeds New Media Processing. *IEEE Micro*, 16(4):10–20, Aug 1996.
- [41] 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*. <http://www.gromacs.org>, 2001.
- [42] T. J. Williams. 3D Gyrokinetic Particle-In-Cell Simulation of Fusion Plasma Microturbulence on Parallel Computers. In *Internal Simulators Conference on High Performance Computing*, Mar/Apr 1993.
- [43] M. Woodacre, D. Robb, D. Roe, and K. Feind. The SGI Altix 3000 Global Shared-Memory Architecture. SGI White Paper, 2003.