# Executing Irregular Scientific Applications on Stream Architectures

Mattan Erez
The University of Texas at Austin

Jung Ho Ahn
Hewlett-Packard Laboratories

Jayanth Gummaraju
Mendel Rosenblum
William J. Dally
Stanford University

## ABSTRACT

The recent emergence of compute-intensive stream processors such as the Cell Broadband Engine, Stanford's Merrimac, and ClearSpeed's CSX600 has made them attractive platforms for scientific high-performance computing. Unstructured mesh and graph applications are an important class of numerical algorithms used in the scientific computing domain, which are particularly challenging for stream architectures. These codes have irregular structures where nodes have a variable number of neighbors, resulting in irregular memory access patterns and irregular control. We study four representative sub-classes of irregular algorithms, including finite-element and finite-volume methods for modeling physical systems, direct methods for n-body problems, and computations involving sparse algebra. We propose a framework for representing the diverse characteristics of these algorithms in the context of the unique properties of stream architectures, and demonstrate it using one representative application from each sub-class. We then develop techniques for mapping the applications onto a stream processor, placing emphasis on data-localization and parallelizations. Our simulations show that efficient stream hardware with restricted control abilities can effectively run challenging irregular applications with, for example, a finite element method and a molecular dynamic code sustaining $69$GFLOP/s and $46$GFLOP/s (64-bit) respectively using a single chip that measures $12mm$ on a side and consumes less than $70$W in $90nm$ technology.

## Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles

## General Terms

Performance

## Keywords

Stream Processors, Scientific Computing, Irregular Control

This work was supported, in part, by the Department of Energy ASCI Alliances Program, Contract LLNL-B523583, with Stanford University.



.

## 1. INTRODUCTION

Stream Processors were originally developed for media applications and have been shown to provide significant performance and efficiency gains over conventional architectures [26, 45]. Recently announced industry and academic Stream Processors, such as the Cell Broadband Engine™(Cell) [22], ClearSpeed's CSX600 [10], and Merrimac [12] are bringing similar gains to scientific computing.

Stream Processors achieve their efficiency and high performance advantage by relying on highly parallel processing elements (PEs), hardware structures that are tuned for minimal data-dependent control, and an explicitly software-managed storage hierarchy optimized for sequential access. While this set of characteristics is ideal for the regular execution traits common to media applications, mapping computations from scientific computing presents significant challenges.

In this paper we focus on mapping the irregular control and data access of unstructured mesh and graph algorithms to Stream Processors. This is an important class of scientific computation and includes finite element and finite volume methods (FEM and FVM), n-body simulations, and sparse algebra. We limit ourselves to algorithms that apply the same computation to each node or edge in the mesh or graph. There is a rich body of research on executing irregular applications on a variety of architectures. We draw from this prior work, adapt it, and evaluate new techniques in the context of the unique properties of stream architectures. First, Stream Processors have an explicitly software managed storage hierarchy that requires data to be *localized* to PE storage before computation can take place. Second, computation must be *parallelized* onto a substrate featuring limited hardware support for data-dependent branching and tightly coupled functional unit control. When used correctly, Stream Processors can provide much higher performance and efficiency than traditional architectures. In this paper we use the Merrimac infrastructure to evaluate our techniques and show, for example, that our finite element method and molecular dynamic codes can sustain $69$GFLOP/s and $46$GFLOP/s (64-bit) respectively using a single chip that measures $12mm$ on a side and consumes less than $70$W in $90nm$ technology.

Our contributions in this paper include:

- Proposing a framework for representing the properties of irregular unstructured mesh and graph applications in the context of Stream Processors (Section 3).
- Detailing the characteristics of stream architectures and their implications on the execution of irregular scientific codes using the above framework (Section 4).

- Developing both a methodology and techniques for mapping irregular applications onto Stream Processors concentrating on localization and parallelization, and describing the tradeoffs involved based on application properties (Section 5).
- Evaluating the performance of the mapping schemes on a set of representative irregular applications, and drawing conclusions on which architectural mechanisms are beneficial within the context of our framework (Section 6).

We also situate the constraints on mapping presented by stream architectures in the context of prior work (Section 2). The main conclusion of this work (see also Section 7) is that architectural features that relax the sequential access and partitioning restrictions along with prudent mapping choices allow very efficient hardware Stream Processor implementations to sustain very high performance on challenging applications. We summarize these points with decision trees for tuning applications to Stream Processors.

## 2. RELATED WORK

A large body of research exists on mapping irregular scientific applications to a wide variety of architectures including: distributed memory systems built from commodity parts such as Beowulf clusters [43] and grid computing [7]; shared memory multi-processors (e.g., [31]); vector processors (e.g., [37, 28]); and more exotic specialized architectures such as the Monsoon dataflow machine [34].

To the best of our knowledge the work in this paper is the first in-depth study of mapping irregular scientific applications onto Stream Processors. The particular case of sparse matrix-vector multiplication was also discussed in [47], where the application was mapped to the Cell processor, which shares many features and constraints with the stream architecture described in this paper.

Multi-processor systems benefit from, and in the case of distributed memory architectures require data localization into each processor's DRAM. Additionally, work must be distributed across all processors. Several runtime systems (e.g., CHAOS [23]) and languages (e.g., UPC [8], Titanium [48], and ZPL [16]) have been developed to address the challenge of efficient data communication and work partitioning with the large memories and coarse granularity of multi-processor systems. These systems often use inspector–executor methods [23, 44, 15] or data speculation [11] to hide data communication time on relatively low-bandwidth connectivities. They also employ multi-threaded methods (e.g., [21]) to partition work and load balance across full processor systems.

We deal with similar issues in data locality and work parallelization but in the context of the storage hierarchy and the constrained execution fabric of a Stream Processor. The problem we address is at a much smaller granularity of both control and storage sizes, as opposed to large scale multiple processor systems. Therefore, even though some concepts developed for large-scale systems could be applied to Stream Processors, the techniques cannot be used directly because the tradeoffs in bandwidths and memory usage are quite different.

Automatic data localization has been a topic of active research in the compiler community [19, 20, 32]. By examining the computation loops and the data access patterns the compiler can potentially rearrange computation using loop transformations (e.g, fission, permutation, tiling) or change data layout through data transformations (e.g., padding, transpose) to improve locality and reduce communication. These techniques could be incorporated into an optimizing stream compiler to automate several of the techniques discussed in this paper.

Stream Processors typically utilize *single instruction multiple data* (SIMD) execution and face similar challenges with executing irregular applications as other SIMD machines. In addition, we address and evaluate localization methods that match the restricted on-chip memory of Stream Processors, an issue that was not discussed in prior work. To the best of our knowledge previous work has focused on either completely general frameworks for executing irregular codes at a cost in performance, or on algorithm-specific solutions outside of a generalized framework. Our work combines multiple techniques that have been used in the past in different contexts into a single framework that is applicable to a large range of unstructured and irregular scientific applications. We give a detailed performance evaluation of such codes on Merrimac and Stream Processors in general.

A review of specific implementations of regular and irregular applications on both SIMD and MIMD (multiple instruction multiple data) systems is given in [17]. The paper also provides a taxonomy of parallel applications and the suitability of various algorithms to SIMD or MIMD. However, only restricted SIMD machines are evaluated that do not support the generalized stream execution model and relaxed SIMD found in Stream Processors.

Another general solution to executing irregular control with SIMD is to emulate MIMD execution through interpretation (e.g., [40, 41]). The idea is that all SIMD PEs emulate a processor pipeline, and the irregular application is compiled to this virtual ISA. Since emulation is done in software, the virtual ISA can be application specific and include coarse-grained operations that are free of irregular control. Note that this technique only works on architectures that provide independent memory addressing to each PE.

Finally, frameworks based on prefix-sum operations have also been used to implement irregular applications on data parallel systems [6]. However, implementing an entire algorithm in prefix-sum form requires significant restructuring and may incur heavy overheads compared to our direct implementations.

## 3. A FRAMEWORK FOR UNSTRUCTURED MESH AND GRAPH APPLICATIONS

Many scientific modeling applications contain irregular computation on unstructured mesh or graph data. The unstructured data is a result of efficient representations of complex physical systems. One cause of irregularity in computation and data accesses is a result of the graph structures used to represent the physical system model, and the spatial dependence on the topology of the model. In this section we develop a framework for representing such irregular computation applications and characterize their fundamental properties. We explore four representative applications from the FEM, FVM, direct n-body, and sparse algebra domains. In this paper we limit ourselves to irregularity that arises from the unstructured data and assume that the computation is identical for all interactions.

**FEM** Evaluates a 3D blast wave using a Discrete Galerkin finite element method for systems of nonlinear conservation laws. In our experiments we use the Euler equations with a piecewise linear approximation [5]. The computation consists of a face loop, which processes faces and the two elements that share each face, and an element loop which processes elements and their faces. The data structures for the face loop are stored as a directed graph with vertices (*nodes*) denoting faces and edges (*neighbors*) representing elements. The roles are reversed for the element loop, where elements are the nodes and faces the neighbors.

**CDP** Solves the transport advective equation for large eddy simulations (LES) using a finite volume method [35]. We evaluate the second-order WENO (Weighted Essentially Non-

```
for (i=0; i<num_nodes; i++) {
  process_node(nodes[i]);
  for (j=neighbor_starts[i]; j<neighbor_starts[i+1];
        j++) {
    process_neighbor(neighbors[neighbor_list[j]]);
  }
}
```

**(a)**

```
for (s=0; s<num_strips; s++) {
  for (i=node_starts[s]; i<node_starts[s+1]; i++) {
    process_node(nodes[i]);
    for (j=neighbor_starts[i]; j<neighbor_starts[i+1];
        j++) {
      process_neighbor(neighbors[neighbor_list[j]]);
    }
  }
}
```

**(b)**

**Figure 1: Pseudo-code for canonical irregular unstructured computation (a), and strip-mined version (b).**

Oscillatory) component of the solver. The WENO code contains two main loops, one loop over control volumes and one over faces. As with FEM, control volumes act as nodes and faces as neighbors during the control volume loop, and vice versa for the face loop.

**GROMACS** Simulates Newton's equations of motion for large n-body systems. The code is specialized for bio-molecules, and we implemented the non-bonded direct water-water interaction computation of the GROMACS package [46]. This algorithm uses an efficient cut-off approximation, where water molecules contain a list of other molecules that they interact with. Therefore, water molecules act as both nodes and neighbors, and edges in the graph imply an interaction.

**SPAS** Computes a sparse algebra matrix vector multiplication [18]. In this paper we explore the compressed sparse row storage scheme, where rows are treated as nodes and vector elements as neighbors, i.e., rows are vertices in the graph data structure and edges are vector elements.

A canonical irregular unstructured application consists of potentially multiple phases that process the nodes and neighbors in the graph representing the data and computation, as shown in Figure 1a. Typically, scientific computation is *strip-mined* [33] – partitioned into subcomputations that each processes a portion of the data (Figure 1b). The amount of computation and data accesses required for the processing functions, as well as the connectivity properties of the graph affect how an algorithm should be mapped to stream processors.

## 3.1 Application Parameter Space

Table 1 lists the properties of the applications above using a small representative data set for each application.[1] The properties are discussed in detail in the following subsections.

### *3.1.1 Mesh Connectivity Degree and Variability*

The *connectivity degree*, i.e, the number of neighbors of each node, is an important metric which affects the locality of data access, the amount of communication, and the style of implementation on a stream architecture. A high connectivity degree tends to place greater pressure on the processor's communication and on-chip storage resources, because data for *all* neighbors must be read in advance before processing a node when using the stream execution model [26]. A large number of neighbors also deemphasizes the arithmetic properties of the node processing (columns 5 and 7 in Table 1). The applications based on element meshes have a low degree, whereas n-body and sparse algebra tend to have a large number of average neighbors. GROMACS in particular has a very large number of potential neighbors, where over 5% of nodes have more than 80 neighbors.

An important property associated with the degree of connectivity is its *variability* (shown as the connectivity standard deviation in column 4). With a fixed connectivity degree, as in the face loops of FVMs and FEMs, the computation of all nodes typically follows the same control path, simplifying the mapping onto a stream processor. In contrast, a high variability, exemplified by the GROMACS n-body algorithm, poses a greater challenge (Subsections 4.3 and 5.2).

Connectivity degree and its variability also influence the amount of locality in the graph that can be exploited. By *locality* we refer to *data reuse* – the number of times each data item (neighbor or node) is accessed during execution. A high connectivity implies large amounts of reuse, whereas low connectivity limits locality. High connectivity variance may make locality in the graph more difficult to detect and may limit the amount of reuse that can be easily exploited in the application (Subsection 5.1).

### *3.1.2 Arithmetic Intensity*

We represent the computational properties of the application inner loops using four parameters: the number of arithmetic operations performed on each individual node and neighbor (`process_node` and `process_neighbor` in Figure 1 and columns 5–6 in Table 1); and *arithmetic intensity* – the ratio between arithmetic operations and the number of input and output words required to compute them in the loop (columns 7–8).

Computations with a low operation count often do not utilize the many functional units of a stream processor well because compiler techniques, such as software pipelining and loop unrolling [30], are hindered by the irregularity of control and data-dependent loop bounds seen in the canonical pseudo-code representation (see also Section 6.2). A low arithmetic intensity also reduces inner loop execution performance as it is limited by on-chip memory bandwidth. Additionally, applications with low arithmetic intensity may require sophisticated software mapping techniques or hardware structures to reduce off-chip bandwidth demands and avoid stalls while waiting on off-chip memory.

### *3.1.3 Runtime Properties*

A final property of unstructured mesh and graph applications is the rate at which the connectivity graph evolves in relation to the algorithm time steps (column 9 of Table 1). While some applications, such as FEM and SPAS have a fixed connectivity, other applications evolve their graph structure over time. For example, CDP occasionally applies *adaptive mesh refinement* to account for changes in the physical model, and GROMACS recomputes the molecule interaction list every several ($\sim 10$) time steps.

# 4. STREAM ARCHITECTURES

In this section we explain the key properties of the stream architecture relating to locality, memory, and parallelism, and outline the challenges they present for efficient execution of unstructured irregular computation.

Modern VLSI technology enables high single-chip arithmetic

[1]Datasets were purposefully kept small to allow for extensive evaluation using a cycle-accurate stream processor simulator.

| (1) Application | (2) #nodes | (3) Average Degree | (4) Degree Standard Deviation | (5) Ops/Node | (6) Ops/Neighbor | (7) Arithmetic Intensity (Node) | (8) Arithmetic Intensity (Neighbor) | (9) Graph Evolution Timescale [steps] |
|---|---|---|---|---|---|---|---|---|
| **FEM (elements)** | 9,664 | 4 | 0 | 1 | 2 | 0.07 | 2.8 | $\infty$ |
| **FEM (faces)** | 19,328 | 2 | 0 | 2 | 159 | 1.6 | 22.3 | $\infty$ |
| **CDP (elements)** | 5,431 | 5.6 | 0.8 | 4 | 11 | 0.3 | 2.2 | $10^5$ |
| **CDP (faces)** | 30,180 | 2 | 0 | 0 | 22 | N/A | 1.8 | $10^5$ |
| **GROMACS** | 4,114 | 26.6 | 27.1 | 18 | 217 | 0.25 | 12.1 | $10^1$ |
| **SPAS** | 7,728 | 13.6 | 3.4 | 1 | 2 | 0.5 | 2.0 | $\infty$ |

**Table 1: Properties of Scientific Applications**

performance by allowing for a large number of high-throughput functional units on a single die. For example, Merrimac [12] and Cell [22] can sustain 128GFLOP/s (double-precision) and 256GFLOP/s (single precision) respectively, requiring over 3TB/s of operand bandwidth. The challenge for these architectures is maintaining such large instruction and operand throughput. These *stream architectures* rely on exploiting locality in the computation to provide for sufficient operand bandwidth using a rich storage hierarchy, and on taking advantage of parallelism to drive the many functional units and tolerate latencies.

## 4.1 Locality

As VLSI technology scales, bandwidth decreases as the distance an operand traverses from storage to functional unit grows [13]. Therefore, to support high sustained operand throughput, stream architectures are strongly partitioned into PEs, which are clusters of functional units with a hierarchy of storage (Figure 2).

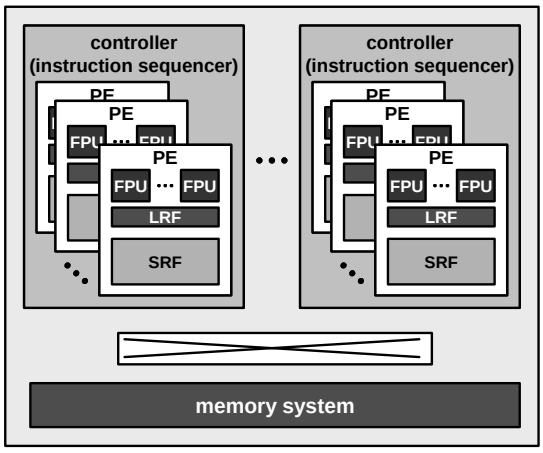



**Figure 2: Canonical stream architecture partitioned into multiple PEs with a rich storage hierarchy.**

The lowest level in the storage hierarchy is the *local register file* (LRF) of each PE, which is directly connected to the functional units over short high bandwidth wires. The LRF exploits short term producer-consumer locality between instructions within a function or inner-loop body [12]. In Merrimac, the LRF is able to sustain the operand throughput demands of the 16 PEs which combine for 64 floating point units (FPUs).

The second level in the storage hierarchy is the *stream register file* (SRF), which is a fast local memory structure. Merrimac's SRF has a 1MB capacity and is partitioned across the PEs. We refer to the 64KB portion of the SRF local to a PE as an *SRF lane*. Being local to a PE enables the SRF to supply data with a high 512GB/s throughput on a Merrimac chip.

The SRF has a local address space in the processor that is separate from global off-chip memory, and requires software to *localize* data into the SRF before computation begins. Explicitly managing this distinct space in software enables two critical functions in the stream architecture. First, the SRF can reduce demands on off-chip memory by providing an on-chip workspace and explicitly capturing long term producer-consumer locality between functions and loops. Second, explicit control allows software to stage large granularity asynchronous bulk transfers into and from the SRF. Doing this allows for a throughput optimized memory system and shifts the responsibility of tolerating the ever-growing memory latencies from hardware to software. Relying on software controlled bulk transfers to hide latency and restricting PE memory access to the local SRF address space result in predictable memory accesses within the execution pipeline. This enables optimized static compiler scheduling of arithmetic operations onto the area and power efficient PE computational fabric.

Beyond the SRF is a global on-chip level of the storage hierarchy composed of an interconnection network that handles direct communication between the PEs and connects them with the memory system. It is possible to use this interconnect to perform dynamic *cross-lane* SRF indexed accesses between PEs in addition to the efficient *in-lane* accesses described above. As discussed in [24], cross-lane indexing incurs both additional hardware complexity and runtime overhead.

### *4.1.1 Locality Implications*

The stream programming model [26] is designed for architectures that include the distinct SRF address space and advocates a *gather–compute–scatter* style, where overall execution is broken down into computational strips as with the strip-mining technique [33]. A prototypical example of a strip-mined program is shown Figure 1b. However, requiring all data that an inner-loop computation accesses to be gathered ahead of the computation poses a problem for the irregular accesses of unstructured mesh algorithms. Additional complexity relates to the tradeoffs of indexed vs. sequential access to the SRF, and whether cross-lane accesses are employed.

## 4.2 Memory System

Stream memory systems provide high throughput of multiple words per cycle using conventional DRAM technology by processing all requests in bulk asynchronous DMA-style transfers [3, 9]. One or more DMA engines process bulk stream requests from the processor and issue them as potentially reordered fine-granularity word accesses to memory. This allows the memory system to support complex addressing modes including gathers, scatters, and atomic data-parallel read-modify-writes (scatter-add) [2].

As with a conventional memory system, a hardware managed cache can also be added between the processor and memory. In the case of a stream processor the *stream cache* backs up the SRF and acts as a bandwidth accelerator and is not used for reducing memory access latency. A cache can be beneficial for unstructured mesh applications because of the inherent locality of nodes sharing neighbor data. However, introducing a cache requires additional

hardware area, which offers a tradeoff between dedicating on-chip memory to the SRF or the cache. Note that not all bulk stream memory transfers are cached. We reduce *cache pollution* effects by limiting caching to those gather/scatter accesses that exhibit temporal locality.

#### 4.2.1 Memory System Implications

Utilizing the memory system for scientific applications is made simple by the stream programming model. We must still consider the tradeoff of using a stream cache. Reducing the SRF to accommodate a cache limits the space available for data and results in shorter strips in the strip-mined computational loop.

### 4.3 Parallelism

Effectively utilizing the many FPUs of a stream processor requires extensive use of parallelism: FPUs must be run concurrently on distinct data; FPUs must be provided with instructions; small execution pipeline latencies must be tolerated; and long memory latencies must be overlapped with computation.

In the case of our applications, and in scientific computing in general, *data level parallelism* (DLP) is abundant, as the same type of computation is applied to all nodes. DLP can be exploited both within a PE and across PEs. In addition to DLP, a small amount of *instruction level parallelism* (ILP) across independent instructions within a loop or function body is also available.

For concurrent execution and instruction supply there are three main hardware mechanisms available for controlling the FPUs and PEs:

- *Single instruction single data* (SISD) execution, exemplified by *VLIW* and *superscalar*, issues distinct instruction to several FPUs. This technique can utilize ILP , but is limited to controlling a small number of FPUs because the cost of storing and issuing individual instructions does not scale.
- *Single instruction multiple data* (SIMD/vector) applies the same computation to a group of data across multiple FPUs. SIMD hardware can efficiently supply instructions to a very large number of FPUs as a single operation is performed on a collection of data. However, all the SIMD-controlled FPUs are restricted to a single instruction sequence.
- *Multiple instruction multiple data* (MIMD) utilizes multiple instruction sequencers for executing independent threads of control. To amortize the hardware cost of a sequencer, this mechanism is typically combined with SIMD and ILP execution [1].

In this paper we use the Merrimac infrastructure for evaluation. Merrimac, has a single instruction sequencer that controls 16 PEs in SIMD fashion. Each PE's 4 FPUs execute VLIW instructions scheduled by the compiler.

#### 4.3.1 Parallelism Implications

The use of SIMD in stream processors poses problems for irregular unstructured computation. We cannot trivially map the computation of Figure 1 as it requires different control paths for different nodes as the number of neighbors per node varies.

## 5. MAPPING IRREGULAR COMPUTATION TO STREAM PROCESSORS

In this section we detail the basic algorithm classes used to perform localization (Subsection 5.1) and parallelize irregular computation (Subsection 5.2) on stream processors.

### 5.1 Localization

As discussed in Subsection 4.1 stream processors cannot make arbitrary memory references from within the PEs, and can only access the SRF local memory. Unlike traditional processors, the local memories have distinct name-spaces and data elements must be *renamed*, or *localized* to the local address space. This is similar to the requirements of distributed memory parallel systems, although the granularity of control and degree of runtime overhead that can be tolerated is much lower in the case of stream processors.

The renaming process entails re-writing the neighbor-list pointers to local memory addresses in a processing step separate from the actual computation. This process is similar to the *inspector–executor* model [39, 29]. While renaming is performed, additional optimizations can be applied. We first describe the basic localization scheme that has minimal preprocessing overhead. We then discuss an optimization that removes duplicate elements referred to by the neighbor list.

#### 5.1.1 Basic Renaming

With *non-duplicate-removal renaming* (nDR), each element of the neighbor list is assigned a unique location in the SRF. This amounts to dereferencing the neighbor-list pointers as they are transferred into the SRF. As a result, duplicate entries may be allocated for a neighbor that is referred to multiple times in the neighbor list. The computational inner loop processes the neighbor data in order out of the SRF as shown in Figure 3. The number of neighbor data elements transferred and stored in the SRF is equal to the total length of the neighbor list.

```
1  for (s=0; s<num_strips; s++) {
2    strip_neighbor_start = neighbor_starts[node_starts[s]];
3    strip_num_neighbors  = neighbor_starts[node_starts[s+1]] -
4                           neighbor_starts[node_starts[s]];
5    // localize data (gather performed by
6    // the memory system)
7    for (i=0; i<strip_num_neighbors; i++) {
8      local_nl[i] =
9        neighbor_data[neighbor_list[strip_neighbor_starts+i]];
10   }
11   for (i=0; i< (node_starts[i+1]-node_starts[s]); i++) {
12     local_nodes[i] = node_data[i+node_starts[s]];
13   }
14   for (i=0; i< (node_starts[i+1]-node_starts[s]); i++) {
15     local_nn[i] = neighbors_per_node[i+node_starts[s]];
16   }
17
18   // process from local memory
19   for (i=0; i< (node_starts[i+1]-node_starts[s]); i++) {
20     n_ptr=0; // current location in local_nl
21     process_node(local_nodes[i]);
22     for (j=0; j<local_nn[i]; j++) {
23       process_neighbor(local_nl[n_ptr++]);
24     }
25   }
26 }
```

**Figure 3: Pseudo-code for nDR processing (`local_` prefix refers to data that has been localized to the PE).**

The main advantage of this scheme is that its implementation is independent of the actual connectivity in the dataset. Each neighbor-list element is simply loaded in neighbor-list order into the SRF; no preprocessing is required as the pointers are dereferenced by the memory system and are not rewritten. Additionally, the PEs access data in the local memories in a streamed fashion utilizing the optimized SRF hardware (Subsection 4.1).

The disadvantages stem from the potentially large number of duplicates created when nodes share the same neighbors. First, keeping multiple copies of data in the SRF reduces the size of the execution strips increasing the overheads associated with starting stream

operations and priming software-pipelined loops. Second, creating the copies in the SRF requires multiple reads of the same data from the memory system. This extra communication is particularly detrimental when memory bandwidth is a performance-limiting factor.

### 5.1.2 Renaming with Duplicate-Removal

The second renaming option explicitly renames the pointers in the neighbor list to on-chip addresses, allowing duplicate data to be eliminated and potentially reducing off-chip memory bandwidth and increasing the effective SRF size and computation strip lengths. This *duplicate removal* (DR) renaming is in contrast to nDR, which discards the pointers by dereferencing all neighbor data directly to the SRF. These benefits, however, come at a cost of PE execution overheads and the run-time processing of the neighbor-list. The pseudo-code for the canonical irregular computation with DR is shown in Figure 4.

```
1  // remove duplicates
2  // reads neighbor list and produces
3  // duplicate removed data (DR_data), the
4  // rewritten pointers (DR_nl),
5  // and the associated stripping information
6
7  DR(neighbor_list, output_DR_nl, output_DR_data,
8          output_DR_starts);
9  // see Algorithm 1 for sample pseudo-code for DR()
10
11 for (s=0; s<num_strips; s++) {
12   strip_neighbor_start = neighbor_starts[node_starts[s]];
13   strip_num_neighbors  = neighbor_starts[node_starts[s+1]] -
14                          neighbor_starts[node_starts[s]];
15   // localize data (gather performed by
16   // the memory system)
17   for (i=0; i<strip_num_neighbors; i++) {
18     local_DR_nl[i] =
19         output_DR_nl[neighbor_list[strip_neighbor_starts+i]];
20   }
21   for (i=0; i< (node_starts[i+1]-node_starts[s]); i++) {
22     local_nodes[i] = node_data[i+node_starts[s]];
23   }
24   for (i=0; i< (node_starts[i+1]-node_starts[s]); i++) {
25     local_nn[i] = neighbors_per_node[i+node_starts[s]];
26   }
27   for (i=0;
28         i<(output_DR_starts[s+1]-output_DR_starts[s];
29         i++) {
30     local_DR_data[i] = output_DR_data[i+output_DR_starts[s];
31   }
32
33   // process from local memory
34   for (i=0; i< (node_starts[i+1]-node_starts[s]); i++) {
35     n_ptr=0; // current location in local_nl
36     process_node(local_nodes[i]);
37     for (j=0; j<local_nn[i]; j++) {
38       process_neighbor(local_DR_data[local_DR_nl[n_ptr++]]);
39     }
40   }
41 }
```

**Figure 4: Pseudo-code for DR processing (`local_` prefix refers to data that has been localized to the PE).**

PE execution overheads stem from performing indexed accesses into the SRF, as opposed to more hardware optimized sequential accesses as in nDR; and are discussed in Section 6.2.

Renaming the pointers requires processing the neighbor list and allocating SRF space based on the actual connectivity, leading to a computational overhead every time the connectivity list is modified. A simple algorithm for DR is shown in Algorithm 1.

The efficiency of the DR method, in terms of reducing SRF space usage and off-chip bandwidth depends on the mesh locality properties. If the mesh has high locality and a high connectivity degree large amounts of neighbor storage can be removed.

**Algorithm 1** Sample algorithm for DR renaming

**Require:** neighbor list (nl)
**Ensure:** renamed neighbor list (DR_nl)
**Ensure:** unique neighbor data (DR_data)

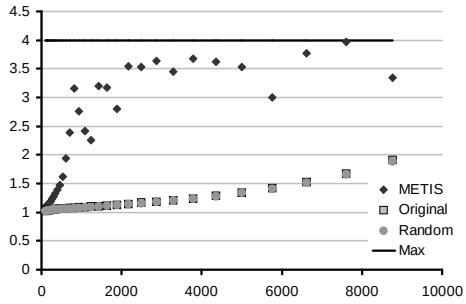
```
clear the renaming_map map
for P in { neighbor list } do
  if P in renaming_map then
    write renaming_map[P] to renamed list
  else
    if cannot allocate more SRF space then
      mark strip end
      reset SRF allocation
      clear renaming_map
    allocate new SRF location for neighbor corresponding to P: P'
    assign renaming_map[P] = P'
    write P' to renamed list
```

Figure 5 presents the locality available in four of the unstructured meshes as a function of available SRF space. In each figure, the x-axis is the number of neighbor elements that fit in one strip in local memory, and the y-axis represents locality – the average number of times each neighbor element is reused while computing on the strip. Each of the four sub-figures shows the locality for the original ordering (light squares) as well as an improved ordering obtained by domain decomposition using METIS [27] (solid diamonds) and a randomized order (light circles). We can see that locality increases significantly with the size of the strip and Subsection 6.3 explores this further. Second, the amount of reuse is strongly related to the connectivity degree of the mesh. In the case of GROMACS, for example, neighbors may be reused more than 20 times on average, whereas the face loop of FEM peaks out at a reuse of 2. An attempt to improve locality with METIS has mixed results. The element loops of CDP and FEM benefit from reordering, the face loop of FEM gains little, and locality in GROMACS is sacrificed. Looking at the randomized ordering, we see that METIS does discover partitions that improve locality. However, the original ordering of the GROMACS dataset is based on a geometric partition that is well suited to the molecular dynamics problem. METIS, it seems, does not deal well with the very high degree and tightness of the connectivity in GROMACS.

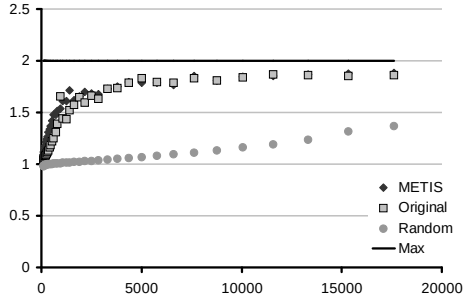
As in traditional processors, placing a cache between off-chip memory and the SRF reduces the potential bottleneck associated with limited DRAM throughput. A cache can eliminate unnecessary data transfers by supplying them from an associative on-chip memory. As discussed in Subsection 4.2, however, the on-chip memory for the cache comes at the expense of SRF space and reduces computational strip lengths. Additionally, while off-chip bandwidth is reduced, on-chip memory utilization is reduced because the PEs must copy the cached data into their local SRF lane.

### 5.1.3 Dynamic Renaming

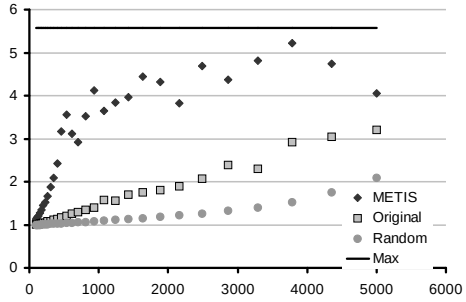
Localization and renaming can also be performed dynamically, for example following the inspector–executor methodology of CHAOS [23]. We expect the overhead of this technique to be very large in the context of the compute intensive and simple control characteristics of stream processors, and a full evaluation is beyond the scope of this paper. In software, dynamic renaming requires building a dynamic renaming hash structure in expensive on-chip memory, introducing high execution overheads and reducing effective strip size and locality. Hardware dynamic renaming does not apply to stream processors and is equivalent to a conventional cache that transparently merges on-chip and off-chip memory into a single address space.



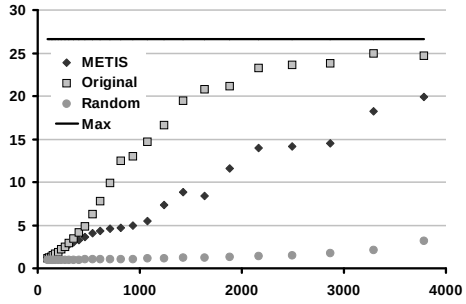

(a) FEM elements, $9,664$ nodes.




(b) FEM faces, $20,080$ nodes.




(c) CDP, $5,431$ nodes.




(d) GROMACS, $4,114$ nodes.

**Figure 5: Locality (reuse) as a function of strip size with the original ordering in the dataset, with METIS reordering for locality, and with a randomized order (horizontal axis – strip size in number of neighbors; vertical axis average number of accesses per neighbor).**

## 5.2 Parallelization

To fully utilize a stream processor the computation must be parallelized to take advantage of the multiple PEs. An irregular application poses tradeoffs with regards to both hardware and software of a streaming system. Below is a description of the classes of mappings that can be used to parallelize an irregular mesh computation on a stream processor. The assumptions are that the execution model is a data-parallel *single program multiple data* (SPMD) model, where all PEs are dedicated to the same computational task. In our applications, a global reduction operation occurs between different computational tasks, preventing more flexible execution models.

We classify hardware into three broad categories: SIMD with sequential stream accesses only, SIMD with a conditional stream access mode [25, 24]; and MIMD capable hardware.

### *5.2.1 SIMD Sequential Mapping*

This is the most efficient hardware category, where the same instruction sequence is broadcast to all PEs and the local memory is tuned for sequential access. To utilize this type of restricted execution and address mode architecture requires that the irregular computation of our applications be regularized. This can be achieved using two general techniques.

The *connectivity sort* (SORT) method sorts the nodes based on their connectivity degree into lists of fixed degree. Computation can then proceed in a regular fashion handling one fixed-connectivity list at a time. This technique is conceptually very simple and potentially incurs no execution overhead, but has drawbacks. First, sorting is a computationally expensive process and leads to large preprocessing overheads. Second, sorting into individual fixed-connectivity lists may result in many short lists leading to increased inner loop startup overheads. A similar method forms the basis of the JAD sparse format [38] and CSR with permutation [14] used for sparse matrix-vector multiplication.

The *fixed padding* (PAD) technique regularizes the computation by ensuring all neighbor lists conform to a predetermined fixed length. Nodes whose neighbor list is shorter than the fixed length $L$ are padded with dummy neighbors, whereas nodes whose list is longer than $L$ are split into multiple length-$L$ lists. Splitting a node entails replicating its data, computing separately on each replica, and then reducing the results. The advantage of this scheme is its suitability for efficient hardware and low preprocessing overhead. The disadvantages include wasted computation on dummy neighbors and the extra work involved in replicating and reducing node data. The ELL [38] format for sparse matrices is equivalent to PAD, where each row of storage contains as many elements as the row with the most non-zeros in the matrix.

It is possible to combine SORT and PAD. For example, the connectivity list can be sorted into bins and then the list of each node padded up to the bin value. We do not evaluate hybrid techniques in this paper and refer the reader to an algorithmic modification of GROMACS that targets the SIMD Quadrics computer [36].

### *5.2.2 SIMD with Conditional Access*

A simple extension to a pure sequential access SIMD stream architecture is the addition of a conditional access mechanism. The basis of this mechanism is to allow the PEs to access their local memories in unison, but not necessarily advance in the stream, i.e. the next access to the stream may read the same values as the previous access independently in each PE. Such a mechanism can be implemented as discussed in [25] and in [42] for vector processors. In our evaluation we use an alternative implementation based on indexable local memory.

With conditional access (COND), the node computation of Figure 1 can be modified to support SIMD execution as shown in Fig-

ure 6. The key is to fold the processing of the node into the inner loop that processes neighbors, but execute it conditionally, as suggested in [4]. The overhead of conditionally executing the node portion of the computation for every neighbor is typically low, as reported in Table 1 and analyzed in Section 6.2.

```
1  i = 0;
2  neighbors_left = 0;
3  for (j=0; j<num_neighbors; j++) {
4    if (neighbors_left == 0) {
5      process_node(nodes[i]);
6      neighbors_left = neighbor_starts[i+1] -
7                       neighbor_starts[i];
8      i++;
9    }
10   process_neighbor(neighbors[neighbor_list[j]]);
11   neighbors_left--;
12 }
```

**Figure 6: Pseudo-code for COND parallelization method**

An additional overhead is related to load balancing. Each PE is responsible for processing a subset of the nodes in each computational strip, but the amount of work varies with the number of neighbors. This leads to imbalance, where all PEs must idle waiting for the PE with the largest number of neighbors to process in the each strip completes. Load balancing techniques have been studied in the past, and the solutions and research presented are applicable to stream processors as well.

### *5.2.3 MIMD Mapping*

With MIMD style execution, the basic node computation of Figure 1 can be implemented directly in parallel, and synchronization between PEs need only occur on strip or phase boundaries. This results in an execution that is very similar to SIMD with conditional access without the overhead of performing node-related instructions in the inner loop.

Similarly to the COND method, the load must be balanced between PEs to ensure optimal performance. The granularity of imbalance, however, is at the level of processing the entire dataset, as opposed to the strip level as in COND.

To support true MIMD execution, the hardware must provide multiple instruction sequencers. These additional structures reduce both the area and energy efficiency of the design. A quantitative analysis of this area/feature tradeoff is left for future work, but we bound the maximal advantage of MIMD over SIMD in Subsection 6.2.

## 6. EVALUATION

In this section we evaluate the performance characteristics of the mapping methods on the cycle-accurate Merrimac simulator for the applications of Section 3. We evaluate two configurations of Merrimac. The default configuration (bold values in Table 2) that has a stream cache and an alternate configuration with an increased SRF size and no stream cache.

We implemented a general software library that processes the connectivity list in any combination of the localization and parallelization methods. Each program provides an input and output module for converting the specific dataset format to and from a general neighbor list format.

For each of the four applications we implemented the SORT, PAD, and COND parallelization techniques (our FEM code has a fixed connectivity degree, and all methods are equivalent). We did not implement the MIMD variant directly, but bound its peak performance advantage at only 12% over the best performing SIMD technique. We estimate MIMD computation time by using the optimized pipeline execution schedule of SORT, which has a minimal inner loop, and eliminate the overheads associated with using multiple sorted connectivity lists. We combine each parallelization scheme with all three localization methods: renaming without removing duplicates (nDR); renaming with duplicate removal where the indexed accesses are in-lane within each PE (nDR_IL); and duplicate removal with dynamic accesses across SRF lanes (DR_XL).

| Parameter | |
|---|---|
| Operating frequency (GHz) | **1** |
| Peak 64-bit FP ops per cycle | **128** |
| SRF bandwidth (GB/s) | **512** |
| SRF size (MB) | [**1**/2] |
| Number of PEs | **16** |
| PE to PE communication | **1** word/cycle per PE |
| Stream cache size (KB) | [512/**0**] |
| Peak DRAM bandwidth (GB/s) | **64** |

**Table 2: Machine parameters**

## 6.1 Performance Summary

Table 3 summarizes the overall sustained performance of the best mapping of each application on the cycle accurate single-chip Merrimac simulator including a model of Rambus XDR DRAM with a peak bandwidth of 64GB/s. Utilizing the mapping techniques to take advantage of both parallelism and locality at the same time, Merrimac can achieve very high performance for challenging irregular applications. FEM and GROMACS, which have significant arithmetic intensity, achieve very high sustained performance of 53% and 36% of Merrimac's peak 128GFLOP/s. CDP and SPAS, on the other hand, have low arithmetic intensity and are memory bound applications. Therefore, they only sustain 8.6GFLOP/s and 3.1GFLOP/s respectively. Note that the absolute peak performance of an AMD $90nm$ dual-core Opteron running at 2.4GHz/s is roughly the same as the measured sustained performance of CDP on Merrimac.

| Application | Variant | Sustained Performance [GFLOP/s] | Sustained BW [GB/s] |
|---|---|---|---|
| CDP | DR_XL_SORT | 8.6 | 34.3 |
| GROMACS | nDR_IL_COND | 45.9 | 51.4 |
| FEM | nDR_IL_ | 60.4 | 25.5 |
| SPAS | DR_XL_PAD | 3.1 | 37.0 |

**Table 3: Performance of best performing variant on Merrimac with the default parameters of Table 2.**

To understand the decisions made to reach these overall performance numbers, we look at compute properties of the mapping variants separately from memory system dependent performance to explain both the rational for parallelization and localization. We then draw conclusions both on how to design Stream Processors and how to tune irregular applications for these systems in Section 7.

## 6.2 Compute Evaluation

To isolate the computational properties of the variants from memory system performance, Figure 7 shows the number of *computation cycles* in each application normalized to the nDR_IL_COND variant. *Computation cycles* are equivalent to running the application with a perfect off-chip memory system. Remember, that in a

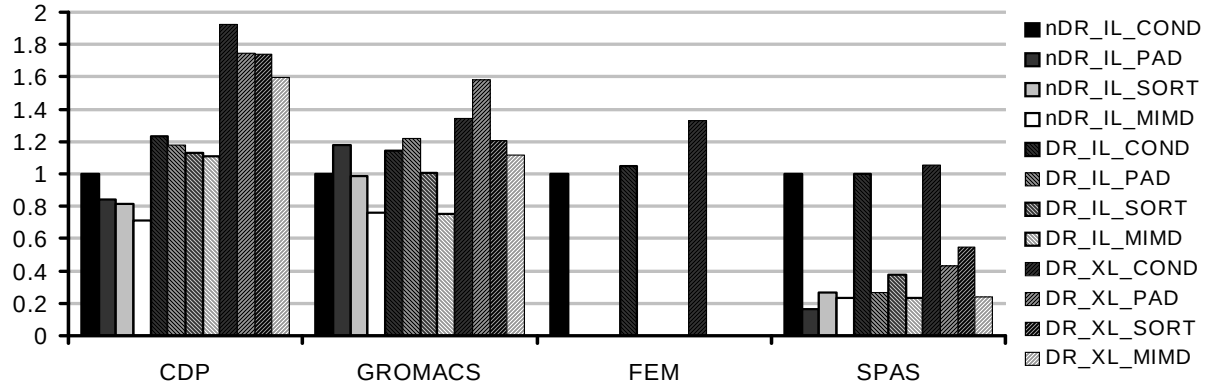



**Figure 7: Computation cycle simulation results for all mapping variants across the four applications. Execution cycles measured with perfect off-chip memory and normalized to the nDR_IL_COND variant of each application.**

stream processor the PEs cannot directly access off-chip memory and all aspects relating to SRF access and PE-to-PE communication are faithfully modeled.

### *6.2.1 Parallelization Compute Properties*

The left-most (solid) group of bars in each application shows the computation cycles of the nDR variants. CDP and SPAS have a small number of operations applied to each neighbor and the COND technique significantly degrades performance as the extra operations per neighbor are relatively numerous (see Table 1). Additionally, the conditional within the inner loop introduces a loop-carried dependency and restricts compiler optimizations. This results in the factor of $5$ computation cycle difference seen in SPAS. GROMACS has a large number of inner loop operations, and COND outperforms both SORT and PAD.

The SORT technique has the least amount of computation performed because there is no overhead associated with either padding or conditionals. However, the computation cycles are also affected by the kernel startup costs, which are greater for SORT as each connectivity degree requires at least one kernel invocation. This has an effect on the execution of SPAS, which has many connectivity bins with few nodes. GROMACS also has a large spread of connectivity degrees, leading to only a marginal improvement in computation cycles of SORT over COND.

Finally, the performance of a MIMD implementation is bounded by a 30% improvement over the best SIMD scheme in all our applications. A true MIMD implementation will not result in execution that is as efficient as this estimate because of load balancing, loop startup costs, and the effects of memory bandwidth and latency. FEM has a fixed connectivity degree, and therefore cannot benefit from MIMD execution. Similarly, CDP is memory bound and shows no potential for improving performance by reducing computational overheads. While GROMACS is not memory bound, the memory subsystem is active for over $88\%$ of the execution time. Hence, MIMD cannot improve performance by more than $12\%$. The only benchmark we studied that has computation that is not overlapped with communication is SPAS. With SPAS the performance potential of MIMD is hindered by the small number of operations within the inner loop of the computation. Because of this small number, the inner loop must be aggressively unrolled or software pipelined to achieve good utilization of the FPUs. These optimization techniques are equivalent to dynamically padding the computation to the degree to which the loop is expanded. Thus, with SPAS, the performance of a MIMD configuration should be similar to that of PAD. A more detailed evaluation of MIMD execution for Stream Processors is presented in [1].

### *6.2.2 Duplicate-Removal Compute Properties*

Renaming while removing duplicates relies on indexing into the SRF, which is less efficient than sequential access. In-lane indexing inserts index-computation operations, and cross-lane indexing has an additional runtime penalty due to dynamic conflicts of multiple requests to a single SRF lane.

In SPAS and CDP, the inner loop contains few instructions and the overhead of indexing significantly increases the number of computation cycles (factors of $1.4-2.6$x). In contrast to the short loops of CDP and SPAS, GROMACS and FEM have high arithmetic intensity. Thus, adding in-lane indexing increases the computation cycles by only $13\%$ for GROMACS and less than $5\%$ in FEM.

Another observation relates to the amount of locality captured by the DR_XL technique. In CDP, SPAS, and FEM, increasing the amount of available state to the entire SRF, increases the amount of locality exploited. Therefore, a large number of nodes may have neighbors in the same SRF lane and will dynamically conflict, increasing the overhead of cross-lane access. Using cross-lane indexing increases the computation cycles by over $70\%$ in CDP, rougly $30\%$ in GROMACS, more than $150\%$ in SPAS, and almost $50\%$ in the case of FEM.

Despite the higher computation cycle count, the next subsection shows that overall application runtime can be significantly improved by employing duplicate removal once realistic memory system performance is accounted for.

## 6.3 Storage Hierarchy Evaluation

Figure 8 shows the execution time of the applications with a simulated $64$GB/s peak Rambus XDR DRAM based memory system. We show the number for the best performing SIMD variant in each application with nDR, DR_IL, and DR_XL options normalized to the application runtime of nDR on a configuration with a stream cache. We also explore two storage hierarchy configurations for analyzing the tradeoffs of using a stream cache vs. increasing strip sizes: *cache* uses a 1MB SRF backed up by a $512$KB cache; and *nocache* does not use a cache and dedicates 2MB of on-chip memory to the SRF (the SRF size must be a power of 2).

In CDP there is an advantage to increasing the strip size in all mapping variants, even when duplicates are not removed in soft-

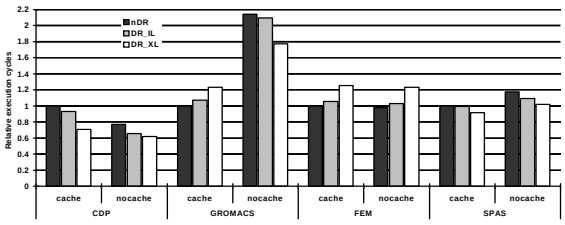



**Figure 8: Performance results for the best performing parallelization variant of each application with a realistic memory system. The *cache* configuration employs a stream cache, while *nocache* dedicates more on-chip memory to the SRF.**

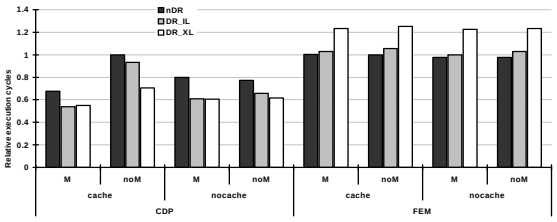



**Figure 9: Performance results for CDP and FEM showing the advantage of applying localization increasing reordering with METIS.**

ware. CDP has low arithmetic intensity (Table 1) and is limited by memory system bandwidth, therefore removing duplicate data loads significantly improves performance. Careful management of duplicates in software outperforms the reactive hardware cache. The more state software controls, the greater the performance advantage. With DR_IL, performance is improved by $35\%$ with a 2MB SRF (128KB per lane), but by only $7\%$ when the SRF is half the size. With cross-lane indexing, DR_XL has access to additional SRF space, and especially with the smaller SRF, this is a big advantage and performance is $30\%$ better than nDR. We also see, that with the large SRF, the additional state is less critical because DR_IL is already able to reduce bandwidth demands.

GROMACS is a compute-bound program in the cached configuration, but memory-bound without a cache. With a cache, nDR performs best because there is no overhead associated with indexing. DR_IL has a $7\%$ execution time increase and the higher computational time overhead of DR_XL results in a slowdown of $24\%$. With no cache, removing duplicates can improve performance. With the nDR technique, performance is degraded by a factor of $2.1$ (compared to cached configuration). DR_IL is unable to utilize much locality due to the large amount of state required for GROMACS (see Figure 5) DR_XL does exploit locality and improves performance by $16\%$ over DR_IL, however, this is still a $77\%$ slowdown compared to cached nDR.

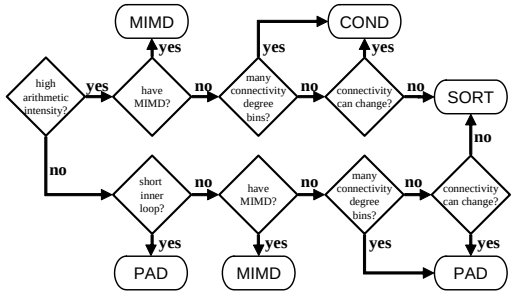
While there is a 17% performance advantage to using a cache for nDR variant of SPAS, the software duplicate-removal methods match the performance of a cached configuration without the additional hardware complexity of a reactive stream cache. FEM has little reuse and benefits minimally from duplicate removal in either hardware or software. The computation overhead of DR_XL decreases performance by $21 - 25\%$.

As shown in Subsection 5.1.2, reordering the nodes and neighbors in the connectivity list using domain decomposition techniques can increase the amount of reuse within a given strip size (CDP and FEM). Figure 9 explores the effects of reordering on performance, where bars labeled with *M* use METIS and those labeled with *noM* do not. We make three observations regarding the effects of using METIS on application performance and conclude that METIS is a generally improves application performance.

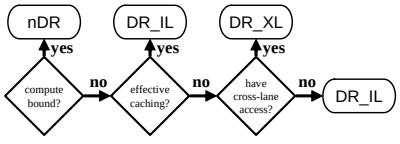
First, using METIS to increase locality significantly improves performance of the cached configurations in CDP, with the best performing cached configuration outperforming the best non-cached configuration by $23\%$. In FEM, the non-cached configurations consistently outperform the cached configuration due to the larger strip sizes leading to reduced kernel startup overheads. Second, regardless of the availability of a cache, METIS always improves the performance of the DR_IL scheme. This is because METIS exposes greater locality that can be exploited by the limited SRF state within each lane. In fact, with METIS enabled, DR_IL can outperform DR_XL by $(-1) - 10\%$, whereas without METIS DR_IL trails the performance of DR_XL by $5 - 33\%$. Finally, while employing METIS improves locality for duplicate removal, the reordering of nodes can reduce DRAM locality and adversely impact performance [3]. We can see this effect in the results for nDR without a cache where METIS degrades performance by about $4\%$ when duplicates are not removed. This is also the reason for the counter-intuitive result of METIS reducing the performance of uncached DR_XL in CDP.

# 7. CONCLUSIONS

In this paper we present a framework for characterizing irregular unstructured mesh and graph based applications in the specific context of stream architectures. Based on our observations and prior work relating to vector and distributed architectures, we develop and classify mapping techniques that address the critical aspects of localization and parallelization under the constraints of restricted control and the explicitly software managed storage hierarchy of stream processors.




(a) Parallelization




(b) Localization

**Figure 10: Decision trees for choosing mapping variant based on application characteristics. Note that MIMD is not strictly required in the decision, and its additional hardware requirements provide only up to $12\%$ performance benefits.**

We then analyze the properties, tradeoffs, and performance characteristics of our techniques on four representative applications from

the finite element, finite volume, n-body simulation, and sparse algebra domains. The evaluation shows that the complexity of cross-lane indexing has significant compute cycle overheads, but when combined with software duplicate removal and locality enhancing reordering can achieve speedups greater than a factor of two compared to a baseline architecture that stresses sequential access. Additionally, we introduce and evaluate a stream cache architecture, which can boost the performance of GROMACS by a factor of $4$, but by reducing the size of the on-chip software controlled SRF local memory hurts performance in the case of CDP and FEM. Finally we argue that restricting control to SIMD style execution, which allows for efficient hardware structures, can come close to an architecture with fully flexible control. We bound the maximal advantage of MIMD style SPMD execution to roughly $30\%$ with optimistically low MIMD overhead assumptions and a perfect memory system and a realistic performance advantage of less than $12\%$.

We summarize our conclusions in Figure 10, which depicts decision trees for choosing the most appropriate parallelization and localization mapping based on application characteristics and Stream Processor capabilities. Note that MIMD requires significant additional hardware over the SIMD parallelization techniques and can provide at most $< 12\%$ performance gain for the style of applications evaluated in this paper. A more detailed evaluation of the benefits and cost of adding MIMD support to a Stream Processor appears in [1].

## 8. REFERENCES


[1] J. Ahn, W. J. Dally, and M. Erez. Tradeoff between Data-, Instruction-, and Thread-level Parallelism in Stream Processors. In *Proceedings of the* $21^{st}$ *ACM International Conference on Supercomputing (ICS'07)*, June 2007.

[2] J. Ahn, M. Erez, and W. J. Dally. Scatter-add in data parallel architectures. In *Proceedings of the Symposium on High Performance Computer Architecture*, Feb. 2005.

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

[4] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 177–189, 1983.

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

[6] G. E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, Massachusetts, USA, 1990.

[7] R. Buyya, D. Abramson, and J. Giddy. A Case for Economy Grid Architecture for Service-Oriented Grid Computing. In *10th IEEE International Heterogeneous Computing Workshop*, 2001.

[8] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. University of California-Berkeley Technical Report: CCS-TR-99-157, 1999.

[9] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, and S. McKee. Impulse: Memory system support for scientific applications. *Journal of Scientific Programming*, 7:195–209, 1999.

[10] ClearSpeed. CSX600 Datasheet. http://www.clearspeed.com/downloads/CSX600Processor.pdf, 2005.

[11] L. Codrescu, D. Wills, and J. Meindl. Architecture of the Atlas chip-multiprocessor: Dynamically parallelizing irregular applications. *IEEE Transactions on Computer*, January 2001.

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

[13] W. J. Dally and W. Poulton. *Digital Systems Engineering*. Cambridge University Press, 1998.

[14] E. F. D'Azevedo, M. R. Fahey, and R. T. Mills. Vectorized sparse matrix multiply for compressed row storage format. In *proceedings of the 2005 International Conference on Computational Science (ICCS'05)*, pages 99–106, May 2005.

[15] S. J. Deitz, B. L. Chamberlain, S.-E. Choi, and L. Snyder. The design and implementation of a parallel array operator for the arbitrary remapping of data. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 155–166, 2003.

[16] S. J. Deitz, B. L. Chamberlain, and L. Snyder. Abstractions for dynamic data distribution. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 42–51. IEEE Computer Society, 2004.

[17] G. C. Fox. What have we learnt from using real parallel machines to solve real problems? In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 897–955, 1988.

[18] N. Goharian, T. El-Ghazawi, D. Grossman, and A. Chowdhury. On the enhancements of a sparse matrix information retrieval approach. *PDPTA*, 2000.

[19] M. Guo. Automatic parallelization and optimization for irregular scientic applications. In *18th International Parallel and Distributed Processing Symposium*, 2005.

[20] H. Han, G. Rivera, and C. Tseng. Software support for improving locality in scientic codes. In *Compilers for Parallel Computation*, 2000.

[21] J. Hippold and G. Runger. Task pool teams for implementing irregular algorithms on clusters of SMPs. In *Parallel and Distributed Processing Symposium*, 2003.

[22] H. P. Hofstee. Power efficient processor architecture and the cell processor. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, Feb 2005.

[23] Y. S. Hwang, B. Moon, S. D. Sharma, R. Ponnusamy, R. Das, and J. H. Saltz. Runtime and language support for compiling adaptive irregular programs on distributed-memory machines. *Softw. Pract. Exper.*, 25(6):597–621, 1995.

[24] N. Jayasena, M. Erez, J. 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.

[25] 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.

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

[27] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[28] K. Kitagawa, S. Tagaya, Y. Hagihara, and Y. Kanoh. A hardware overview of SX-6 and SX-7 supercomputer. *NEC Research and Development*, 44(1):27, January 2003.

[29] C. Koelbel and P. Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):440–451, 1991.

[30] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 318–328, 1988.

[31] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

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

[33] 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, 1976.

[34] G. Papadopoulos and D. Culler. Monsoon: an explicit token-store architecture. *ACM SIGARCH Computer Architecture News*, 18(3):82–91, 1990.

[35] D. I. Pullin and D. J. Hill. Computational methods for shock-driven turbulence and les of the richtmyer-meshkov instability. *USNCCM*, 2003.

[36] D. Roccatano, R. Bizzarri, G. Chillemi, N. Sanna, and A. D. Nola. Development of a parallel molecular dynamics code on SIMD computers: Algorithm for use of pair list criterion. *Journal of Computational Chemistry*, 19(7):685–694, 1998.

[37] R. M. Russell. The Cray-1 computer system. *Communications of the ACM*, 21(1):63–72, 1978.

[38] Y. Saad. *Iterative Methods for Sparse Linear Systems*. 2nd edition, January 2000.

[39] J. H. Salz, R. Mirchandaney, and K. Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Trans. Comput.*, 40(5):603–612, 1991.

[40] P. Sanders. Efficient emulation of MIMD behavior on SIMD machines. Technical Report iratr-1995-29, 1995.

[41] W. Shu and M.-Y. Wu. Asynchronous problems on SIMD parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 06(7):704–713, 1995.

[42] J. E. Smith, G. Faanes, and R. Sugumar. Vector instruction set support for conditional operations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 260–269, June 2000.

[43] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, 1995.

[44] J. Su and K. Yelick. Automatic support for irregular computations in a high-level language. In *19th International Parallel and Distributed Processing Symposium*, 2005.

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

[46] 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.

[47] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, 2006.

[48] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Stanford, California, 1998.