International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 9, Number 1, pages 70–96, January 2019

A Shared Memory Parallel Block Streaming Model for Irregular Applications

Anup Zope
Center for Advanced Vehicular Systems
Mississippi State University
Starkville, USA

and

Edward Luke
Department of Computer Science and Engineering,
Mississippi State University
Starkville, USA

Received: June 22, 2018
Revised: October 18, 2018
Accepted: December 4, 2018
Communicated by Susumu Matsumae

Abstract

Due to worsening machine balance, a lightweight irregular application can utilize only a small fraction of the peak computational capacity on modern processors. Performance of such an application is also unpredictable due to the scattered data accesses. Even though architectural features such as a cache system, hardware prefetchers etc., that reduce the cost of irregular access, are commonly found in most of the modern processors, their design parameters differ widely from one processor to another. Therefore, a performance improving programming technique still needs extensive tuning to gain maximum benefit on a target processor. In this scenario, achieving portable performance becomes difficult. This work proposes a block streaming machine model and hypothesizes that an algorithm based on the model has predictable execution time. To enable adaptation of this model for irregular applications, we also provide algorithmic transformations that can be used to replace the scattered accesses with streaming accesses in a cost predictable way. Further, we experimentally demonstrate usefulness of the model and the transformations for static lightweight irregular computations such as those performed by a numerical partial differential equation solver on modern multicore processors.

Keywords: bridging model, block streaming model, look back streaming access, performance predictability, performance portability, irregular applications, static and dynamic schedule, Intel®Xeon®Phi ™

1 Introduction

The need to achieve higher floating point throughput with lower power consumption has compelled processor designers to develop multi-core and many-core processors equipped with SIMD vector units for data level parallelism (DLP), instruction pipelining and superscalar out-of-order execution engine for instruction level parallelism (ILP) [21]. They are also equipped with resources designed to

improve memory access time such as a large and sophisticated hierarchical cache system, hardware and software prefetchers, wide loads and stores, streaming stores, large line fill and load/store buffers. Simultaneous multithreading allows fast thread context switching so that when a thread blocks for data, other ready-to-run thread can be quickly scheduled for execution. This approach is known as latency hiding.

Developing a software that can effectively utilize these architectural resources is notoriously difficult because of coupled interaction between them as well as their often undocumented specifications. For example, simultaneous multithreading produces cache contention if the cores are oversubscribed. As a result, even a seemingly simple programming technique may require extensive tuning (or autotuning) to gain maximum benefit on a given platform [19, 20]. Also, the specifications of these architectural resources differ from one processor to another. Therefore, the same programming techniques may not work on another platform or may require re-tuning/autotuning. As a result, on modern processors, it difficult to know in advance the amount of performance gain that can be obtained from the software redesigning and tuning efforts.

Lightweight irregular applications are particularly challenging to optimize. They have low computational intensity components that perform scattered data accesses. Due to low operational intensity, they require high rate of data access but due to the scattered access, they often have low cache hit rate. As a result, their performance is often dominated by the large DRAM latency of the data access. Also, the latency is unpredictable for a modern hierarchical memory system due to its transparent cache line replacement policy. Therefore, attempts to predict the cost of scattered accesses are often futile even though various memory models (e.g. [3]) are available. Hence, it is hard to know a priori the amount of performance gain that can be obtained from a transformation that improves locality or trades-off computations for data accesses and vice versa. The unpredictability is detrimental to the performance of a large irregular application which often relies on a sophisticated scheduler to obtain optimized computation schedule (e.g. [33, 34]). Performance predictability is essential for the operation of such a scheduler.

Streaming access is more predictable even when the cache system is present and has lower cost due to hardware prefetchers. Hence, they are more desirable for achieving peak performance of a lightweight computation. Unfortunately, the prefetch mechanisms on modern processors are delicate. It is easy to create situations that render them ineffective either due to failing to train the prefetchers or due to accidentally triggering too aggressive prefetching. Also, the cache associativity can cause conflicts that can evict the prefetched data. As a result, a computation structure intended for achieving peak bandwidth may fail to do so in reality and result in degraded performance.

In this paper [1], we present a block streaming model that abstracts a machine that guarantees predictable and consistent performance of streaming access as long as it satisfies the constraints imposed by the model. We think that these constraints are not too harsh to be useful to software designers as well as not too difficult for hardware designers to incorporate into existing processor architectures. The model establishes a bridge of expectations between software developers and hardware designers that ensures predictable performance of conforming software running on a conforming hardware. We also present transformations that can be applied to an irregular computation to express it using the streaming model. Further, we demonstrate that the transformations indeed produce implementations that are cost predictable on a modern multi-core processor and also achieve better performance than an optimized scattered access implementation.

2 The Machine Model

This section describes a machine model that executes a block streaming computation in a shared memory parallel execution environment. Schematic of the machine is as shown in Figure 1.

1. The processor has P identical cores each having a local memory of W words. Cost of accessing data from this memory is low and uniform. The cores explicitly manage placement of data in this memory.

[1] This paper extends the work presented in [53].

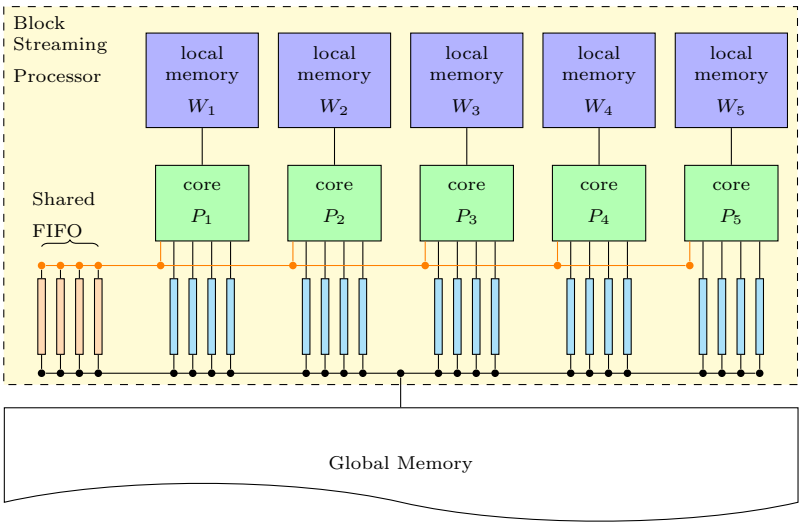


Figure 1: Schematic of the machine model.

2. The cores access data in the global memory through a set of configurable FIFO channels. The channels provide strictly sequential access to the data in the global memory. No two channels can access the same region in the global memory simultaneously. Also, a channel can be configured for either reading or writing data to the global memory, but not both at the same time. Configuration of a channel involves activation to start the data transfer and deactivation to stop the data transfer. Configuration overhead is o. The data access requests from all the channels to the global memory are served with equal priority.

3. Each core has N exclusive channels. All the cores have access to N_s shared channels that provide read only access to the data in the global memory. Data access from a shared channel by the cores is synchronized. Therefore, each access to a shared channel incurs an overhead of s.

4. Each exclusive channel configured for reading provides look back access to the data that was accessed certain spatial distance before the current streaming position. For this purpose, it uses a look back buffer on the core that is shared among the exclusive channels on the core. This buffer provides a low and uniform cost of the look back access up to a distance of L words.

5. Per core bandwidth provided by the exclusive channels is B_c, irrespective of the number of active channels on it. Bandwidth for the entire processor is B. If there are P_e cores active at a time, the effective bandwidth is $B_e = \min(B, P_e B_c)$.

6. Computations performed by the processor consists of a series of **kernels**. All or a subset of the cores participate in the execution of a kernel and operate on it in three phases.

   (a) Initialization: The core configures and activates required number of exclusive and/or shared channels.

   (b) Computation: The core streams in the data provided by the exclusive read channels, performs computations on it and streams out the results of the computation to the global memory through the exclusive write channels. The local memory is used by the core to store intermediate results of computations. In this phase, the core can also configure a new exclusive channel and reconfigure an already active exclusive channel.

   (c) Finalization: The core deactivates all the active channels.

7. All the cores synchronize on deactivation of all the channels on the processor.

When a core is executing a kernel, at least one channel (shared or exclusive) is active. Therefore, the only situation in which all the channels on the processor are in the deactivated state is when all the participating cores finish execution of the kernel, that is, after the finalization phase of each core. At this time, the cores synchronize. This is similar to the barrier synchronization but without explicit communication between the cores.

This model does not provide any means for the cores to communicate with each other. Even the synchronization is performed without explicit communication. In addition to this, the channels do not support concurrent reads and writes from the same region in the global memory. Therefore, the cores that participate in the execution of a kernel must work on different parts of the kernel that are data independent. A kernel is assumed to have multiple such parts. They are called as **segments**. A segment of a kernel is defined as a data independent, indivisible unit of work which is executed entirely on one core.

When the distribution of the segments in a kernel to the cores is predetermined (static schedule), each core knows which regions in the global memory to access. But when it is dynamic, they might need to configure the exclusive channels on the fly in the computation phase. The shared channels are provided to supply the configuration data (such as the start and end of the data for a chunk of segments in the global memory) to the cores during the computation phase. This can be used to implement dynamic scheduling strategies such as work stealing [11].

At a coarser level, this model is similar to the BSP model [48] except that the cores do not communicate with each other. The time slab between the start and end of execution of a kernel is similar to a superstep of the BSP model without the communication.

2.1 The Cost Model

An important objective of the cost model is to identify whether the data access cost, compute cost or hardware capacity is limiting factor for execution of a kernel so that suitable algorithmic transformations can be chosen to mitigate the bottleneck. This model assumes that a kernel has **abundant data parallelism**, that is, it has a large number of computational segments that are data independent. A segment of a kernel is an indivisible unit of work which is executed entirely on one core. It assumes that the kernel is **homogeneous**, that is, each segment has roughly the same computational intensity so that any schedule of the kernel that uniformly distributes the workload among the participating cores results in uniform performance characteristics on all the cores. It also assumes that the kernel is **calculable**, that is, a representative value of computational intensity of a segment of the kernel can be determined a priori.

For a kernel, let D be the **words per segment**, that is, the number of words consumed from and produced to the exclusive channels per segment of the kernel and C be the **time per segment**. Then, $\frac{D}{C}$ is computational bandwidth per segment and $\frac{C}{D}$ is computational intensity per segment. Let P_e be the number of participating cores. Since the kernel execution is homogeneous on all the cores, **computational bandwidth** and **computational intensity** of the kernel for the P_e cores is $\frac{P_eD}{C}$ and $\frac{C}{P_eD}$ respectively. Effective bandwidth of the P_e cores is $B_e = \min(B, P_eB_c)$. Therefore, if $\frac{P_eD}{C} < B_e$, the kernel is **compute bounded**, otherwise it is **bandwidth bounded**. If total volume of the read and write data for the kernel is V, its **execution time** is given by,

$$t = \max\left(\frac{C}{P_eD}, \frac{1}{B_e}\right) \times V + O(o) \tag{1}$$

The overhead, o, is due to the activation and deactivation of the exclusive channels. For a static schedule of execution, distribution of the segments to the cores is known a priori. If data of all the segments executed on a core is arranged sequentially in the global memory, it is possible to restrict the activation and deactivation to the initialization and finalization phases. Therefore, for a static schedule, it is possible to limit the overhead to a small constant factor.

For a dynamic schedule (e.g. work-stealing [11]), the shared channels can be used to perform dynamic segment distribution to the cores. This might introduce a large contribution from the overhead (o) in the total cost of the kernel due to frequent activation and deactivation of the exclusive channels. However, if the kernel is written with the assumption that there are more than one virtual

cores per physical core, the activation and deactivation of the FIFO channels can be scheduled such that the overhead overlaps with the computations, provided each newly activated channel reads or writes data larger than a certain threshold, called *critical stream length*. It is determined by the latency of filling or emptying the look back buffer of a channel. The only restriction on the virtual cores is that the sum of the number of their exclusive channels must be equal or less than the number of exclusive channels per physical core. In this case, the total overhead is limited to a small constant factor. This is similar to the latency hiding in the BSP model [48] using parallel slackness.

3 The Programming Model

This section specifies a programming model that provides application developers a language to express computations without worrying about how they map to the machine described in Section 2.

3.1 Preliminaries

The concept of a stream and a streaming operator are central to the programming model. But, before describing the model, some preliminaries are in order. A **value type** provides an operational semantics to an ordered collection of a fixed number of words. The collection of words is called as an **instance** or a **value** of the value type. Any two values of a value type have the same number of words. A value type may be a composite, that is, it may be an aggregation of other value types. A **list** is a composite value type formed by aggregation of a fixed number of ordered instances of a value type. If the number of instances in a list is variable, it is called as a **cluster**. The number of values in a cluster is called its **cluster size**. An **array** is an ordered collection of a large but finite number of instances of a value type stored consecutively in the global memory.

Various operations can be performed on the values. A **join** is an associative and referentially transparent operation that produces a single value from a set of values of the same type. Examples are sum, multiplication etc. A **concatenation** is a referentially transparent operation that produces a composite value using a set of values of the same or heterogeneous types. A **mapped join** on a set of clusters is a join performed on selected values from the clusters. The selection is specified using a **map**.

3.2 Streams and a Streaming Operator

A **stream** provides strictly sequential access semantics for the values stored in an array. The array is called as the **target array** of the stream. A stream maintains a pointer to current access position in the target array which can be moved in forward direction only. A stream can be either a **read stream** or a **write stream**. A read stream has a *get(d)* operation that reads a value from the target array at a look back distance of d from the current position. It has an *advance(d)* operation that advances the current position in the target array by d values. It also has a *pop()* operation that returns value at the current position and advances it to the next position. A write stream provides a *push(v)* operation that writes a value v to the target array at current position and advances it to the next location in the array. A series of push operations on a write stream results into sequential word writes to the target array. The *pop()*, *advance(d)* and *push(v)* operations cannot be undone. Once a stream reaches the end of the target array, they act as no-op.

A **streaming operator** is a computational unit that has following three phases of execution.

1. *Initialization*: In this phase, the operator creates read and/or write streams required for the computation phase.

2. *Computation*: In this phase, the operator iteratively executes a series of data independent tasks. Each iteration executes one or more tasks. Each task advances the read streams by certain distance, gets values from them, performs computations and pushes the result to the write streams.

3. *Finalization*: In this phase, the operator destroys the streams that were created in the initialization phase.

A streaming operator needs a **control stream** containing a series of control words in the order in which the tasks are executed by the operator. A control word determines which read streams and write streams are used for a task in the sequence. If an operator uses all the read and write streams for each task, then the control stream is implicit.

The collection of values advanced/pushed to a stream in a task of an operator is called as a **block** since they form a contiguous chunk of words in the global memory. The number of values in a block is called its **block size**. A streaming operator also needs a **block stream** that specifies block sizes of the values advanced/pushed to each stream in each task of the operator. If the block size of each stream is fixed for each task, the block stream is implicit.

A streaming operator also needs a **grain stream** that specifies how many tasks are executed by an iteration of the operator. Granularity of an operator is finest when each iteration executes only one task. This is the upper limit on granularity. By increasing the number of tasks in an iteration, granularity can be reduced. This is called as coarsening of the operator. Coarsening or refinement of an operator is controlled using the grain stream. If the number of tasks is fixed for each iteration, then the grain stream is implicit.

Schematic of a streaming operator is shown in Figure 2.

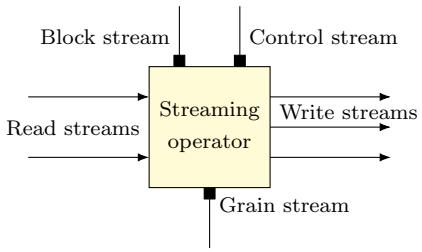


Figure 2: Schematic of a streaming operator.

This computational model is software abstraction of the machine model presented in Section 2. An array is software abstraction of a contiguous region of the global memory. A stream is software abstraction of an exclusive channel. Once a stream is created, it configures and activates one of the free exclusive channels. Once it is destroyed, it deactivates the channel. A streaming operator is software abstraction of the computation performed by a core. Since the tasks of an operator are data independent, they can be scheduled for concurrent execution on the cores.

This computational model suggests a loose definition of streaming which performs block-by-block streaming data access but allows a limited form of scattered data access that is bounded by extents of the blocks used in a task as well as the look back distance. In this text, this type of streaming is termed as **block streaming** to distinguish it from common use of the term "streaming" that refers to strictly sequential or fixed strided data access [28, 36].

3.3 Fundamental Operator Types

Zipper operator A task of a zipper operator pops one value/list/cluster from each read stream, performs computations on them and pushes one value/list/cluster to each write stream. A zipper operator suggests one-to-one correspondence between the values/lists/clusters from each stream. The control stream is implicit for this operator. If the block stream is explicit, that is, if at least one task pops/pushes a cluster to a stream, it is called as a *clustered zipper operator*. Otherwise, it is called as a *simple zipper operator*. The grain stream may be either implicit or explicit.

Distribution operator A task of a distribution operator pops one value from each read stream, aggregates them into a single value and pushes it to a subset of the write streams. The control

stream is explicit and the block stream is implicit for this operator. The grain stream may be either implicit or explicit.

Mapped join operator A task of a mapped join operator pops one cluster from each read stream but pushes a cluster to only a subset of the write streams. Each value in a pushed cluster is computed using a mapped join on the values in the clusters read from the read streams. The block stream as well as the control stream is explicit for this operator. The grain stream may be either implicit or explicit.

If a mapped join operator or a distribution operator has output streams such that their target arrays are non-overlapping with their data arranged consecutively one after another in the global memory, then it is called as a *collated mapped join operator* or *a collated distribution operator* respectively.

Note: Distribution and mapped join operator are semantic inverses of one another. A distribution operator performs replication of input data while a mapped join operator performs contraction of input data. However, they are also similar to each other in that they both function similar to a multi-way quicksort or a distribution sort, that is, each of their task conditionally writes result of their computation to one or more write streams.

Next sections discuss application of the programming model to computations of practical importance.

4 Test Problem and Test Platform

Many computational algorithms require scattered access to the data stored in DRAM. To demonstrate usability of the block streaming model, we consider applications that numerically solve partial differential equations (PDE). These solvers apply discretized form of the equations they intend to solve over a discretized domain in order to obtain solution of desired quantities. We take an example of the gradient computation in a finite volume scheme. It is an important and time consuming component of these solvers. Due to low computational intensity, its execution time is dominated by the time required to perform scattered data access. Many other computations used by the solvers also exhibit similar characteristics. Also, the scattered data access pattern is determined by topology of the discretization which is known a priori and is fixed for the duration of the computations. Therefore, these applications are also *static* in addition to being *lightweight* and *irregular*.

The gradient computation is performed by applying discretized form of the Green-Gauss theorem to each element in the discretization [46]. Figure 3 shows a small computational domain discretized by triangles as a demonstration. In practice, the discretization can be performed by any convex polygonal shape in 2d and polyhedral shape in 3d. The triangles are called as cells and the edges are called as faces. Topology of the mesh establishes a face to cell map $M : F \to C$ and a cell to face map $M^{-1} : C \to F$. For a cell based finite volume scheme, goal of the gradient computation is to determine gradient at the cells using values of a scalar quantity specified at the faces. The gradient computation equation is as follows.

$$\nabla s_i = \frac{1}{V_i} \sum_{f \in M^{-1}(i)} s_f \times A_f \times \hat{n}_f \tag{2}$$

Here, s_f is average value of a scalar quantity at the face f. $\hat{n}_f$ and A_f are the normal and area of the face f. V_i and ∇s_i are the cell volume and average gradient at the cell i. Computation on each cell is data independent. In terms of the models discussed earlier, it is a segment (or task) of the kernel (or operator). It also satisfies the properties of abundant data parallelism, homogeneity and calculability. Calculability is due to the fact that each cell has a predictable compute cost and data footprint. The tests reported in this work were conducted on following platforms and problem sets.

Here, we demonstrate how this computation is traditionally realized. Let's assume that a mesh has N_c cells and N_f faces. Let's also assume that the cells are numbered as $c_i; i = 0, ..., N_c - 1$ and the faces are numbered as $f_j; j = 0, ..., N_f - 1$. Let `g` and `V` be arrays of size N_c that store the gradient (∇s) and volume (V) of the cells, respectively, in the given cell order. Similarly, let `s`,

`A`, and `n` be the arrays of size N_f that store the scalar value (s), area (A), and normal ($\hat{n}$) of the faces, respectively, in the given face order. Let `M` and `I` be the arrays of indices that store the cell to face connectivity such that for a cell c_i, the faces $f \in M^{-1}(i)$ related to it (according to the mesh topology) are given by indices stored in the range `I[i]` to `I[i+1]-1` of the array `M`. Therefore, the pseudocode for performing the gradient computation is given by listing 1. Even though the code performs streaming access to the arrays of cell values (e.g. `g` and `V`), the access to the arrays of face values (e.g. `s`, `A`, and `n`) is irregular due to the map M^{-1}. Note that, in an actual PDE solver, this computation is repeated several times with different values of the elements in `s`. The values in the arrays `V`, `A`, and `n` do not change since the mesh is non-deforming. Also, the arrays `M` and `I` do not change since the mesh topology does not change during the computation. Since the scattered data access pattern is dictated by the invariant arrays `M` and `I`, we termed this computation as static and irregular.

```
for i in 0 to N_c-1:
  gt = [0, 0, 0]
  for f in I[i] to I[i+1]-1:
    tmp = s[M[f]] * A[M[f]]
    gt[0] += tmp * n[M[f]][0]
    gt[1] += tmp * n[M[f]][1]
    gt[2] += tmp * n[M[f]][2]
  invv = 1.0/V[i]
  g[i][0] = gt[0] * invv
  g[i][1] = gt[1] * invv
  g[i][2] = gt[2] * invv
```

Listing 1: Pseudocode for the gradient computation.

The Intel®Xeon®E5-2680 v2 processor (codename: Ivy Bridge) It is a NUMA processor with two sockets and ten cores per socket. Hyper-threading is not available and all the hardware prefetchers were active. All the tests were conducted with ten threads pinned on a single socket and all memory allocated on the local DRAM of the socket. A three-dimensional grid of tetrahedral elements with 5436517 ($\approx$ 5.5 million) cells and 10966730 ($\approx$ 11 million) faces was used to conduct the tests on this platform. The gradient computation on this grid has memory footprint large enough to overflow the last level cache.

The Intel®Xeon®Phi ™5110P coprocessor (codename: Knights Corner) [26] It has 60 physical cores and four hardware threads per core. This gives applications 240 threads in total to perform the computation. The hardware scheduler for the threads uses round-robin style of scheduling. However, it cannot issue instructions to the same thread in consecutive cycles. Therefore, to fully utilize all the ALUs, there need to be at least two threads per core. Each core has 32 KB L1 data cache which is shared by all the threads on the core. Each core has 512 KB of unified L2 cache which is connected to the L2 cache of the other cores in a ring. Though the coprocessor can be used in offload mode, all the tests reported in this paper were conducted in native mode. There were two threads per core which gives 120 threads in total. A three-dimensional grid of tetrahedral elements with 1452758 ($\approx$ 1.5 million) cells and 2944379 ($\approx$ 3 million) faces was used to conduct the tests on this platform. The reduced size grid was necessary since the DRAM available on this platform is only 8 GB. However, this grid is characteristically similar to the grid used for the tests on the Intel®Xeon®E5-2680 v2 processor.

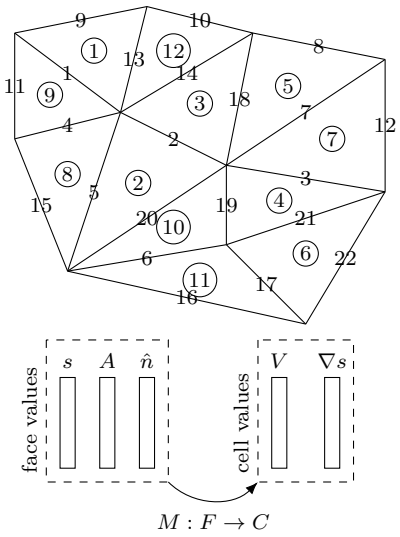


Figure 3: Example of a discretized domain for the gradient computation in a cell based solver.

5 Mapped Reduction

The gradient computation is an example of a **mapped reduction**. The cells and faces are given unique numbers. Their data is stored in arrays in the order of their numbering. The summation operation in Equation (2) represents reduction of the face values related to the cell i through M^{-1}. Therefore, this type of computation is termed as a mapped reduction. Its specification includes a map, a unit value and a join operation. The unit value is usually identity of the join operation. The join is an associative and referentially transparent operation that specifies how one or more values in the source array are combined with the value of an element in the destination array.

Modern processors provide read/write to any location in the global memory in any desired order. Therefore, implementation of the mapped reduction on these random access machines follows the mathematical description. There are two ways to perform the computation based on whether the map M or M^{-1} is used.

Pull reduction In this approach, iterations are performed on the elements in the destination array. In each iteration, value of the current destination array element is set to the unit value. The map M^{-1} is used to read values of related elements from the source array. Then they are combined with value of the current destination element using the join operation. $M^{-1} : C \to F$ is called as a pull map since it represents inflow of data from the source arrays.

Push replication In this approach, value of all the destination array elements is set to the unit value. Then iterations are performed on the elements in the source array. In each iteration, value of the current source element is used to update value of related elements in the destination array according to the join operation. The map $M : F \to C$ is called as a push map.

These variants are depicted in Figure 4. Effective utilization of a multi-core or many-core system requires attention to many details. On these processors, the read/write requests to the DRAM are served through a hierarchical cache system which is also responsible for transparently retaining frequently used data in a higher level so that it can be served to the cores with a low cost. For effective utilization of the cache system, numerical PDE solvers reorder the cells and faces such that the mapped accesses present higher temporal locality of data to the cache system. This rearrangement results in higher cache hit rate, hence, lower access time. Some examples of locality improving orders are reverse Cuthill-McKee (RCM), space filling curve (SF) etc. [14, 15, 38]. Different levels of the cache have different latency of the data access. Due to difficulty in determining the level of the cache

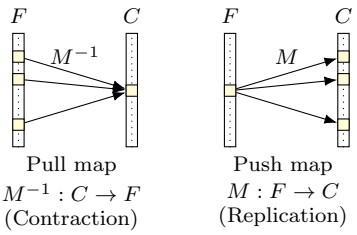


Figure 4: Pull reduction and push replication.

from which a particular data item will be fetched during execution of an irregular computation, it is hard to predict most suitable order for the scattered data accesses.

The shared DRAM and the private caches of the cores present another overhead to the scattered accesses. Since these processors need to provide concurrent read and write to a location in the shared global memory, it leads to race conditions that result in indeterminacy of the computational results. The cache system handles the concurrency by enforcing a cache coherency protocol (e.g. MESI [28]) that maintains a consistent state of the data across the private caches of these cores. Programs that need to enforce a particular order of the reads and writes across the cores need to tap into the cache coherency protocol by using mutex or atomic primitives (such as `std::atomic` of the C++ 11 standard [2], `atomic` pragma of the OpenMP standard [42]) or processor provided memory fence instructions (such as `mfence`, `sfence` and `lfence` of the Intel processors [28]). These operations are expensive. Therefore, programming techniques that reduce the overhead of these accesses while maintaining determinacy of computations are essential [8, 39]. Also, special care is required in the distribution of computations and data to the cores to avoid hidden performance penalty situations such as false sharing [47].

In addition to this, the memory system parameters such as the number of cache levels, cache size and associativity, width of the load and store operands, overhead of cache coherency varies from processor to processor. Also, the hardware and software prefetchers have different behavior on each processor, which may help or hinder the memory system operation for different types of computational loads and access patterns [31]. These factors make it hard to achieve portable performance across different architecture types or even across different generations of the same architecture.

The mapped reduction is much straightforward to schedule for multi-core execution. Since it has abundant data parallelism, a schedule that maps the computations to the cores such that each core gets a uniform and contiguous working set is sufficient. The OpenMP code for both the pull and push versions of a mapped reduction are shown in Listing 5a and 5b. To implement the pull mapped reduction, the destination array needs to be partitioned and distributed to the cores. This causes concurrent reads in the input arrays but makes the writes exclusive. Therefore, it is free from race condition. On the other hand, the push replication requires partitioning and distribution of the source arrays. This causes concurrent writes which require atomic primitives to avoid indeterminacy. Also, the push replication requires an extra pass over the destination array to set each element's value to identity of the reduction operation. Hence, the push replication is usually slower than the pull reduction.

Table 1 shows performance results of the multithreaded gradient computation on the test platform. Both the pull and push versions were tested with three different cell orders - the original order as specified in the grid file, the order obtained from reverse Cuthill-McKee [14, 15] and that obtained from Hilbert space filling curve [5, 22]. As expected, the pull reduction is much faster than the push reduction. Also, the locality improvement resulted in lower execution time.

```
template<typename T>
void sum_pull_reduction(
  T * dst, int const ndst,
  T const * src,
  int const nsrc,
  int const ** pull_map) {
    #pragma omp parallel for
    for(int i = 0; i < ndst; ++i) {
      dst[i] = 0.0;
      int const * mp = pull_map[i];
      while(mp < pull_map[i+1]) {
        dst[i] += src[*mp];
        ++mp;
      }
    }
}
```

(a) OpenMP code for pull reduction.

```
template<typename T>
void sum_push_replication(
  T * dst,
  int const ndst,
  T const * src,
  int const nsrc,
  int const ** push_map) {
    #pragma omp parallel for
    for(int i = 0; i < ndst; ++i)
      dst[i] = 0.0;

    #pragma omp parallel for
    for(int i = 0; i < nsrc; ++i) {
      int const * mp = push_map[i];
      while(mp < push_map[i+1]) {
        #pragma omp atomic
        dst[*mp] += src[i];
        ++mp;
      }
    }
}
```

(b) OpenMP code for push replication.

Figure 5: OpenMP code for pull reduction and and push replication.

Table 1: Time of scattered access gradient computation.

Cell Order	Pull Reduction (sec)	Push Replication (sec)
Original	0.08283049	0.145739641
Reverse Cuthill-McKee	0.03962708	0.078268944
Space filling	0.025988533	0.077740784

Now, let's consider the test grid used for the Intel®Xeon®E5-2680 v2 processor. Theoretically, each tetrahedral cell requires 192 bytes of data for the gradient computation. Since each cell computation is a segment in terms of the block streaming model, $D = 192$. The time required for a single cell computation was experimentally determined as $C = 3.83 \times 10^{-9}$ seconds. It is the time required when all the data required for the computation is available in the local memory (L1 cache in case of the test platform). For the gradient computation, it was obtained by averaging the repeated gradient computation time for a single tetrahedral cell. This gives $\frac{P_e D}{C} = \frac{10 \times 192}{3.83 \times 10^{-9}} = 501.3 \times 10^9$GB/s which is greater than 45 GB/s of peak bandwidth of the test platform (obtained from the STREAM benchmark [36]). Hence, according to the cost model (Section 2.1), the gradient computation is bandwidth bounded. Also, each cell requires 32 arithmetic operations. This gives $32 \times 5436517 = 173968544$ floating point operations for all the 5436517 cells in the mesh. The pull reduction version requires 786606312 bytes of data to store the required quantities associated with the faces and cells (including the cell to face map). The push replication version requires 961324400 bytes of data. Therefore, the flops/byte ratio is 0.22 and 0.18 for the two versions, respectively. This data was compared with the roofline plot of the test platform as shown in Figure 6. The roofline plot was obtained using the Empirical Roofline Toolkit (ERT) [1, 32, 51]. It shows that the performance of the gradient computation is bounded by data access cost for both the versions and all the three orderings. This finding matches with the prediction from the cost model. It also shows that there is still some potential for performance improvement if the cost of data access could be reduced to the cost of streaming access.

The block streaming model presented in this work is free from the concerns of indeterminacy, cache coherency and unpredictability of the data access. The FIFO channels provide a buffer similar

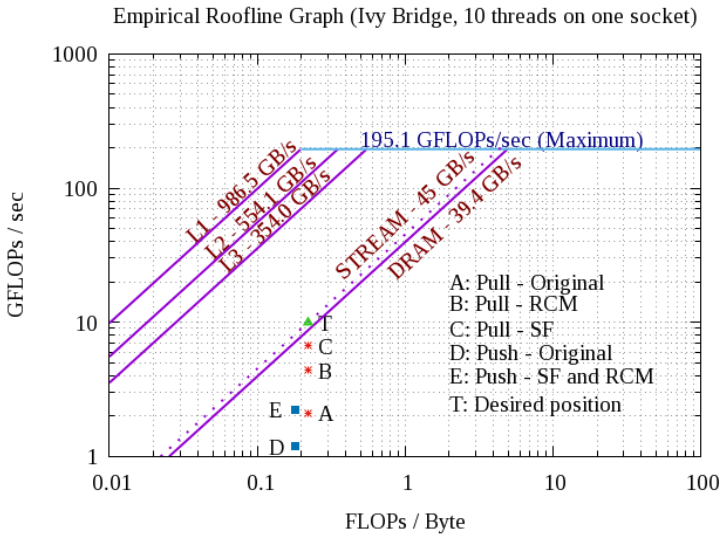


Figure 6: Performance of the gradient computation that uses scattered accesses with respect to the roofline plot of the Intel®Xeon®E5-2680 v2 processor. Point T indicates the desired performance of the gradient computation that uses the pull reduction.

to the cache system but the exclusive reads/writes from/to the global memory avoids indeterminacy as well as the need to have a coherent cache. Also the channels are designed to sequentially access contiguously stored data in the global memory. This makes the access cost uniform and predictable. Different machines can have different parameters for the bandwidth or the size of the look back buffer, but they are deterministic and invariant under any type of computational load. Since performance of a computation on this architecture depends only on these parameters, it is also predictable. While the restrictions on the data access appear severe and applicable to a narrow range of scenarios, it is not true. Next sections demonstrate how the model can be applied to irregular computations by using the mapped reduction as an example. Note that this model is not a replacement for the scattered access model. Its purpose is to provide a middle ground for algorithm designers and hardware designers so that predictable and portable performance of algorithms designed within this framework can be guaranteed if hardware vendors provide processors that conform to the model.

6 Distribution Schedule for the Mapped Reduction

Consider the push replication. It accesses the input values in sequential order but replicates them to the destination array by scattered writes. This is semantically equivalent to the operation of a distribution operator (Section 3.3). The extent of scattered write by the operator is limited due to the limited number of exclusive channels as well as the sequential writes enforced by them. To realize full replication of the source array values (as per the map), multiple stages of the distribution operators are required. These stages are represented by a tree as shown in Figure 7. The root of this tree forms the first stage, its children form the second stage and so on. Each stage produces data that is more localized than in the previous stage. After the final stage, the source data required by each destination element is in consecutive order in the staged array. At this point, a clustered zipper operator that performs the join operation on the consecutive values is applied to produce the values at the cells. Even though the amount of data grows with each successive stage, the staged array has bounded size due to finite arity of the destination to source map.

An execution schedule for the staged push replication requires determination of the control streams and extents of the input and output arrays for each distribution operator in the tree. The control streams can be determined using MSB radix partitioning over connections of the push map $M : F \rightarrow C$ [35]. A static irregular applications uses a given pattern of scattered accesses several

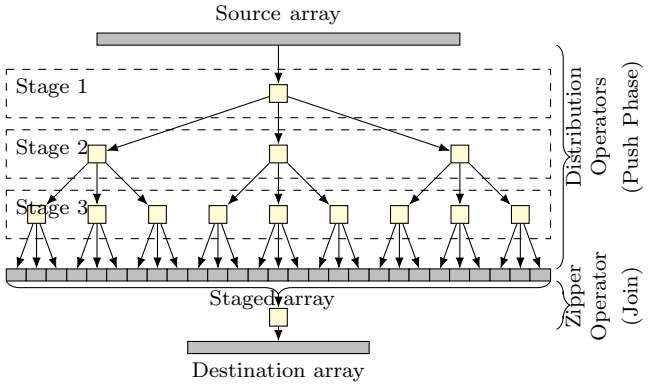


Figure 7: Mapped reduction using a tree of distribution operators.

times. Therefore, once the distribution schedule is determined it can be used several times, thus amortizing the cost of its generation.

Since the distribution stages perform only sequential data movement and no arithmetic operations, their performance is bandwidth bounded. Also the join operation is light weight. Hence, execution time of this transformation depends only on the data volume. If the destination array contains N values and if the number of write streams of each distribution operator is b, this algorithm requires $s = \left\lceil \frac{\lceil \log_2(N) \rceil}{\lfloor \log_2(b) \rfloor} \right\rceil$ stages. Each stage consumes $O(n)$ words and produces $O(n)$ words where n is the amount of data in the source array. Therefore, the amount of data transferred from and to the global memory is $V = O((s+1) \times 2n)$. The $+1$ in $(s+1)$ is due to the zipper operator. If B_e is the effective bandwidth of the participating cores, the time required for this computation is $t = O\left(\frac{2n(s+1)}{B_e}\right)$ (as per Equation (1)) on the machine described in Section 2. For the test problem described in Section 4 the exact amount of data accessed can be determined a priori once the distribution schedule is generated. Hence, it is possible to get accurate estimation of the execution time of this schedule. Also, due to abundant data parallelism, it is trivial to generate a parallel execution schedule for the distribution stages.

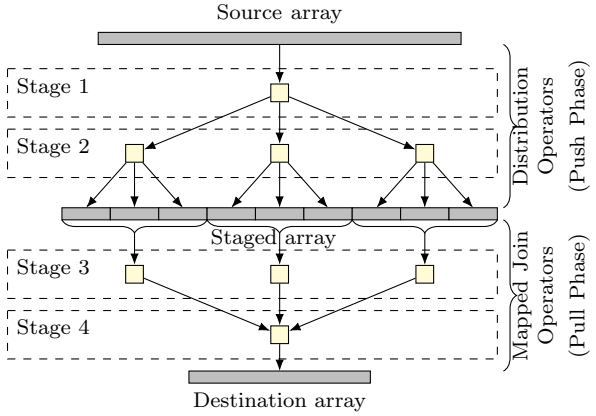


Figure 8: Mapped reduction using distribution and mapped join operators.

Improvement in the execution time of this algorithm can be obtained by using the local memory available on each core. In this case, we don't need all the $\left\lceil \frac{\lceil \log_2(N) \rceil}{\lfloor \log_2(b) \rfloor} \right\rceil$ stages. If the local memory is large enough to store k values, the distribution tree can be truncated as soon as all the output streams of a stage reach the size of $\leq k$. At this point, the join operation of the zipper operator can

be fused with the replication operation of the distribution operators to directly produce values in the destination array. Fusion of the zipper operator and the distribution operators in the last stage produces mapped join operators. Similarly, more distribution stages can be fused to produce more stages of the mapped join operators. Distribution phases represent expansion of data. Therefore, they are similar to the push replication. On the other hand, the mapped join operators perform contraction of data. Therefore, they are equivalent to the pull reduction. In this way, the fusion produces a hybrid variant of the push and pull reduction. This gives algorithm designers control over expansion of the data and the number of stages in a cost predictable way. The fusion transformation is depicted in Figure 8.

Though this transformation replaces low performance scattered access with high performance streaming access, it requires several passes over the source data. Therefore, if the scattered data accesses are rearranged for higher temporal locality, larger L2 and L3 caches (typically found on the modern processors) can easily reduce the effective data access latency. As a result, the relative benefit from streaming is reduced simply because the distribution passes need to traverse the entire data several times. Therefore, we found that unless the given source to destination map has poor locality and/or very high arity, the distribution schedule performs poorly compared to the scattered access version for the test problems considered in this study. Therefore, we suggest another schedule for the mapped reduction as explained in the next section.

7 Look Back Schedule for the Mapped Reduction

Cost of the transformation described in the previous section increases as the number of stages increases. In this section we suggest another transformation to perform the mapped reduction at a significantly lower cost. It uses the look back feature provided by the buffer of an exclusive read channel. The look back enables a read stream to obtain previously read data that is within the look back distance from the current position in the target array.

This variant of the mapped reduction is based on the pull reduction. But it requires a particular order of the face data called *first read order*, in addition to the locality improving order of cells. For a given cell order, let's assume that F_i is the set of faces accessed by the cells in the set $\{c_0, ..., c_i\}$. If $j < k$ for each pair of faces $(f_j, f_k) \in F_{i-1} \times (F_i \setminus F_{i-1})$, then the faces are in first read order. For a given cell order and cell to face map, the first read order of the faces can be obtained by greedy numbering of the faces as they are visited while enumerating the cells in the given order. The order is depicted in Figure 9. Further, if the cells are arranged using a locality improving order, then most of the face data can be found in the look back of the face data stream as computations are performed on the cells in their given order. The face data that is not available in the look back of the face data stream is supplied to the computations from a separate stream called *copy stream*. The cell computation can access the data in this stream also with a look back. As a result, it has a significantly smaller size than the size of the face stream.

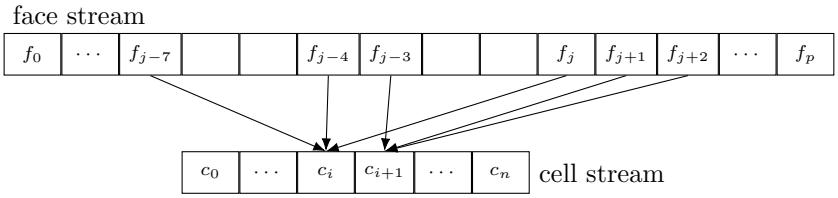
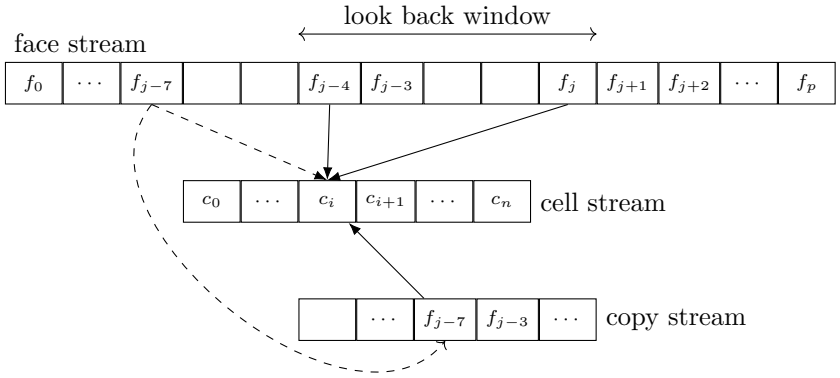
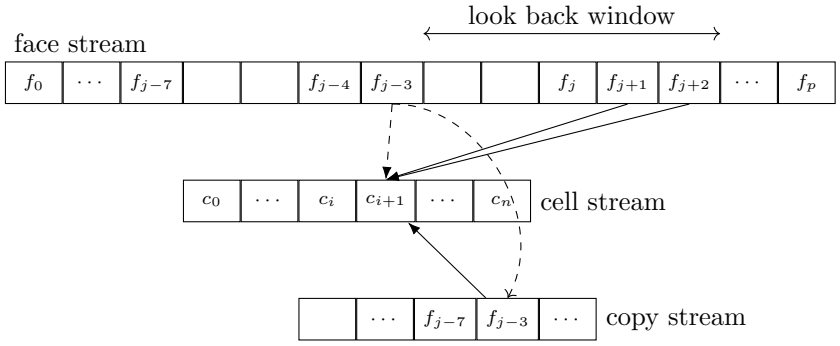


Figure 9: First read order of the faces with respect to the given order of cells.

Execution of this algorithm requires two phases - copy and compute. The copy phase generates the copy stream using distribution stages as described in Section 6. Due to small size of the copy stream, its generation requires a very small amount of time. The compute phase performs mapped reduction by performing only one pass over the data in the main stream and the copy stream. Hence, this strategy is more likely to produce speed up over the scattered access pull version. The operation of the compute phase is demonstrated in Figure 10.



(a) Computation of cell c_i. Note that the value f_{j-7} is not available in look back due to limited look back distance. It is supplied to the computation by the copy stream.



(b) Computation of cell c_{i+1}. Note that the value f_{j-3} is not available in look back since the look back window has advanced to f_{j+2}. It is supplied to the computation by the copy stream.

Figure 10: Demonstration of look back schedule for mapped reduction.

For a static application, the distribution schedule for copy phase and the look back schedule for the compute phase can be generated once and used several times later. Hence, the schedule generation cost is amortized. Also, there are many opportunities to reuse a copy stream in a large application. This will amortize the cost of generating the copy stream as well.

7.1 Multithreaded Look Back Schedule

Copy stream contains face data duplicated from the face stream. The faces whose values appear in the copy stream are termed as copy faces. The copy stream generation algorithm uses distribution schedule. If the copy faces are scattered throughout the face stream, the first stage of the distribution schedule needs to scan the entire face array but process only a small fraction of it. This increases the cost of copy stream generation significantly and wastes bandwidth. To alleviate this issue, the copy faces should be arranged consecutively in the face stream. Due to this rearrangement they cannot appear in first read order. Hence, their values are supplied to the computation exclusively by the copy stream. As a result, the face stream has two parts, one that has copy faces and another that has all the remaining faces arranged in first read order of the cell access.

Also, for multithreaded execution, the cell array is partitioned into equal sized chunks and each thread is assigned one chunk. As a result some faces may be accessed from more than one thread partitions. The look back schedule, therefore, segregates these thread shared faces and supplies them

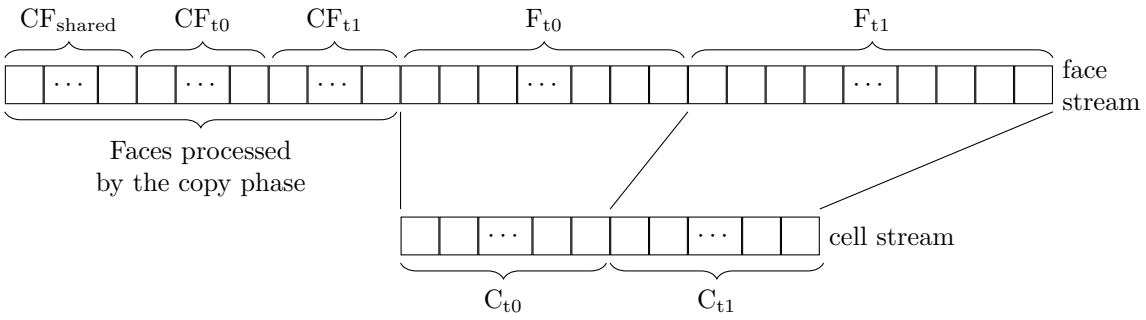


Figure 11: The face order for multithreaded execution of the look back schedule. This figure demonstrates the face order for two threads. C_{t0} are the cells processed by thread t0 and C_{t1} are the cells processed by thread t1. CF_{shared} are the copy faces accessed by both the threads. CF_{t0} and CF_{t1} are the copy faces exclusive to the respective threads. F_{t0} and F_{t1} are the faces accessed exclusively by threads t0 and t1 respectively. They are arranged in first read order with respect to the order of cells in C_{t0} and C_{t1}.

to the computation via the copy stream.

The face numbering, therefore, involves three sections. In the first section, all the thread shared copy faces appear, followed by the copy faces for each thread, followed by the remaining faces for each thread partition arranged in first read order according to the cell order in that partition. This is depicted in Figure 11. After this face reordering, the copy stream generation for each thread needs to scan only the thread shared portion and thread specific section of copy faces in the first stage of the distribution schedule. This results in significantly lower cost of generating the copy stream than the case where the copy faces are scattered throughout the face array.

8 Implementation Details

Modern processors do not strictly conform to the model presented in Section 2. However, features similar to the FIFO channels, look back buffer and local memory are implied in the memory system design. For example, these processors are equipped with hardware prefetchers that are similar to the FIFO channels and a hierarchical cache that functions similar to the local memory and look back buffer. But they do not guarantee data placement in the cache or prefetching rate. For example, the Intel®Xeon®E5-2680 v2 processor has four types of hardware prefetchers - the data cache unit and instruction stride based prefetchers that prefetch to the L1 cache, the spatial prefetcher and streamer for the L2 and L3 cache [27, 28]. These prefetchers are triggered by sequential or strided access in either forward or backward direction within a 4K page. There can be up to 32 such streams on a core and the streamer can run up to 20 cache lines ahead of current load request. This sounds like a very aggressive prefetching. However, the strict sequential/strided access pattern they expect has limited applications. Also, the amount of prefetching varies based on other activities in the memory system and these interactions are not well documented. In addition to this, it is easy create situations that reduce effectiveness of the prefetchers. Consider an example of LU decomposition of a small fixed size matrix. It has irregular access pattern but it is limited to the extents of the matrix size. When the computation is sequentially applied to such matrices stored in an array, effectiveness of the hardware prefetchers may reduce and result in load latency experienced by the computations. To handle this type of situation, the processors also provide software prefetch instructions. Their judicial use is essential since they consume processor resources. On the other hand, they operate on a single cache line. Hence, a given memory block size may require many prefetch instructions. The cache system also functions as a temporary storage for frequently used data which may get evicted if the prefetchers become too aggressive. Due to these factors, platform specific tuning is required for the block streaming access on modern processors.

The read streams used for the look back gradient computation required software prefetch instruc-

tions to gain maximum bandwidth. This required manual tuning for the prefetch distance, prefetch cache level and the number of prefetch instructions on both the test platforms. The processors also provide streaming store instructions that perform DRAM writes, bypassing the cache system. These are useful for data written using write FIFO channels. These stores free the cache for other uses. They require cache line aligned accesses and full cache line writes. The penalty of partial cache line writes is significant since it incurs the cost of a full bus transaction. However, the streaming store instructions on the Intel®Xeon®E5-2680 v2 processor write to only half of the cache line at a time. To enable full cache line writes, the processor maintains a write combining buffer which buffers partial cache line writes issued within a small time window. Once it is full, a bus transaction is issued. This is called as write combining. To use write combining on this platform, the write streams used a small buffer for the write data which is streamed to the DRAM once it is full. This is called as software write combining [27, 28]. Also, special care was required at the beginning and end of a write stream for the partial cache line writes. The masked streaming store instruction - `maskmovdqu` - was used for this purpose. For the Intel®Xeon®Phi processor, each streaming store write is of the size of one full cache line when the `vmovnrngoapd` instruction is used [25].

Each task of a distribution operator needs to know the write streams to which the concatenated value should be pushed. The control stream provides this information to the operator. In the implementation, each control word in the stream is represented by a bit mask, where the number of bits in it is equal to the number of write streams of the operator. The bit mask representation gives a vary compact representation since it does not require keeping track of the number of write streams used in each task. Integer data type is a natural fit for this. So, if a 64 bit integer is used to represent a control word, it can be used by a distribution operator that has up to 64 write streams.

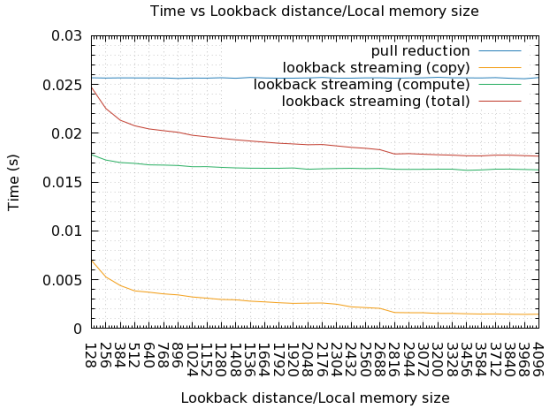
With these platform specific tuning and special code, fairly good bandwidth for both the distribution and look back schedule could be achieved. On a different hardware, it might require retuning. The need for performance tuning is an artifact resulting from the lack of hardware guarantee of the data access performance. If it is provided under the circumstances of the block streaming data access, there will be no need of tuning. Also, conforming applications will have predictable cost across all the conforming architectures. This will ensure performance portability as well.

9 Results and Discussion

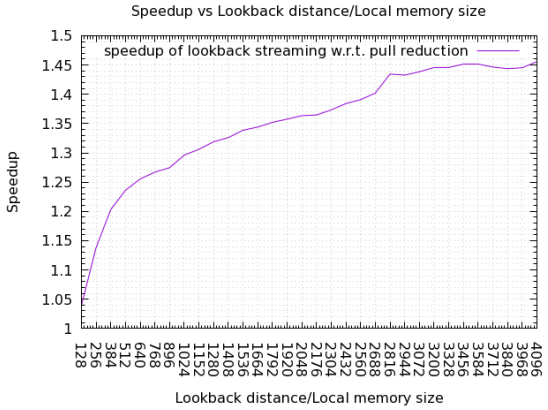
This section presents performance results of the gradient computation that uses the look back schedule on the two test platforms. Its execution time was compared to that of the optimal (space filling ordered) pull reduction that uses scattered accesses for determining speedup.

9.1 For the Intel®Xeon®E5-2680 v2 Processor

Figure 12a and 12b show that on this test platform, the look back schedule of gradient computation achieved a speed up of ≈ 1.45. It also achieved a flop rate of 10.69 Gflops/s for the compute phase and 9.86 Gflops/s for the total computation (including the copy phase). This shows that the transformation could close the performance gap observed in the roofline plot of Figure 6. Also, note that the pull reduction does not use look back. Hence, it has a constant execution time in Figure 12a. However, it is not predictable since it is dominated by latency of the scattered data accesses, which is unpredictable. When a different ordering was presented, the pull reduction had different execution time (Table 1). On the other hand, execution time of the look back streaming version is predictable as it solely depends on the system bandwidth (which is deterministic as per the machine model) and data volume (which can be estimated precisely from the look back schedule that is derived from the machine model and the look back transformation). It was predicted (see Figure 13a) from Equation (1) which reduces to V/B_e for the bandwidth bounded gradient computation with a static schedule. B_e is 45 GB/s as indicated by the STREAM benchmark [36] and V is estimated from the look back schedule and plotted for various look back buffer sizes as shown in Figure 13b. In Figure 13a, the dashed lines indicate the actual time. It shows that the prediction is accurate enough to know in advance the amount of performance improvement that can be obtained from the transformation. The reason for the slight undershooting of the predicted time is revealed when bandwidth of the

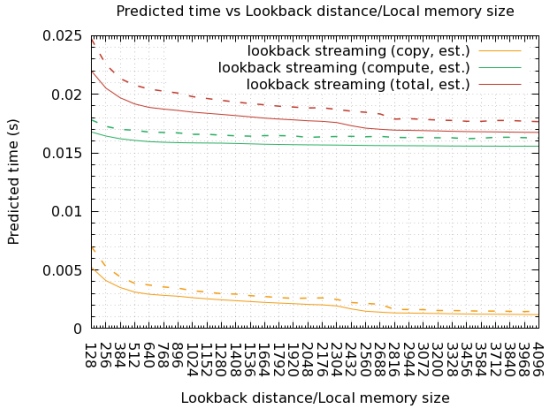


(a) Time vs look back distance (or local memory size) for the look back streaming version of the gradient computation.

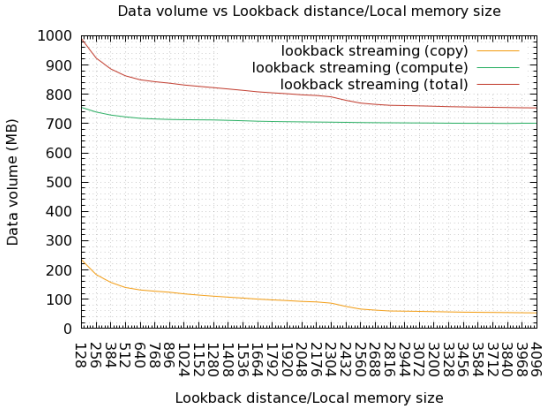


(b) Speedup vs look back distance (or local memory size) for the look back streaming version of the gradient computation with respect to the optimized (ordered by space filling curve) pull reduction that uses scattered accesses.

Figure 12: Results of the gradient computation on the Intel®Xeon®E5-2680 v2 processor.



(a) Predicted time vs look back distance (or local memory size) for the look back streaming version of the gradient computation.



(b) Predicted data volume vs look back distance (or local memory size) for the look back streaming version of the gradient computation.

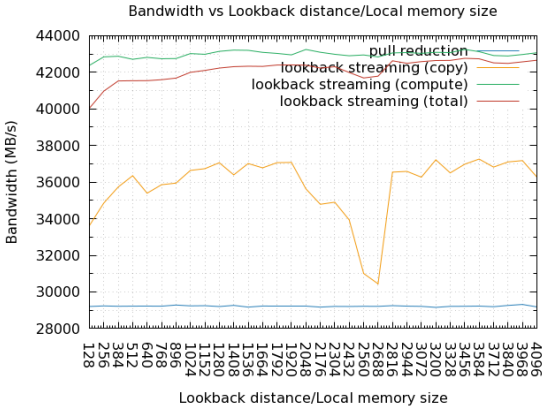
Figure 13: Results of the gradient computation on the Intel®Xeon®E5-2680 v2 processor.

copy and compute phases was observed (Figure 14a). The compute phase achieved ≈ 43 GB/s while the copy phase achieved only ≈ 37 GB/s of bandwidth. Tuning for software prefetching did not improve the bandwidth any further.

9.2 For the Intel®Xeon®Phi ™Processor

It is clear from the results in the last section that higher the bandwidth and larger the look back, higher is the performance of a lightweight irregular computation expressed using the block streaming model. The Intel®Xeon®Phi ™processor provides plenty of parallelism and large DRAM bandwidth. If the processor could conform to the block streaming model, it would provide significant performance boost to the irregular computations. This section assesses usefulness of the processor for the model.

The STREAM benchmark was used to get an estimate of sustainable bandwidth on the processor. It reported peak bandwidth of 147 GB/s with 120 threads (two threads per core). Increasing the

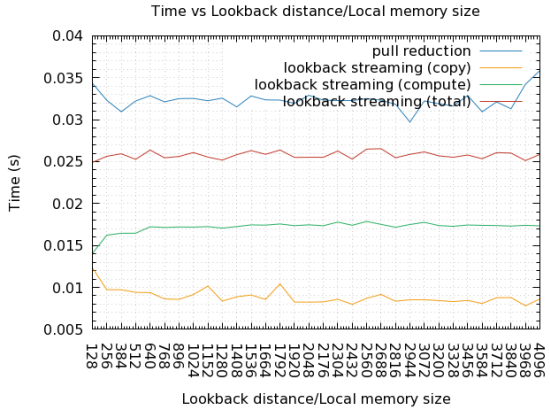


(a) Bandwidth vs look back distance (or local memory size) for the look back streaming version of the gradient computation.

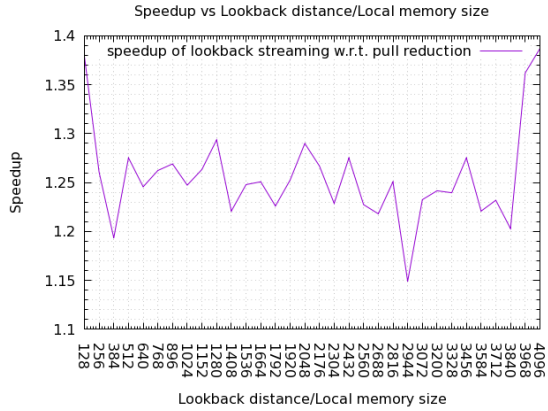
Figure 14: Results of the gradient computation on the Intel®Xeon®E5-2680 v2 processor.

number of threads did not improve the bandwidth. This indicates that two threads per core is sufficient to saturate the memory system [29].

The bandwidth reported by the STREAM benchmark is due to the autovectorization and loop peeling performed by the compiler. These transformations result in SIMD vector load and store from 64 byte aligned addresses in the vectorized portion of the loop. Also, it generates streaming store instruction for the store operation. To ensure aligned load and streaming stores, these tests used a `simdvector<double>` data type that contains eight `double`s and supports arithmetic operations that internally perform explicit vector operations on the data. Also, all the arrays of this data type were allocated on 64 byte boundary. As a result, these tests used only aligned loads. The read streams also used software prefetch instructions that fetches current cache line to the L1 cache and eighth cache line to the L2 cache. For write streams, we used the non-globally ordered `vmovnrngoapd` instruction since it gives better performance compared to the `vmovnrapd` instruction for write only data.

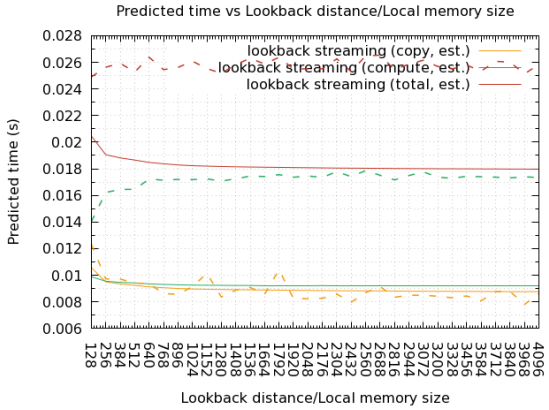


(a) Time vs look back distance (or local memory size) for the look back streaming version of the gradient computation.

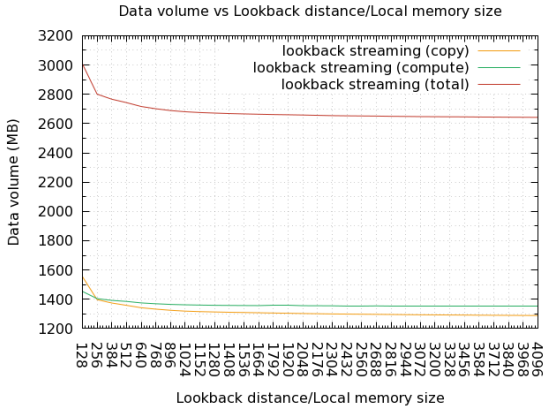


(b) Speedup vs look back distance (or local memory size) for the look back streaming version of the gradient computation with respect to the optimized (ordered by space filling curve) pull reduction that uses scattered accesses.

Figure 15: Results of the gradient computation on the Intel®Xeon®Phi ™processor.

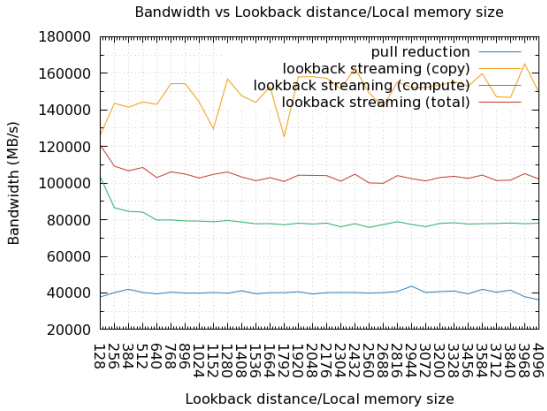


(a) Predicted time vs look back distance (or local memory size) for the look back streaming version of the gradient computation.



(b) Predicted data volume vs look back distance (or local memory size) for the look back streaming version of the gradient computation.

Figure 16: Results of the gradient computation on the Intel®Xeon®Phi ™processor.



(a) Bandwidth vs look back distance (or local memory size) for the look back streaming version of the gradient computation.

Figure 17: Results of the gradient computation on the Intel®Xeon®Phi ™processor.

With this configuration, the gradient computation performed using the look back schedule gave results as shown in Figures 15, 16 and 17. The graphs in Figure 15 show that the look back version of the gradient computation performs better than the scattered access pull reduction version and obtains a speedup of ≈ 1.25. However, the predicted time in Figure 16a shows that though the copy phase time accurately matches with the predicted time, the compute phase takes almost twice the amount of predicted time. The reason for this discrepancy was revealed when the bandwidth was analyzed as shown in Figure 17. The copy phase could achieve peak bandwidth while the compute phase could achieve only 80 GB/s to 100 GB/s of bandwidth, which is significantly lower than the peak sustainable bandwidth. The copy phase uses strictly streaming access that the STREAM benchmark also uses. But the compute phase uses look back streaming which requires not only efficient hardware or software prefetching, but also suitable cache replacement policy that retains streamed look back data. The Figure 17 also reveals that the scattered access version achieved bandwidth of only 40 GB/s. Hence, even with poorly performing compute phase, the look back version could perform better than the scattered access version. If it could have achieved peak

bandwidth, we could have obtained a speedup of ≈ 2.

To investigate how the Intel®Xeon®Phi ™processor responds to the look back access, we developed a micro benchmark that reads four to six values from a given number of read streams with a look back access, performs a trivial computation on the read data and writes the result sequentially to a given number of write streams. Each stream uses the `simdvector<double>` data type and uses the same platform tuning that the streams in the gradient computation used. The benchmark measured the bandwidth of this computation for various look back distances. The results are as shows in Figure 18. They were obtained under identical thread configuration as the gradient computation, that is two threads on each core, which gives 120 threads in total. The graph shows that the look back streaming bandwidth drops rapidly as the look back distance grows. The result in Figure 17a did not show such a dramatic decrease in the bandwidth because the look back accesses in that case were more localized compared to the look back accesses in the benchmark runs. For a processor conforming to the block streaming model, the theoretical estimation of the number of streams for various look back distances is given by $\lfloor \frac{M}{L*S} \rfloor$, where M is the size of the look back buffer, L is the look back distance and S is the byte size of the stream data type. If we assume that each core has two threads, L2 cache functions as the look back buffer, it is shared equally by the two threads and the stream data type is `simdvector<double>` (i.e. $M = 256$KB, $S = 64$bytes), the estimation of the theoretical number of read streams that the processor should support when conforming to the block streaming model is as shown in Table 2.

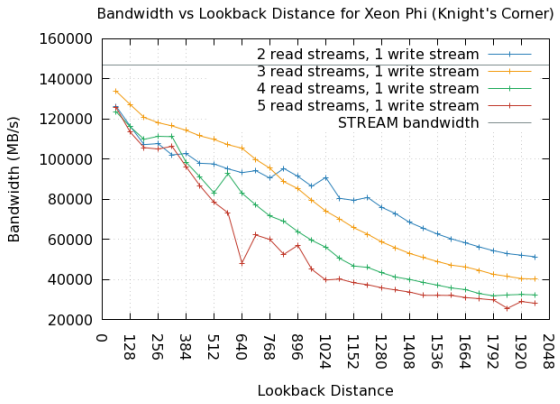


Figure 18: Look back streaming bandwidth for various look back distances and various combinations of the number of read and write streams on the Intel®Xeon®Phi ™processor.

Table 2: Theoretical number of read streams per thread for a given look back distance on the Intel®Xeon®Phi ™processor, assuming that each core has two threads and the L2 cache is used as look back buffer for each read stream.

Look back distance	Maximum number of read streams
64	64
128	32
192	21
256	16
320	12
384	10
448	9
512	8
576	7
640	6

The results of the benchmark and the theoretical estimation indicates that the issue with the

look back streaming on the Intel®Xeon®Phi ™processor is not due to the total L2 cache capacity but maybe due to the cache replacement policy that evicts the cached data too early or maybe due to the inability of the cache system to segregate the exclusive data of each thread to its local L2 cache.

9.3 Discussion

The results presented in this section point to a limitation of the modern processors. They have sophisticated memory system that caters to the scattered access very well, but lacks performance guarantee of a simple sequential access such as the block streaming and look back access required by this model. If hardware provides certain mode that allows it to conform to the model, programmers will not get surprises when the intent is clear from use of the model. Also, it will make the cost prediction highly accurate.

10 Related Work

The gap between computational rate and the data access rate have been growing exponentially each year [36,37]. As a result, the performance of a low computational intensity computation is bounded by the I/O cost. There are primarily two architectural features intended to reduce the I/O cost – multi-level cache and multi-context cores.

From a long time, processors have provided a hierarchical memory (cache) that applications can utilize to reduce the effective latency of the I/O operations. Cache blocking, that involves dividing the total working set into chunks so that each chunk fits in some level of the cache, is often used to improve performance of algorithms on a cache-based processor. Since the execution time of an I/O bounded computation depends on the amount of data movement across the memory hierarchy, cache complexity is used as a metric to determine performance of a computational schedule. To use the cache efficiently, algorithm designers need to use some form of progressive partitioning or clustering to form chunks that fit in some level of the cache. These algorithms require tuning on a specific hardware architecture to determine the threshold for partitioning or clustering. Larger than the optimal-sized chunks increase cache thrashing and degrade performance. On the other hand, smaller than the optimal-sized chunks increase the overhead and degrades performance. This overhead is due to scheduling [7] and/or the replication of data and computations at the chunk boundaries [16] and/or concurrent read-after-write coherency traffic at the chunk boundaries [7]. An important metric that captures the performance of a cache-based algorithm is the cache complexity. There are several models for the analysis. They differ in the assumptions they make about the architecture. For example, the parallel external memory (PEM) model [4] assumes that each core in a multi-core processor has private cache, they share an external memory, and the data transfer between the two levels of memories takes place in blocks. Therefore, the model estimates the number of blocks that transfer between the two levels of memories and uses it to determine an efficient algorithm. The universal multi-core model (UMM) [44] extends the memory hierarchy game (MHG) of [43] to a mulicore processor with various degrees of cache sharing. They observed that appropriate blocking of the computation DAG results in better cache complexity. The Multi-BSP model [49] assumes a hierarchical memory and corresponding hierarchical computation structure. Each level in the computation hierarchy has computation, communication, and synchronization phases just like the BSP model [48]. There are several schedulers that reduce the cache misses for a given computational DAG under different assumptions about the cache hierarchy and sharing. Some examples are the work-stealing scheduler [12] for a private cache multiprocessor, the parallel depth first (PDF) scheduler [9, 10, 13] for a shared cache multi-core processor, and the CONTROLLED-PDF scheduler [7] which is suitable for private-shared cache multi-core processor.

Some modern processors provide concurrent execution contexts on each core with inexpensive context switches [17, 18, 26, 40, 41]. Therefore, if a context stalls due to the data access latency, another ready-to-execute context can be quickly scheduled for execution. The scope of this approach is wider than the cache-based approach since it aims to hide the memory, arithmetic and any other latencies in a computation and achieve the peak machine throughput. The key tuning parameter is

the concurrency. Using less than the optimal concurrency exposes the latency to the computation, while using more than the optimal concurrency does not provide any benefit in terms of improvement in throughput of the computation. On the contrary, it reduces the availability of resources such as registers and scratchpad memory per context. In practice, determining the optimal concurrency is a challenging problem. There are several models for this [6, 23, 24, 45, 50, 52]. The work of Volkov [50] gives an insight into the interplay between concurrency and arithmetic intensity of a computation, arithmetic latency and throughput, memory latency and throughput of the processor. It highlights the relationship among these factors and the peak flop rate that the computation can achieve. For a low arithmetic intensity computation, the peak flop rate with sufficiently large concurrency is bounded by peak throughput of the memory system multiplied by the arithmetic intensity. These computations are categorized as bandwidth-bounded. For a high arithmetic intensity computation, the peak flop rate with sufficiently large concurrency is bounded by the peak throughput of the arithmetic units. They are categorized as compute-bounded. The type of boundedness determines whether the concurrency is used for hiding the memory latency or the arithmetic latency.

Recent trends on the CPU [36,37] and GPU [50] architectures show that the arithmetic throughput is improving exponentially faster than the memory throughput, and the arithmetic latency is reducing much faster than the memory latency with every new generation of the processor. This means that the threshold of arithmetic intensity required to make a computation compute-bounded is increasing exponentially with each new processor generation. The cusp graphs in Figure 4.6 of [50] show this trend. It implies that a computation that is compute-bounded on a processor architecture may become bandwidth-bounded on a future generation of the processor. In addition to this, many computations with low arithmetic intensity (e.g. 0.18 flop/byte in case of the gradient computation) are already bandwidth-bounded. Therefore, on a multi-context processor, the concurrency is required for hiding the memory latency in all the present and prospective bandwidth-bounded computations. Even with sufficient concurrency to hide the memory latency, the flop rate of the computation is bounded by the memory throughput of the memory system, not by the computational throughput of the processor. This shows that it is imperative to deal with the memory latency and throughput rather than with the arithmetic latency and throughput if these algorithms need sustainable performance as processor technology evolves.

If the memory throughput achieved by a bandwidth-bounded computation determines its flop rate, the question is, whether it should use scattered access or the streaming access to achieve the peak bandwidth. Algorithm design is much easier if it is allowed to use scattered access. However, this convenience has some consequences. Scattered access cannot be planned in advance. Hence, it has to be initiated when the program control reaches the operations which require the data. As a result, concurrency is essential to hide the latency and achieve peak memory throughput of the hardware. A highly concurrent architecture is complicated to design since it requires a scheduler capable of dynamic scheduling of the concurrent execution contexts. This also implies that algorithms need to expose sufficient concurrency, or otherwise bear the memory latency. Moreover, the scattered access is fine grained, which causes inefficient utilization of the DRAM row buffers. This requires higher energy consumption for the data access. Further, the performance model is complicated since it has to take into account a number of hardware and algorithmic parameters such as memory throughput and latency, arithmetic throughput and latency, concurrency, and arithmetic intensity. This makes it harder to determine impact of changes to an algorithm on the overall performance. In contrast to the scattered access, the streaming access requires a simple hardware to hide the memory latency, does not require significantly large concurrency, and reduces energy wastage due to higher utilization of DRAM row buffers. Also, it is much easier to reason about algorithmic changes since the performance model requires consideration of only the processor's arithmetic and memory throughput, and the algorithm's arithmetic intensity.

Though the performance improvements on the test problem and the test platforms considered in this paper are not enough to draw wider conclusions about the usability of the block streaming model, it highlights the importance of considering the impact of the fundamental physical constraints of the memory system design and access, and the processor's architectural parameters (e.g. cache replacement policy, capacity, prefetchers etc.) on algorithm performance. Moreover, this work demonstrates that hardware designers and algorithm designers need to pay attention to these

constraints to ensure sustainable and portable performance of algorithms on future generation of processors.

Most of the modern processors allow fine-grained synchronization among the cores. This gives flexibility to the algorithm designers. However, the synchronization is expensive since it requires cache coherency. Therefore, good schedulers (e.g. [7]) try to minimize the synchronization points. That is, they try to determine largest chunks of work that can be executed by the cores without requiring communication with the other cores. The block streaming model makes this requirement explicit in the model specification by enforcing that the access regions of the channels do not overlap at any instant. Therefore, the cores can share data only at the synchronization points via the global memory. While this restriction is inconvenient for the design of algorithms, it ensures that the performance is predictable.

The transformation described in Section 6 is similar to the transformation performed by the `milk` OpenMP extension [30] that uses multi-pass radix partitioning. However, there are several differences. It performs full replication of the source data in the first pass and generates key-value pairs. Then it performs contraction of the data by joining the values that are simultaneously present in the cache resident data containers in the successive stages. In contrast to this, our approach expands the data only when it needs to. In addition to this, it also provides opportunities to control the data expansion through fusion of the streaming stages. In context of the transformation described in Section 6, the `milk` extension works similar to the pull reduction obtained by replacing all the distribution stages with mapped join stages. In this way, our approach is inclusive. Further, the cost model gives an accurate estimate of the execution time. Indeed, through this estimation we quickly predicted that using only the radix clustering will produce a highly inefficient implementation for our case. It provided us the motivation to find the look back reordering optimization that produced performance improvement on both the test platforms.

11 Conclusion

This work outlined a machine model that guarantees data access performance at the expense of having only block streaming access in the global memory. However, algorithmic transformations can be devised to work around the limitation. This work demonstrated use of the model for a practical case of irregular access computation and showed that the extra effort of redesigning the algorithm produces an implementation with predictable execution time provided the processor conforms to the block streaming model. We obtained a speed up of ≈ 1.45 for the Intel®Xeon®E5-2680 v2 processor and ≈ 1.25 for the Intel®Xeon®Phi ™processor compared to the traditional optimized way of performing the computations that uses scattered access and space filling curve order to improve locality. The computation could have achieved a speed up of ≈ 2 on the Intel®Xeon®Phi ™processor and a speed up of slightly higher than 1.45 on the Intel®Xeon®E5-2680 v2 processor if they were model conforming. The results also suggest that it is possible to achieve portable performance on any hardware that conforms to the model. Such a model can provide a road map for hardware vendors to provide a stable set of features, which will be required to support software investment in designing run time and scheduling systems, that these architectures desperately need.

12 Acknowledgment

Thanks to the High Performance Computing Collaboratory and the Center for Advanced Vehicular Systems at the Mississippi State University for supporting this work and providing necessary resources. We also thank the anonymous reviewers for helpful comments on this paper.

References

[1] CS roofline toolkit, 2016. `https://bitbucket.org/berkeleylab/cs-roofline-toolkit`.

[2] ISO International Standard ISO/IEC 14882:2017(E) – Programming Language C++, 2017. https://isocpp.org/std/the-standard.

[3] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 305–314, New York, NY, USA, 1987. ACM.

[4] Lars Arge, Michael T Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 197–206. ACM, 2008.

[5] Michael Bader. *Space-filling curves: an introduction with applications in scientific computing*, volume 9. Springer Science & Business Media, 2012.

[6] Sara S Baghsorkhi, Matthieu Delahaye, Sanjay J Patel, William D Gropp, and Wen-mei W Hwu. An adaptive performance modeling tool for gpu architectures. In *ACM Sigplan Notices*, volume 45, pages 105–114. ACM, 2010.

[7] Guy E Blelloch, Rezaul A Chowdhury, Phillip B Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 501–510. Society for Industrial and Applied Mathematics, 2008.

[8] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. *SIGPLAN Not.*, 47(8):181–192, February 2012.

[9] Guy E Blelloch and Phillip B Gibbons. Effectively sharing a cache among threads. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 235–244. ACM, 2004.

[10] Guy E Blelloch, Phillip B Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM (JACM)*, 46(2):281–321, 1999.

[11] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[12] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[13] Shimin Chen, Phillip B Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C Mowry, et al. Scheduling threads for constructive cache sharing on cmps. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 105–115. ACM, 2007.

[14] Elizabeth Cuthill. Several strategies for reducing the bandwidth of matrices. In *Sparse Matrices and Their Applications*, pages 157–166. Springer, 1972.

[15] Elizabeth Cuthill and James McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172. ACM, 1969.

[16] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. Avoiding communication in sparse matrix computations. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[17] Martti Forsell. Configurable emulated shared memory architecture for general purpose mpsocs and noc regions. In *Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, pages 163–172. IEEE Computer Society, 2009.

[18] Martti Forsell, Erik Hansson, Christoph Kessler, Jari-Matti Mäkelä, and Ville Leppänen. Numa computing with hardware and software co-support on configurable emulated shared memory architectures. *International Journal of Networking and Computing*, 4(1):189–206, 2014.

[19] Matteo Frigo and Steven G Johnson. FFTW: An adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384. IEEE, 1998.

[20] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE, 1999.

[21] Samuel H. Fuller and Lynette I. Millett. Computing performance: Game over or next level? *Computer*, 44(1):31–38, 2011.

[22] David Hilbert. Ueber die stetige abbildung einer line auf ein flächenstück. *Mathematische Annalen*, 38(3):459–460, 1891.

[23] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.

[24] Jen-Cheng Huang, Joo Hwan Lee, Hyesoon Kim, and Hsien-Hsin S Lee. Gpumech: Gpu performance modeling technique based on interval analysis. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 268–279. IEEE Computer Society, 2014.

[25] Intel. *Intel® Xeon® Phi ™ coprocessor instruction set architecture reference manual*, September 2012.

[26] Intel. *Intel® Xeon® Phi ™ coprocessor system software developers guide*, March 2014.

[27] Intel. *Intel ® 64 and IA-32 architectures optimization reference manual*, June 2016.

[28] Intel. *Intel ® 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, July 2017.

[29] Intel. *Intel® Xeon® Phi ™ processor software optimization guide*, April 2017.

[30] Vladimir Kiriansky, Yunming Zhang, and Saman Amarasinghe. Optimizing indirect memory references with `milk`. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation - PACT '16*, September 2016.

[31] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't and why. *ACM Trans. Archit. Code Optim.*, 9(1):2:1–2:29, March 2012.

[32] Yu Jung Lo, Samuel Williams, Brian Van Straalen, Terry J Ligocki, Matthew J Cordery, Nicholas J Wright, Mary W Hall, and Leonid Oliker. Roofline model toolkit: A practical tool for architectural and program analysis. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 129–148. Springer, 2014.

[33] EA Luke and Pasqualle Cinnella. Numerical simulations of mixtures of fluids using upwind algorithms. *Computers & Fluids*, 36(10):1547–1566, 2007.

[34] Edward A Luke and Thomas George. Loci: A rule-based framework for parallel multi-disciplinary simulation synthesis. *Journal of Functional Programming*, 15(3):477–502, 2005.

[35] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):709–730, 2002.

[36] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[37] John D. McCalpin. Memory bandwidth and system balance in hpc systems. Invited talk, Supercomputing 2016, Salt Lake City, Utah, 2016.

[38] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 425–433, New York, NY, USA, 1999. ACM.

[39] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Atomic-free irregular computations on gpus. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 96–107, New York, NY, USA, 2013. ACM.

[40] John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, 30(2), 2010.

[41] NVIDIA. CUDA toolkit documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[42] OpenMP Architecture Review Board. OpenMP application program interface version 4.5, 2015. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[43] John E Savage. Extending the hong-kung model to memory hierarchies. In *International Computing and Combinatorics Conference*, pages 270–281. Springer, 1995.

[44] John E Savage and Mohammad Zubair. A unified model for multicore architectures. In *Proceedings of the 1st international forum on Next-generation multicore/manycore technologies*, page 9. ACM, 2008.

[45] Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. In *ACM SIGPLAN Notices*, volume 47, pages 11–22. ACM, 2012.

[46] Emre Sozer, Christoph Brehm, and Cetin C Kiris. Gradient calculation methods on arbitrary polyhedral unstructured meshes for cell-centered cfd solvers. In *52nd Aerospace Sciences Meeting*, page 1440, 2014.

[47] Josep Torrellas, HS Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[48] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[49] Leslie G Valiant. A bridging model for multi-core computing. *Journal of Computer and System Sciences*, 77(1):154–166, 2011.

[50] Vasily Volkov. *Understanding latency hiding on gpus.* PhD thesis, UC Berkeley, 2016.

[51] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[52] Yao Zhang and John D Owens. A quantitative performance analysis model for gpu architectures. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 382–393. IEEE, 2011.

[53] Anup Zope and Edward Luke. A block streaming model for irregular applications. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (APDCM)*, May 2018.