Intel's Array Building Blocks: A Retargetable, Dynamic Compiler and Embedded Language

Chris J. Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu and Dan Zhang
Performance and Productivity Libraries, Software and Services Group, Intel Corporation

Our ability to create systems with large amount of hardware parallelism is exceeding the average software developer's ability to effectively program them. This is a problem that plagues our industry. Since the vast majority of the world's software developers are not parallel programming experts, making it easy to write, port, and debug applications with sufficient core and vector parallelism is essential to enabling the use of multi- and many-core processor architectures. However, hardware architectures and vector ISAs are also shifting and diversifying quickly, making it difficult for a single binary to run well on all possible targets. Because of this, retargetability and dynamic compilation are of growing relevance.

This paper introduces Intel® Array Building Blocks (ArBB), which is a retargetable dynamic compilation framework. This system focuses on making it easier to write and port programs so that they can harvest data and thread parallelism on both multi-core and heterogeneous many-core architectures, while staying within standard C++. ArBB interoperates with other programming models to help meet the demands we hear from customers for a solution with both greater programmer productivity and good performance.

This work makes contributions in language features, compiler architecture, code transformations and optimizations. It presents performance data from the current beta release of ArBB and quantitatively shows the impact of some key analyses, enabling transformations and optimizations for a variety of benchmarks that are of interest to our customers.

I. INTRODUCTION

Parallelism has been a feature of mainstream processor architecture for at least two decades. From the processor architect's point of view, increasing parallelism (in functional units, in the memory system, and in core count) is an inexpensive mechanism for increasing performance. However, the pursuit of micro-architectures that automate the dispatch of parallel work from traditional sequential programs has hit the point of diminishing returns, limited by power constraints and design complexity.

The recent attention directed towards exposing parallelism in software is the result of a major shift in responsibility from processor architects to software developers. This trend is manifest in several software-visible changes:

- The enrichment and widening of vector instructions, where SIMD vector parallelism is explicit (SSE, AVX, Intel® Many-Integrated Core Architecture (MIC) [Skaugen 2010], Altivec) or implicit (GPUs [Buck 2007, Owens 2005]);
- The use of simultaneous multithreading techniques in cores (hyper-threading, MIC, Sun Niagara);
- The use of multi-core architectures on a single die, composable into multi-socket systems (Pentium D, Sun Niagara, Core 2, Core i*, etc.);
- The use of heterogeneous computing engines with differentiated memory, ISA, and performance behaviors (Sandy Bridge, AMD Fusion).

Each of these architectural trends requires software to expose and manage parallelism in order to effectively utilize the hardware. But that is difficult. Practitioners of high performance computing are well versed in low-level parallel programming techniques to directly target each of these mechanisms. However, mainstream developers typically have very limited exposure to (and facility with) these mechanisms. Parallel programming is also complicated by the need to isolate independent computations. While software engineering mechanisms such as encapsulation hint at the need to isolate *effects* to the bare minimum necessary for a functional interface, they do not have the same burden of completeness (for correctness' sake) that parallelism has. Parallelism bugs related to poor isolation (*i.e.*, data races) are a newer class of error that has a particularly insidious effect on productivity and quality: they are difficult to reproduce and often lay dormant until used in a particular context.

As computation resources become less expensive, productivity becomes more of a concern. Modularity and composability are often essential elements of good software engineering. But most parallel programming models compose poorly. Modern software development tools support modular interfaces through late binding (for example, C++ inheritance and virtual functions; dynamic class loading; dynamic types in scripting languages, etc.). Late binding increases the flexibility and expressiveness of software components. However, late binding can defeat efforts at compiler optimization. The most visible recent language developments in the area of parallelism [Buck 2007, Nickolls 2008, Munshi 2008] essentially require that parallel kernels be inlined first. This is barely an advance from existing parallel programming models, such as OpenMP.

The current trade-off is unattractive: increasing performance through parallelism, in exchange for software that is brittle, non-composing, difficult to understand and maintain, and often non-portable. In light of this, the lack of parallelization in mainstream software development is not surprising. In our experience, most mainstream software developers will prefer lower development cost/risk/time to marginal (and transient) performance improvements.

The design of Intel® Array Building Blocks (**ArBB**) was motivated by several factors. First, in order to be future proofed and portable, a programming model needs to be both adaptive to the hardware architecture and abstracted from it. Portability is a crucial factor in order to allow a significant code base to be built up over time, and is central to productivity and to reducing the cost of maintenance over time. In order to satisfy software engineering practices, a programming model should also support modularity without giving up performance.

ArBB achieves these goals by combining several key technologies. First, it supports generic programming based on dynamic code generation. This enables code generation to adapt to the target architecture, while still allowing the programming model to be abstract. Second, the programming model is based on a structured form of data parallelism that avoids, by construction, common sources of error such as race conditions and deadlock, thereby enhancing productivity. Data parallelism also often scales better than task parallelism. Third, optimizations are included in ArBB that allow computations specified far apart in the source code to be fused together into arithmetically-intensive kernels. This supports modular and generic programming while still achieving high performance, but without the need for manual inlining or flattening.

ArBB has recently been released in beta form by Intel [ArBB 2011], and is a component of the Intel® Parallel Building Blocks. The ArBB interface design and compiler architecture is the result of combining efforts from the Ct project developed at Intel Labs and the commercial RapidMind platform developed at the University of Waterloo. ArBB therefore includes features developed during years of customer interaction and production experience at RapidMind. ArBB provides a unified framework for targeting multiple parallelism mechanisms core, thread, vector (SIMD), and instruction-level parallelism from a simple high level specification of latent parallelism. In ArBB's programming model, synchronization is implicit and the results of all computations are consistent with a single, deterministic, serial ordering. This determinism and sequential consistency simplify maintenance, testing, and debugging. The memory model also provides isolation and can support both shared and distributed memory models. The ArBB architecture supports novel features such as dynamic inlining which can significantly increase performance above and beyond that provided by parallelism. ArBB also includes first-class representations of code objects (closures) allowing explicit manipulation of the code generation process when this is useful. The JIT code generator itself and the interface are designed to allow predictable timing and control over JIT compilation.

We begin with a description of ArBB's language features and embedded interface, which allows for powerful forms of generic programming. We then give an overview of the compilation system and its architecture, and highlight some of the optimizations the compiler uses. We focus on optimizations that that are closely tied to ArBB's central language features and target architectures. Finally, we present data on the compiler's overall performance and discuss the performance impact of these optimizations.

II. ArBB Embedded Language

ArBB supports a structured and deterministic data-parallel programming model. It is an embedded, compiled language whose syntax is implemented as an API. It can be considered a language, although in practice it is used and deployed just like a library.

Standard C++ mechanisms are used to define ArBB types. ArBB types include both scalar values (e.g. integers and floating point numbers) and collections of these. Sequences of operations on these types can then be specified by the programmer in the usual imperative fashion. However, unlike the built-in C++ types, sequences of operations on ArBB types can be "captured" and translated dynamically (at runtime) into vectorized machine language suitable for a variety of target systems. In addition, ArBB uses by-value semantics for assignment between collection variables. Expressions over collections always act as if a completely new collection were created and assignment between collections always behaves as if the entire input collection were copied. This simplifies use of the system: for example, collections can be returned from functions just like a floating point number would. More importantly, it avoids aliasing, since collections have the semantics of values, not pointers. In practice, however, extensive optimizations remove almost all copying.

The ArBB C++ API is layered on top of a virtual machine (VM) [Du Toit 2010], which has its own "extern C" API that can be used with alternative front-ends in the future. The ArBB VM, which is isolated in a dynamic library, implements the compilation and parallelization. The semantics of the ArBB VM is based on abstract forms of parallelism. Lots of latent parallelism can be expressed by use of both sequences of vector operations and the application of elemental functions to collections. Elemental functions are mapped over all collection elements. Vector operations act directly on collections.

The VM is responsible for mapping the abstract, latent parallelism expressed through its API onto actual parallel mechanisms in the physical machine. Currently, ArBB supports both SIMD instruction (vector) and core (thread) hardware parallelism. It can also use latent parallelism for other purposes, for example to hide latency via streaming, prefetching, and pipelining. The physical vector width and number of cores are abstracted away in the VM. Because of this, and since the vector machine language is generated dynamically, code written with ArBB is portable across multiple vector ISAs (SSEn, AVX, MIC), *even without recompilation by C++*, and scales automatically to use available cores. Special features of the target instruction sets, such as scatter/gather, are also supported. The ArBB VM also supports an abstract model of memory that can transparently support remote memory, such as that in an attached co-processor (such as the MIC Architecture), while automatically avoiding unnecessary communication costs.

The ArBB language is designed to improve productivity in two ways. First, it presents a simplified programming model that focuses on latent parallelism, memory locality, and common structured patterns of computation [Siu 1996, Skillicorn 1998, Bosch 1998, Bromling 2002, Aldinucci 2007, McCool 2010]. This simplifies the software developer's task, especially since the deterministic patterns used avoid, "by design," common sources of error, such as race conditions and deadlock. The code is shorter and closer to a mathematical specification of the problem, which simplifies maintenance. At the same time, the two key factors needed for performance, parallelism and locality, are cleanly expressible. Second, the code is portable. This enhances productivity since minimal to no effort is required to move applications to new processors, even those with radically different underlying architectures.

The details of the ArBB C++ interface are best presented using an example. Figures 1 to 3 give two implementations of the Mandelbrot set computation. The first is in terms of elemental functions; the second is in terms of sequences of vector operations. These two implementations are equivalent.

Figure 1 specifies an elemental function for computing a single pixel of the Mandelbrot set output. An **elemental function** is conceptually a scalar computation, but is applied to every element of a collection. The independent computations on every element can be performed in parallel across both cores and SIMD lanes. Figure 2 shows the code for setting up and invoking this elemental function.

```
int max_count;                  // C++ non-local; by value
std::complex<f32> offset;       // ArBB non-local; by ref
void mandel(
  i32& d,                       // elemental output
  std::complex<f32> c           // elemental input
) {
  i32 i;
  c += offset;
  std::complex<f32> z = 0.0f;
  _for (i = 0, i < max_count, i++) {
    _if (abs(z) >= 2.0f) {
      _break;
    } _end_if;
    z = z*z + c;
  } _end_for;
  d = i;
}
```

*Figure 1: Elemental function for computing the **Mandelbrot** set. C++ non-local variables are bound by value ("frozen") when the ArBB function is first compiled, i.e., at capture time (see below) or upon first invocation via a* `map` *or* `call`*. ArBB non-local variables are bound by reference and may be updated dynamically. Note that complex numbers can be expressed using the standard templates in C++. In general, arbitrary user types, including operator overloading, are supported in ArBB.*

```
void mandelbrot1(
  dense<i32,2>& dest,    // array output
  dense<f32> sR,         // row scale (input)
  dense<f32> sC          // column scale (input)
) {
  dense<f32,2> sR2 = repeat_col(sR, N_COLS);
  dense<f32,2> sC2 = repeat_row(sC, N_ROWS);
  dense<std::complex<f32>,2> tscale;
  tscale.set<0>(sR2);   tscale.set<1>(sC2);
  mandel(dest, tscale);
}
void mandelbrot1_call(int* res_arbb) {
  dense<f32> sR; bind(sR, scale, N_ROWS);
  dense<f32> sC; bind(sC, scale, N_COLS);
  dense<i32,2> dest;
  bind(dest, res_arbb, N_COLS, N_ROWS);
  call(mandelbrot1)(dest, sR, sC);
}
```

*Figure 2: Wrapper code to invoke the elemental function implementation of the **Mandelbrot** set. The* `map` *operation applies the elemental function to every element of a collection. A* `call` *operation invokes a sequence of parallel operations (here consisting only of a single map). The* `bind` *function is one of two ways (the other uses iterators) to transfer data in and out of ArBB data space.*

Figure 3 shows an alternative implementation of the same computation, but this time using vector operations directly on collections. The performance of the two forms should be similar, so the software developer can choose whichever approach is more convenient.

```
void mandelbrot2(
  dense<i32,2>& dest,
  dense<f32> sR, dense<f32> sC
) {
  dense<f32,2> sR2 = repeat_col(sR, N_COLS);
  dense<f32,2> sC2 = repeat_row(sC, N_ROWS);
  dense<std::complex<f32>,2> tscale;
  tscale.set<0>(sR2);
  tscale.set<1>(sC2);
  dest = fill(i32(0), N_COLS, N_ROWS);
  dense<std::complex<f32>,2> z =
    fill(std::complex<f32>(0.f,0.f),
    N_COLS, N_ROWS);
  i32 i;
  _for (i = 0, i < max_count, i++) {
    dense<boolean,2> done = (abs(z) < 2);
    dest = select(done, dest + 1, dest);
    z = select(done, z*z + tscale + offset, z);
  } _end_for;
}
void mandelbrot2_call(int* res_arbb) {
  dense<f32> sR; bind(sR, scale, N_ROWS);
  dense<f32> sC; bind(sC, scale, N_COLS);
  dense<i32,2> dest;
  bind(dest, res_arbb, N_COLS, N_ROWS);
  call(mandelbrot2)(dest, sR, sC);
}
```

*Figure 3: Another implementation of the **Mandelbrot** set using vector operations. Here the* `call` *operation, instead of invoking a* `map`*, performs a sequence of vector operations directly. These are fused in a form equivalent to the elemental function implementation, including early exit optimizations.*

Figure 4 demonstrates an additional novel feature of ArBB: the ability to explicitly construct and manipulate code objects, which we call `closures`. The `call` operation combines several steps. The first time it is called with a particular function pointer, ArBB captures the sequence of operations over ArBB types and creates IR. ArBB then compiles this sequence to vector machine language, caches the resulting machine code, and then invokes it (in parallel on multiple cores, if appropriate). The second time `call` is invoked on the same function pointer, ArBB reuses the cached machine language rather than regenerating it.

```
typedef
void F(dense<i32,2>&, dense<f32>, dense<f32>);
max_count = 100;
closure<F> mandelA = capture(mandelbrot2);
max_count = 1000;
closure<F> mandelB = capture(mandelbrot2);
```

*Figure 4: Code for capturing two different variants of the vector implementation of the **Mandelbrot** set. The C++ non-local variable* `max_count` *is frozen during capture, but we can update it and capture multiple times to create variants. Once constructed, closures can be invoked like functions. The* `call` *operation actually returns a* `closure`*. Note that* `closures` *are typed based on the type of function they are constructed from. Dynamically typed versions of* `closures` *are also available for convenience, but generally ArBB is statically type checked.*

However, each of these steps can be invoked individually when necessary, and the caching can also be avoided when it is not appropriate. In particular, the `capture` API call only (and always) does the capture step, and returns an object called a `closure` representing the captured code sequence. This `closure` object can be invoked like a function in a `call` or `map`. Every `capture` "freezes" the values of C++ non-locals and the effects of non-ArBB control flow, which can be varied for different captured closures. *This allows C++ to be used as a metaprogramming language for generating ArBB variants, supporting a powerful form of generic programming.*

The form of metaprogramming used by ArBB is different from, and more powerful than, template metaprogramming. "Template metaprogramming" is used for several high performance C++ libraries [Veldhuizen 1999, Abrahams 2004]. Template metaprogramming uses the template rewriting mechanisms in C++ to transform user-provided code sequences into more efficient forms. In such template libraries, the template rewriting rules are used as an "accidental functional language" to manipulate C++ code at C++ compile time. However, template metaprogramming significantly increases C++ compile times and library complexity, and is limited in the sophistication of the optimizations that can be performed. Additionally, the end user needs access to the source code of the library. In contrast, ArBB generates code at run time, and ordinary C++ can be used to manipulate the generated code. The term generative metaprogramming [Herrington 2003, Czarnecki 2000] has been used for this form of metaprogramming.

Templates *are* used in the C++ API to ArBB, but only for the usual purposes: to provide a more generic and parameterizable type system. It is *not* necessary for a library writer using ArBB to expose their source code: such libraries can be provided in precompiled binary form only. Even in this case, library writers using ArBB can implement significant optimizations such as (dynamically) inlining both function pointer-based callbacks and virtual function overloads provided by the users of their libraries.

Several other examples are used later in this paper. In order to make this paper self-contained, Table I, below, summarizes the ArBB operations used in these examples. Please see the ArBB documentation [ArBB 2011] for more details.

III. Compiler System Overview

It is our philosophy that enabling productivity entails leaving programmers as free as possible to code in their language of choice. To that end, our long-term vision includes being able to support a variety of front ends. Some possible front ends could include Python® [Clyther 2010], Microsoft® .NET, quant-centric languages from the financial services industry, and shader languages. It's important to be able to support a large space of target architectures, including multi-core and many-core CPUs and many ISAs, including those with SSE, AVX, and the instruction set for the MIC architecture.

TABLE I. ARRAY BUILDING BLOCKS KEYWORDS

`repeat_row, repeat_col` Extend a 1D array to a 2D array by replication of a single row or column
`fill` Create a new array with all elements initialized to some value.
`*collection*.set<*i*>` Initialize the *i*th component of an collection (array) of structures
`bind` Associate a C array with an ArBB collection.
`replace` Update a given element of a collection with a new value. Returns an updated collection.
`_for, _end_for, _while, _end_while, etc.` Versions of control flow constructs that can be captured into closures. Ordinary C control is executed only when the closure is constructed (which is still useful for generic programming). Only the ArBB forms of control flow can depend on computed values of ArBB types.
`uncaptured<T>::type` Find the C type corresponding to a given ArBB type.
`f32, f64, i32, u32` Scalar ArBB types corresponding to float, double and signed and unsigned 32-bit integers.
`dense<*T*, *D*>` A dense array with elements of type *T* (which may be single ArBB scalars or structures of such scalars) of dimensionality *D*. The size of the collection varies at runtime. Assignment is semantically by-value.
`add_reduce` Combine all the elements in the last dimension of a collection into a single element by addition, reducing the dimension of the collection by one. Returns a scalar for 1D inputs, 1D collections for 2D inputs, and 2D collections for 3D inputs. Generalizations for other associative operators are available.
`add_scan, add_iscan, etc.` Parallel prefix operations, exclusive and inclusive. Generalizations for other associative operators are available.
`pack, unpack` Select elements from a collection and place adjacently in a new collection, and the inverse operation.
`select` Based on a Boolean collection, generate a new collection by drawing elements from one of two input collections. A vectorized version of the ?: operator.
`*collection*[u]` Random read (gather) from a collection at location(s) `u`.
`call` Invoke a sequence of (usually parallel) operations. The argument can be a C function (which is captured, compiled, and cached) or a closure.
`capture` Construct a `closure` by invoking a C function and recording the sequence of ArBB operations invoked.
`map` Generate a parallel operation by replicating a function over the elements of one or more colections with the same shape.
`closure<*F*>` Type representing a captured function with signature *F*.
`shift, neighbor` Returns the elements from the container at the given offset from the current position.

We also have an interest in supporting standard back ends like OpenCL and LLVM. This creates an “M by N” problem, as shown in Figure 5, with a combinatorial explosion of front end languages and back end targets. We address this problem, in part, by creating an open interface to our VM [Du Toit 2010], that has interfaces for different front ends to our common compiler infrastructure, and that offers a variety of selectable back end options.

We stated above that the compiler is actually a library, and that it works with standard compilers. A new target may be supported by simply deploying an updated ArBB dynamic library, without modifying the source code or invoking the C++ compiler.

Here’s a summary of how the system works. The C++ compiler uses header files and templates to express the embedded language functions like `call` and `map`, as well as constructors and operations over ArBB types, in terms of calls to ArBB dynamic library entry points.

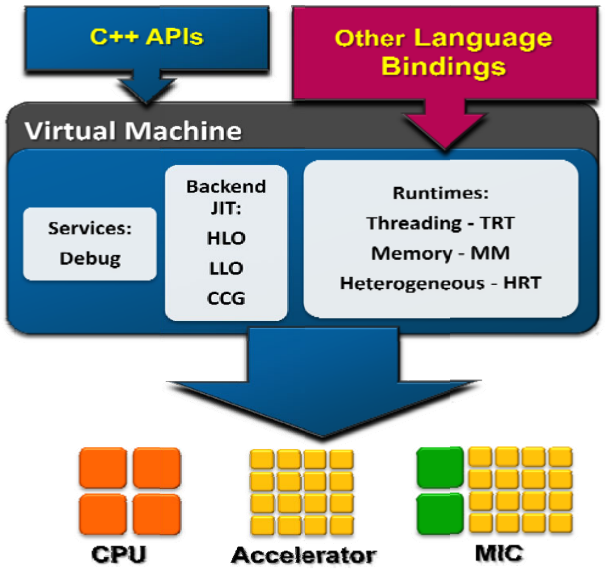

Figure 5: ArBB compiler architecture. The back end of the compiler consists of a High-Level Optimizer (HLO), Low-Level Optimizer (LLO), and Converged Code Generator (CCG). The Threading Runtime (TRT) is built on top of Intel® Threading Building Blocks, and share a resource manager with other Intel and third-party parallelization tools. The Heterogeneous Runtime (HRT) is used to coordinate the loading and executing code and moving data to and from one or more (possibly remote) acceleration devices. Vector and scalar memory spaces are garbage collected by the memory manager.

When `call` (or `capture`) is invoked the ArBB library enters a “capture mode” and then calls the C++ function specifying the operation. In capture mode, instead of executing operations on ArBB types immediately, a trace of the invoked sequence of operations on ArBB types (along with tokens from ArBB control flow) is recorded and used to create an internal representation (**IR**) specifying the computation. Upon returning from such functions, code is JITted for the target(s), and is available for subsequent reuse without recompilation. Native C/C++ code that is intermixed with ArBB code is executed only during the initial capture.

Because native and ArBB code operate in different logical memory spaces, dependences between them arise only from ArBB operators such as `bind` and read/write range iterators. Since data copies and synchronization are implicit, the mechanisms or even necessity of copying and synchronization can be left up to the implementation. This is an important performance optimization and productivity enhancement with respect to CUDA [Nickolls 2008], for example.

IV. Code Optimizations

The ArBB compiler builds and optimizes the intermediate representation (IR) to improve performance at different levels. The High-Level Optimizer (**HLO**) performs architecture-independent optimizations to reduce memory usage, threading overhead, and redundant computation, and improve data locality and affinity. The Low-Level Optimizer (**LLO**) takes the HLO IR and generates platform-independent code with SIMDization and thread parallelization. Finally, the Converged Code Generator (**CCG**) generates an optimized binary to run on a target platform. In principle, some of the analysis phases of this compilation could be done offline, while more machine specific and run-time data-dependent phases could be done dynamically [Nuzman 2011].

A. High-Level Optimizer

There are about forty optimizations in HLO, which can be grouped into three phases. Optimizations in italics are highlighted and described with examples in Section VI.D. Figure 6(a) is a sample ArBB program that illustrates how HLO transforms the IR into an optimized form in different phases. HLO initially builds an IR that retains high-level semantics, as illustrated in Figure 6(b).

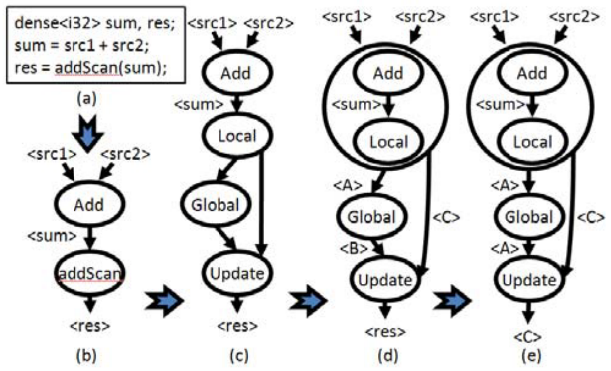

Figure 6: HLO Transformations. (a) Original code (b) High-level IR (c) Decomposed into sub-primitives (d) Fused IR (e) non-SSA IR

1. Phase 1

The first HLO phase performs classical compiler optimizations such as static single assignment form (SSA) transformation, copy propagation, dead code elimination, and constant folding [Allen 2002]. HLO performs these optimizations repeatedly as we transform the IR.

The phase 1 performs several ArBB-unique optimizations. The *SIMDize map* transformation converts a `map` function written in a scalar form to a vectorized `call` function in order to exploit data parallelism.

```
Procedure decompose(graph g)
  for each node n in g
    if n is a hierarchy node
      decompose(n->subgraph);
    else
      decompose(n);
Procedure decompose(node n)
  if opclass(n)=element-wise
    output(n) = local(input(n))
  else if opclass(n)=reduce
    if (n is 2D or 3D) return;
    /* generate following code sequence */
    temp = localReduce(input(n));
    output(n) = globalReduce(temp);
  else if opclass(n)=scan
    if (n is 2D or 3D) return;
    /* generate following code sequence */
    <temp1,temp2> = localScan(input(n));
    temp3 = globalScan(temp2);
    output(n) = updateScan(temp1,temp3);
  else if opclass(n)=permute
    if opkind(n)=pack
      /* generate following code sequence */
      temp1 = localPack(input(n));
      <output(n),temp2> = globalPack(temp1);
      output(n) = updatePack(input(n),temp2);
    else if opkind(n)= scatter
      /* generate following code sequence */
      temp = localScatter(input(n));
      output(n) = globalScatter(temp);
    else … /* other permute kind omitted */
```
Figure 7: Sub-Primitive Decomposition Algorithms.

Sub-primitive decomposition partitions a high-level primitive IR into up to three categories of low-level IRs, called sub-primitives, *i.e.*, local, global, and update. The local sub-primitive computes the element-wise operation within the local task. The global sub-primitive collects and propagates the results of the local sub-primitive from all of the parallel tasks, *e.g.*, for a reduction or scan, thus requiring synchronization. The update sub-primitive computes the final result in each task using the result of the global sub-primitive.

By decomposing the data parallel operation into functional sub-primitives, we can group the ones that have the same communication and synchronization pattern, and implement them together. As a result, the ArBB operators are automatically composable by the compiler. Once a high-level primitive is decomposed into sub-primitives, it does not retain high-level semantics. Figure 7 describes the high level algorithm of Sub-primitive decomposition. It traverses all nodes in an Evaluation Unit (EU), which consists of all ArBB primitives in a function. Depending on the operation class, the high-level primitives are decomposed into different sets of sub-primitives. Figure 6(c) shows the result of Sub-primitive decomposition applied to Figure *6*(b). For example, 'add' is an element-wise operator and it is only decomposed into its local sub-primitives, while 'addScan' is decomposed into local, global and update sub-primitives.

2. Phase 2

The second HLO phase performs optimizations on the low-level sub-primitives. It includes *Fusion, Fuse shapes, Fuse different types*, Parallel loop analysis, Fuse mask, Shift fusion, and Bool lowering.

Fusion is one of the most important performance optimizations in ArBB. Since each ArBB operator is data parallel, it inherently requires one or more loops or loop nests to iterate over each element in the vector operand, either sequentially or in parallel. We call such loops *implicit* loops. Fusion collects multiple operators into a group that can be implemented in a common loop nest. The main purposes of Fusion are to reduce memory usage and to reduce synchronization and parallelization overhead. By fusing several ArBB operators, we do not need to read and write all intermediate results from and to memory between the operators. Instead, we store them in the SIMD registers and pipe them through to the next operators in a chain. In addition, data parallel tasks tend to scale much better after fusion since the granularity of the task is coarser.

For now, fusable operators must not carry any dependences or require any synchronization between threads, for simplicity of implementation in parallel tasks. By default, we also fuse only operators on vectors with the same number of elements. We use the term 'fuse node' to refer to a list of fused ArBB operations. We implement shape analysis to collect the loops implied by the fused nodes, and optimize the loop order to achieve better data locality. Instead of using the data shape of multi-dimensional vectors, we analyze the shapes implied by the ArBB operators. For example, any element-wise operator has a 1D shape no matter how many dimensions its operands are, while `shift` operators in both row and column directions have a 2D shape. All the shapes in a fused node are aggregated and combined into one common shape, so that the parallelization, SIMDization and blocking can be decided and applied to each specific dimension of the common shape with some heuristics. Based on our own shape analysis, we are able to accomplish many traditional loop optimizations.

Figure 8 shows our fusion algorithm. **EU** represents a JIT evaluation unit which has a graph IR member that represents a set of ArBB operators. First, we initialize the fusion-related data structure for each node. The element size is checked for compatibility, the input or output fusion symbols that represent the implementation loop structure are initialized, and the fusion type (shape) is checked. Then, we recursively find the fusable code regions in a greedy fashion using a linear scan of the IR, resulting in a fusion map. We also maintain the fuse shape until we meet a control-flow barrier, which inhibits further fusion. Once the fusion maps are analyzed, we create a hierarchy node to group all of the fusable nodes inside it.

Figure 6(d) shows the result of Fusion applied to Figure 6(c). The 'add' and 'local' nodes can be fused together because they are both element-wise operators. However, 'global' cannot be included in the same fused node because it requires synchronization.

The *Fuse different types* transformation fuses operations with the same number of elements but different element type widths, as follows. Consider the case of fusing a mixture of single- and double-precision computation that is packed into SIMD registers. Fusion only occurs if the shapes of the collections are identical, which implies that they have the same total number of elements. And since all of the work is to be fused into a single a loop, there must be the same total number of iterations for both single and double precision. Since a double precision number is twice as wide as a single precision number, and there are half as many elements per SIMD register, the original double precision loop is unrolled by a factor of two and combined with the single precision loop.

Parallel loop analysis looks at the IR and marks loops without loop-carried dependences as parallelizable. This analysis results can help LLO determine the parallelization strategy of data parallelism or task parallelism later on. This implicit parallelism can be used to augment explicit parallelism from other expressions, such as `map` functions.

The Fuse mask optimization transforms a series of masked operators that have the same mask to unmasked operators, except for the final result. This minimizes the use of mask operators on micro-architectures that perform them poorly. It is especially useful for optimizing the IR after the Map Function Transformation, which introduces many mask operators to select whether each vector element participates in a vector operation and to emulate control flow.

Shift fusion optimizes a chain of `shift` operators by merging them with algebraic rules. For example, right-shift by one followed by two is the same as right-shift by three. Since `shift` operators inherently create dependences between parallel tasks, these `shift` chains cannot be fused together unless Shift fusion is enabled.

Bool lowering converts the bool type to an internal bool type with the same bit width as the operation where a bool is defined, because the ArBB bool type does not specify the bit width, in contrast to other ArBB data types. This optimization also minimizes the cast operators due to this disambiguation.

```
Procedure initFuseInfo(graph g)
  for each node n in g
    if n is a hierarchy node
      initFuseInfo(n->subgraph);
    else
      setElemSize(n);
      setFuseSymbol(n);
      setFuseType(n);

  for each node n in g
    if n is a controlFlowBarrier
      stopAllFusionResults(n);
    else if fusable(n, fuseShape)
      updateFuseShape(fuseShape, n);
      insertInFusionMap(fuseMap, n);

  for each fuse region r in fuseMap
    fnode = createFuseNode(r);
    setFuseSymbol(fnode, r->FuseSym);
    linkFuseNodeToGraph(fnode, g);

Procedure fuseGraphRecursive(graph g)
  for each node n in g
    if n is a hierarchy node
      fuseGraph(n->subgraph);

  collectFuseInfo(g);
  transformFusion(g);
  fuseGraphRecursive(g);

Procedure fuse (EU eu)
  initFuseInfo(eu->graph);
  fuseGraph(eu->graph);
```

Figure 8: High-Level Fusion Algorithm.

3. Phase 3

The third HLO phase mainly performs Memory optimization and Loop order analysis. In the first and second phase, HLO maintains the IR in the SSA form. Memory optimization allows different vectors to share the same memory space, as long as their live ranges do not overlap and their sizes are the same. This reduces total memory usage. However, as a result, the IR becomes a non-SSA form. Figure 6(e) shows the result of Memory Optimization. Collections <A> and <B> can share the same memory space, as can <C> and <res>.

Loop order analysis determines the optimal order among explicit loops such as `_for`, `_while` and `_do` and implicit loops in a fused node by analyzing data reuse opportunities in each loop to improve data locality.

B. LLO

LLO performs both parallelization and SIMDization. Its parallelization is implemented via function outlining. The outlined tasks each obtain their own portion of work after argument demarshalling, in the same way as the OpenMP implementation [Brunschen 2000]. Behind the scenes, the threading runtime (TRT) is invoked to spawn outlined tasks. There are two types of parallelism currently supported by ArBB: data parallelism and "parallel for"-type task parallelism. So we partition either the common shape of a fused node as described in Section IV.A.2, or we partition the iteration spaces. We only employ task parallelism if a `_for` encloses a fused node, not the other way around, since increasing the size of the parallel region better amortizes the parallelization overhead. The input and output arguments of a fused node inside a parallel `_for` loop become private to a task, if they are not live out of the `_for` construct. We use local storage to manage the task-private variables. Meanwhile, LLO selects the data partitioning policy with or without alignment consideration. Most operators require that each task receive aligned data after partitioning the vectors in order to access a portion of the vector which is still aligned. But this is not true for global reduce and scan, as each task only provides or receives one local scalar sum, so the partitioning policy needn't take alignment into account. LLO takes care of task granularity management and scalability issues. There are primarily two types of synchronization generated:

- The global phase of some operators, like `pack, reduce` and *`scan`* [ArBB 2011], requires a synchronization point to combine local results.
- The compiler supports fine-grained dependence tracking between tasks, enabling the use of synchronization only between dependent tasks, rather than requiring a barrier, where that is beneficial.

ArBB's SIMDization process includes implementation of efficient SIMD algorithms, residue handling, alignment handling, and balancing scalarization [Zhao 2005]. LLO has implemented efficient algorithms for each of local, global and update sub-primitives. Residue handling processes boundary elements of data sets that don't filled vector registers. There are two ways of handling residues: (1) using masks to keep results only from valid elements, and (2) generating scalarized code to handle each element separately inside a scalar loop.

We adopt the first approach in most cases, for the sake of software reuse. Alignment handling is required for SSE/SSE2 instructions. LLO maintains the invariant of aligning starting memory addresses for both input and output arguments for a fused node. As almost all the remaining data is promoted to registers, most operators do not have additional alignment requirements, except for permutation operations.

LLO optimizes `shift` operations since they are common in stencil applications. A `shift` operation usually reads from an unaligned address. In order to maximize the SIMDization benefit, we peel the SIMD loop with aligned accesses and leave the less important SIMD loops with unaligned accesses. If the shift distance of a shift operation is unknown at compile time, we are unable to fuse this `shift` with other `shift` operations because we don't know where to do loop peeling. But we can fuse `shift` with `_for` loops when the loop induction variable is used as the shift distance, since we can analyze the shift direction and know the maximum shift amount and therefore will be able to peel off the SIMD loop with aligned accesses.

V. RELATED WORK

The data-parallel programming model used by ArBB is similar to several other previous languages, beginning with APL [APL] and C* [Hillis 1986]. Although ArBB is an imperative language, the NESL functional data-parallel language inspired some of its key features, including segmented (nested) arrays [Blelloch 1990, 1993, 1996] (which are however not discussed at length in this paper).

The work described in [Chatterjee 1993] studied data parallelism based on their intermediate language called VCODE, a stack based IR similar to our work in many ways. Both systems support nested data parallel types and many data parallel operators including permutation and scan. Their intermediate representation is also a graph notation similar to ours. Our compiler analyses and transformations include most of their techniques such as size inference, access inference, cluster partitioning, and storage optimization. However, ArBB differs from [Chatterjee 1993] in several respects. Unlike VCODE in [Chatterjee 1993], ArBB borrows some features from functional languages like closures, but has imperative semantics and extends more mainstream languages like C/C++. ArBB can also fuse ArBB operators even in the presence of imperative control flows constructs like `_for` and _if. This allows us to thus further expand the granularity of fusion, which, as we show below, can have a large performance impact. ArBB employs task and data parallelization (threading and SIMDization) all in a unified framework. This allows us to choose between task and data parallelism or combine them, whichever is deemed more beneficial.

Nested data parallelism has also been supported in Haskell [Chakravarty 2001]. The heavy influence of pattern-based parallel programming has already been cited [Cole 1989, Gamma 1994, Siu 1996, Skillicorn 1998, Bosch 1998, Massingill 1999, Bromling 2002, Tan 2003, Cole 2004, MacDonald 2002, Mattson 2004, Aldinucci 2007, McCool 2010] but related work on the patterns used in typical parallel workloads at Berkeley was also influential [Asanovic 2006]. Predecessor systems to ArBB include RapidMind [McCool 2006], RapidMind's predecessor, Sh [McCool 2002, 2004a, 2004b] and Ct [Ghuloum 2007]. Recent parallel work for high productivity data-parallel languages using code specialization in Python share many similarities [Catanzaro 2010a, Catanzaro 2010b]. The support of user-directed code generation as a language feature was inspired by work in ML [Lee 1996] and of course by template metaprogramming [Veldhuizen 1999, Abrahams 2004] and generative metaprogramming [Herrington 2003].

VI. PERFORMANCE RESULTS

We demonstrate three things with performance results:

- The ArBB compiler is capable of generating high-quality code, and its performance compares favorably with other compilers.
- It is able to gain significant speedups by harvesting SIMD and thread parallelism.
- Selected optimizations that are of greatest interest to the ArBB programming model can show significant gains where they apply.

We demonstrate the first two items by showing speedups of ArBB code over C, at different ArBB compiler optimization levels, in Section VI.C. The impact of optimizations is shown in Section VI.D.

A. Workloads

We used the 17 workloads that are part of the ArBB beta release [ArBB 2011] for our analysis. A description of these workloads may be found in Table II. These workloads span a variety of different application domains, and were selected based on customer input and their suitability for harvesting data parallelism.

B. Methodology

The kernels in each sample are coded in both (scalar) C and in one or more expressions (kernels) of ArBB code. The duration of kernel execution is timed after a warm up run, such that compilation and cache warm up has already occurred. We offer small and big data sets in our Beta release; we used the big data sets here. We used one or two (see below) Nehalem sockets with 4 cores and 2 threads, running at 3.3 GHz with 8MB cache and 32GB of memory. We used RedHat EL6.0 Beta 1 for the switch comparisons and Windows 2008 SP2 for the overall speedup results. All workloads were compiled at 64 bits.

For speedup results, we take the minimum execution time across ten runs of the baseline, and compare that with the minimum time of the ten runs of the compared kernel. Other than a couple of outliers, the baseline runs were within 2%. All of the reported results are statistically significant. The run to run variation was found to be minimized when tasks are over-decomposed so as to have 4 tasks per hardware thread. That was the configuration used for the switch setting comparisons. The overall speedup results were in a more standard 2 socket, 4 core, 2 thread, one task per thread configuration.

TABLE II: WORKLOADS IN ARBB BETA 3 RELEASE SAMPLES

Finance	binomial-tree	Numerical lattice for pricing European options
	black-scholes	Analytical method for pricing European options. Optionally evaluates or approximates polynomials.
	monte-carlo	Stochastic method for computing financial options using the Black Scholes formula given randomly varying prices. Can optionally generate the sequence of random numbers using a multiplicative congruential generator (MCG).
	poisson-solver	Monte-Carlo method to solve Poisson functions (MCP solver). Uses a sequence of random numbers from a linear congruential generator (LCG).
Graphics	raytracing 1	A kernel to create a realistic visualization of a scene when tracing rays from a camera through an image plane to a light source. For each pixel in a 2D array, the kernel determines the closest ray-triangle intersection and evaluates the pixel shade using a lighting calculation.
	raytracing 2	A variation on raytracing1 where ray triangle intersection is limited to triangles in grid cells that intersect with rays. In other words, a uniform spatial partition is used for acceleration.
	mandelbrot	Fractal data set generation with a quadratic polynomial map
Image proc	convolve	Convolution of a 2D image with a discrete Gaussian function.
	gauss-convolve	Similar to convolve. Uses different stencil sizes and does not assume odd stencil sizes.
	sobel	An edge detection filter for a 2D image that uses the gradient (rate of change) of image intensities.
Medical	3D-dilate	A morphological operator for dilation applied to 3D grayscale images.
	3D-erode	A morphological operator for erosion applied to 3D grayscale images. Similar to 3D-dilate, except that the smallest difference is output.
	3D-gauss-convolve	Convolution of a 3D image with a discrete Gaussian function.
	back-projection	A technique for image reconstruction used with inputs from computed axial tomography (CAT) scans.
Seismic	3dstencil	Convolution used in reverse time migration (RTM).
	convolution	1D and 2D convolution for a seismic image.
	kirchhoff	Generic Kirchhoff migration assuming constant velocity of seismic waves through a sub-surface.

For speedup results, we take the minimum execution time across ten runs of the baseline, and compare that with the minimum time of the ten runs of the compared kernel. Other than a couple of outliers, the baseline runs were within 2%. All of the reported results are statistically significant. The run to run variation was found to be minimized when tasks are over-decomposed so as to have 4 tasks per hardware thread. That was the configuration used for the switch setting comparisons. The overall speedup results were in a more standard 2 socket, 4 core, 2 thread, one task per thread configuration.

The scalar compilers used were Microsoft® Visual C++ Compiler 2008 (**VCC**), the Intel C/C++ compiler version 12.0 0 (**ICC**) on and gcc 4.4.3 (**GCC**). The performance-related compiler switch settings correspond to those used in our development environment for reasonable build time and good performance: O2, -msse2, -fno-stack-protector (no overrun protection), -fno-strict-aliasing (disable use of ANSI aliasing rules) and -fp-speculation=fast (allow speculation on floating point operations).

C.Overall Performance

The ArBB compiler has its own optimization levels, ARBB_OPT_LEVEL=O2 and O3. ArBB's **O2** applies all optimizations, including SIMDization, but runs on only one thread. **O3** adds thread parallelization to O2. Table III shows the geomeans of speedups of ArBB relative to other compilers' generated code that runs on one thread. No special effort was made to make the C versions of the kernels more vectorizable. Figure 9 shows speedup results (on a log scale) for ArBB's O2 and O3, relative to single-threaded C versions that were compiled by each of the ICC and VCC compilers on Windows. Linux results are similar. Since some workloads have more than one kernel, the kernel version with the best ArBB implementation (as measured by speedup) was used. The kernel is labeled with k1, k2, etc. to identify which kernel within the workload was selected. Workloads are arranged in Figure 9 in order of increasing speedup relative to the most pervasive baseline, VCC. This arrangement is commonly called an S-curve.

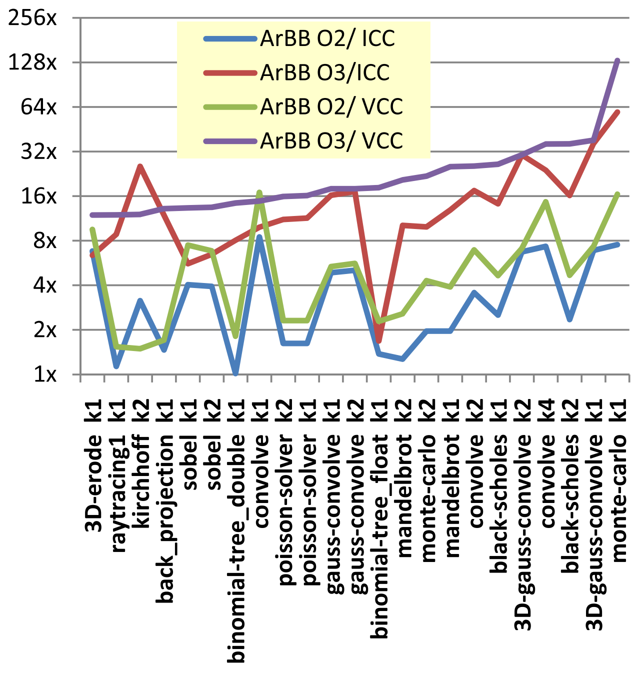

Figure 9: Speedup relative to Intel and Microsoft C/C++ compilers, at ArBB opt. levels O2 (SIMD) and O3 (SIMD+threading), on a $\log_2$ scale. The S-curve is sorted by increasing O3 speedup vs. VCC.

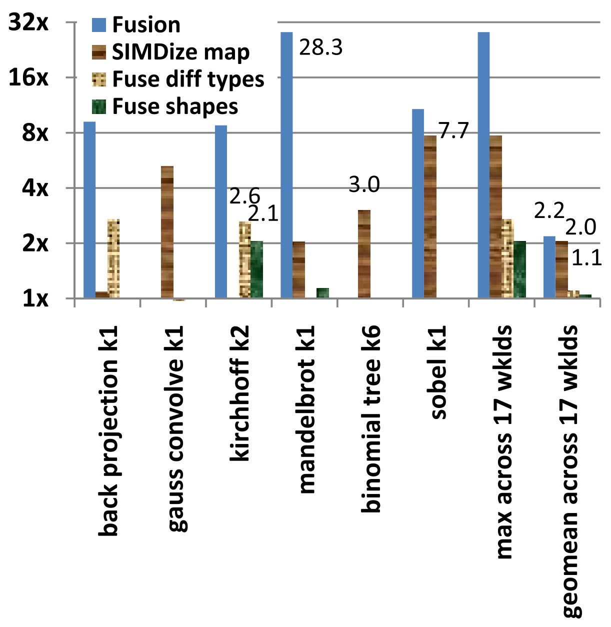

Figure 10: Speedup impact of fusion-related optimizations on selected workloads, and on the whole set of Beta release samples.

Table III: Geomean and max speedups for ArBB Optimization Level 2 (O2) and 3 (O3, threading), over the C baseline with the specified compiler on 1 thread.

		Windows		Linux	
		ICC	VCC	ICC	GCC
Geomean	O2	1.4x	2.0x	1.7x	2.1x
	O3	4.3x	6.1x	4.4x	5.8x
Maximum	O2	8.5x	16.9x	9.9x	10.9x
	O3	59.2x	131.9x	38.8x	54.9x

The geomean of the speedups, listed in Table III, are well above 1 for O2 for all compilers (1.4x-2.1x), suggesting that the ArBB compiler is making better use of the SIMD hardware than the other compilers. As you can see from the S-curve in Figure 9 all speedups are greater than 1. The O3 speedup (8.5x-16.9x) is often linear, when not memory bound, showing good use of thread parallelism. Notice, for example, the excellent scaling for kirchhoff, which does a random gather, and convolve, which has conditionals and neighbor accesses in a nested loop. Some optimizations are still being tuned in this beta release, e.g. to reduce the number of vector copies to ease the memory bottleneck and improve thread scaling. Superlinear speedups are obtained by reducing memory traffic and reducing control overhead, e.g. from Fusion.

D. Optimization Impact

As discussed in Section IV, several of the ArBB compiler optimizations are of special interest, given the language and architecture of the compiler. In this section, we show the impact of turning each of a selected subset of optimizations off, and measure the speedup impact on performance (optimization-off time/optimization-on time). The results are shown in Figure 10 for a selected subset that we discuss in greater detail, and geomean and max for the overall set of samples in the Beta release is shown to the right. To aid the discussion, we present the source code for a subset of these examples: mandelbrot's first kernel (k1) in Figure 3, kirchhoff k2 in Figure 11, and sobel k2 in Figure 12.

The *Fusion* optimization has a speedup impact of 28.3x for mandelbrot k1 whose ArBB code is as shown in Figure 3. Fusing all of the operators allows the temporary variables such as *done*, *dest*, and *z*z* to be promoted to vector registers instead of saving them into local memory. Fusing the `_for` loop has a large performance impact because then all of the output vectors that are live out can share the same memory handle without allocating and deallocating memory in every iteration of the `_for` loop. An array of structures (AoS) to structure of arrays (SoA) transformation is applied to the real and imaginary parts of std::complex to increase locality.

Figure 11 shows the kirchhoff code example. We are able to fuse not only the normal ArBB operators but also the entire `_for` loop. *Fuse different types* accommodates the mixture of f32 and usize (unsigned 32- or 64-bit, depending on the target) types in the fused code region to attain the 2.6x performance improvement shown in Figure 10. In addition, this kernel includes computations on two different shapes, 1D and 2D. Fuse shapes will allow fusion of shape changing operators in kirchhoff and thus enables the fusion of the entire loop. This is reflected in the 2.1x performance impact in Figure 10. Otherwise, there are two different fusion code regions, one for each shape of code region.

```
_for (jcx = (uncaptured<usize>::type)0,
      jcx < NX_LEN, jcx ++) {
   f32 cx = f32(jcx) * dx;
   dense<usize,2> vIt  =  (dense<usize,2>)(sqrt((vX
- cx) * (vX - cx) + vZ * vZ) * reciVdt + 0.5f);
   dense<usize,2> vIu = fill(jcx, nx, nt);
   dense< array<usize, 2>, 2> vIdx;
   vIdx.set<0>(vIt);
   vIdx.set<1>(vIu);
   vModl += vData[vIdx];
} _end_for;
```

Figure 11: Code example for ***kirchhoff*** *seismic reconstruction.*

The code example of sobel is depicted in Figure 12. The vectorized implementation of sobel k2 boasts a 7.7X speedup over the scalar implementation. The vectorized implementation of the `map` function embraces efficient 2D vector operators. For example, the `neighbor` operator just loads one scalar element from memory, and may be less optimized than the vector operations. Thus, the ArBB compiler internally implements the `neighbor` operator as a `shift2D` operation, which is SIMDized and optimized efficiently, as discussed in Section IV.B. On the contrary, the naïve scalarized implementation of the sobel kernel introduces two nested loops and leaves scalar operations on each vector element as is.

E. Use of loop analysis in the code generator

Since ArBB has a variety of target platforms, it supports code generation for all flavors of SSE, AVX and the ISA for the MIC Architecture. Part of the metadata information that HLO and LLO pass to CCG is an indication of what the loops

```
f32 sobelX2(f32& src)
{
  // Uses relative indices to gather pixels
  // in the 3 x 3 box around the pixel 'src'
  return neighbor(src, -1, -1) + f32(2.0f) *
    neighbor(src, 0, -1) + neighbor(src, 1, -1) +
    f32(-1.0f) * neighbor(src, -1, 1) +
    f32(-2.0f) * neighbor(src, 0, 1) +
    f32(-1.0f) * neighbor(src, 1, 1);
}

f32 sobelY2(f32& src)
{
  // Uses relative indices to gather pixels in
  // the 3 x 3 box around the pixel 'src'
  return neighbor(src, -1, -1) * f32(-1.0f) +
     neighbor(src, 1, -1) +
     neighbor(src, -1, 0) * f32(-2.0f) +
     neighbor(src, 1, 0)  * f32(2.0f) +
     neighbor(src, -1, 1) * f32(-1.0f) +
     neighbor(src, 1,  1);
}

template <typename T>
void sobel3x3Mp(f32 src, T& res)
{
  assert(typeid(typename  uncaptured<T>::type)  !=
typeid(double));

  f32 x = sobelX2(src);
  f32 y = sobelY2(src);

  // Uses a ternary 'select' statement to choose
  // the larger of the derivatives in X and Y
  f32 voxel = select(abs(x) > abs(y), x, y);

  // Threshold the result to [0, max<T>()]
  typedef typename uncaptured<T>::type scalar_t;
  voxel = max(f32(0.0f), voxel);
  voxel = min(voxel,
    f32(std::numeric_limits<scalar_t>::max()));
  res = voxel;
}
```

Figure 12: Code example for ***sobel*** *edge detection filter.*

are, whether they are SIMDized, and whether they are hot loops. The number of iterations may be resolved dynamically, such that ArBB, as a dynamic compiler, can sometimes provide more accurate information than a static compiler could. The metadata information is used to determine the profitability of breaking SIMD or scalar register live ranges that span hot loops, but are not used in them. The availability of that information had a geomean impact of 6.8% across all of the workloads, with a max impact of over 60% for the two binomial tree kernels.

VII. Summary and Conclusions

The Array Building Blocks compiler makes use of a language embedded within C++ (and potentially other languages) to specify data and some task parallelism in a natural and relatively productive way. By letting the programmer specify *what* to do rather than *how* to do it, the total cost of ownership from development, debugging, porting and maintaining code is reduced. The paper offers the following contributions:

- A unified framework for harvesting thread and vector (SIMD) parallelism
- A higher level of abstraction for expressing semantics that enables the compiler to span unified and disjoint and even remote memory models, uniprocessors up to many-core architectures, SIMD and vector ISAs of various widths, without requiring over-specification and detailed micro-architectural knowledge
- A novel dynamic compiler architecture that enables retargetability, future-proofing, and dynamic inlining that helps enable modularity through lower function nesting costs
- A new capture mechanism to explicitly manage context-sensitive compilation, for powerful specialization, generic programming, and deterministic JIT compilation performance
- Synchronization that is implicit, but well defined, upon access to data.
- Safety that is enforced by using a logically-separate data space, such that a whole class of parallel programming bugs from data race conditions are precluded
- Code generation optimizations that are either original or that are applied in a new way to this language and target architecture context, particularly with respect to fusion
- Programming model based on composing structured and deterministic patterns for parallel computation. Built-in support for common application patterns, like stencil.

The performance of ArBB is competitive with production compilers. It makes good use of SIMD instructions and cores, with vector and thread parallelism, with a geomean speedup on a single core in the range of 1.4-2.1x, and a geomean speedup on 8 dual-threaded cores in the range of 4.3-6.1. Speedups of over 100x have been observed on just 8 logical CPUs and 4 SIMD lanes.

The ArBB compiler makes use of optimizations that are either new or applied in a novel way. A subset of these optimizations is illustrated by explaining how they are applied to code examples. The performance on those code examples and our overall suite of samples in our beta release is shown. Speedups of nearly 30x can occur from fusion, indicating that it is a primary source of parallelization performance, while many others, like enabling SIMD in elemental functions, fusing shape-related operators, and fusion of operations on different data types can far more than double performance.

VIII. Future Work

Some topics of interest that are beyond the scope of this paper include detailed discussions of the virtual machine interface, how offloading of computation to the MIC[Skaugen 2010] architecture works and performs, and how ArBB can be extended to operate on clusters.

ArBB is currently a product in beta form, and much more performance optimization head room remains. Memory bandwidth appears to limit performance in many cases, and we're investigating how to reduce memory traffic to gain scalability.

References

[Abrahams 2004] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming*, Addison-Wesley, 2004.

[Aldinucci 2007] M. Aldinucci and M. Danelutto, *Skeleton-based parallel programming: Functional and parallel semantics in a single shot*, Comput. Lang. Syst. Struct., 33(3-4), 2007, pp. 179-192.

[Allen 2002] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures.,* Morgan Kaufmann Publishers, San Francisco, 2002.

[APL] APL: A Programming Language. URL: http://www.users.cloud9.net/~bradmcc/APL.html

[ArBB 2011] Intel® Array Building Blocks Documentation, URL : http://software.intel.com/en-us/articles/intel-array-building-blocks-documentation/, or http://software.intel.com/en-us/articles/intel-array-building-blocks/, 2010

[Asanovic 2006] K. Asanovic et al, *The Landscape of Parallel Computing Research: A View from Berkeley*, EECS Department, University of California, Berkeley EECS-2006-183, 2006.

[Blelloch 1990] G. E. Blelloch, *Vector models for data-parallel computing*, MIT Press, 1990.

[Blelloch 1993] G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein and M. Zagha, *Implementation of a portable nested data-parallel language*, PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming, 1993, pp. 102-111

[Blelloch 1996] G. E. Blelloch, *Programming parallel algorithms*, Commun. ACM, 39(3), 1996, pp. 85-97.

[Bosch 1998] J. Bosch. *Design patterns as language constructs*. Journal of Object-Oriented Programming, 11(2):18-32, 1998.

[Bromling 2002] S. Bromling, S. MacDonald, J. Anvik, J. Schaefer, D. Szafron, K. Tan, *Pattern-based parallel programming*, Proceedings of the International Conference on Parallel Programming (ICPP'2002), August 2002, Vancouver Canada, pp. 257-265.

[Brunschen 2000] Brunschen, C., Brorsson, M.: OdinMP/CCp - a portable implementation of OpenMP for C. Concurrency - Practice and Experience 12 (2000) 1193–1203

[Buck 2007] I. Buck, *GPU Computing: Programming a Massively Parallel Processor*, Proceedings of the International Symposium on Code Generation and Optimization, March 11-14, 2007.

[Catanzaro 2010a] Catanzaro, B. , Garland, M. and Keutzer, K. *Copperhead: Compiling an Embedded Data Parallel Language.* Technical Report UCB/EECS-2010-124, EECS Department, University of California, Berkeley, September 16, 2010.

[Catanzaro 2010b] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovi´c, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. *SEJITS: Getting Productivity and Performance With Selective Embedded JIT Specialization.* Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, 2010.

[Chakravarty 2001] Manuel M. T. Chakravarty and Gabriele Keller and Roman Lechtchinsky and Wolf Pfannenstiel. *Nepal --- Nested Data Parallelism in Haskell*. Proc. 7th International Euro-Par Conference, volume 2150 of Lecture Notes in Computer Science, pp 524-534. Springer-Verlag, 2001.

[Chatterjee 1993] *Compiling nested data-parallel programs for shared-memory multiprocessors.* ACM Trans. Program. Lang. Syst. 15(3), July 1993, pp 400-462.

[Clyther 2010] *Clyther: Python language extension for OpenCL.* URL: http://clyther.sourceforge.net/, 2010.

[Cole 1989] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman/MIT Press, 1989.

[Cole 2004] M. Cole. *Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming,* Parallel Computing, 30(3), pp. 389-406, March 2004

[Czarnecki 2000] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, 2000, ACM Press/Addison-Wesley.

[Du Toit 2010] S. Du Toit, A Data-parallel Virtual Machine, URL: http://software.intel.com/en-us/articles/data-parallel-vm/, 2010

[Gamma 1994] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Ghuloum 2007] A. Ghuloum, E. Sprangle, J. Fang, G.Wu, and X. Zhou. *Ct: A Flexible Parallel Programming Model for Tera-scale Architectures.* Technical Report White Paper, Intel Corporation, 2007.

[Herrington 2003] J. Herrington, *Code Generation in Action*, 2003, Manning Publications.

[Hillis 1986] W. D. Hillis and J. Guy L. Steele. Data *parallel algorithms*. Communications of the ACM, 29(12):1170–1183, 1986.

[Lee 1996] P. Lee and M. Leone, *Optimizing ML with run-time code generation*, Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation, p.137-148, May 1996.

[MacDonald 2002] S. MacDonald, J. Anvik, S. Bromling, D. Szafron, J. Schaeffer and K. Tan. *From patterns to frameworks to parallel programs*, Parallel Computing, 28(12);1663-1683, 2002.

[Massingill 1999] M. Massingill, T. Mattson, and B. Sanders. *A pattern language for parallel application programs*. Technical Report CISE TR 99-022, University of Florida, 1999.

[Mattson 2004] T. Mattson, B. Sanders, B. Massingill, *Patterns for Parallel Programming*, 2004, Pearson Education.

[McCool 2002] M. McCool, Z. Qin, and T. Popa, *Shader metaprogramming*, HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2002, pp. 57-68.

[McCool 2004a] M. McCool, S. Du Toit, T. Popa, B. Chan and K. Moule, *Shader algebra*, ACM Trans. Graph., 23(3), 2004, pp. 787-795.

[McCool 2004b] M. McCool and S. Du Toit, *Metaprogramming GPUs with Sh*, 2004, AK Peters.

[McCool 2006] M. McCool. Data-Parallel Programming on the Cell BE and the GPU Using the RapidMind Development Platform. GSPx Multicore Applications Conference, 9 pages, 2006.

[McCool 2010] M. McCool. *Structured Parallel Programming with Deterministic Patterns*, HotPar 2010 (2nd USENIX Workshop on Hot Topics in Parallelism), Berkeley, CA, 14-15 June 2010.

[Munshi 2008] A. Munshi, *OpenCL Specification Version 1.0*, The Khronos Group, 2008.

[Nickolls 2008] J. Nickolls, I. Buck, M. Garland, and K. Skadron. *Scalable parallel programming with CUDA.* Queue, 6(2):40–53, Mar/Apr 2008.

[Nuzman 2011] D. Nuzman, I Rosen, S. Dyshel, A. Zaks, E. Rohou, K. Williams, A. Cohen, and D. Yuste, *Auto-Vectorize Once, Run Everywhere,* Proceedings of the International Symposium on Code Generation and Optimization, 2011

[Owens 2005] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krueger, A. E. Lefohn, and T. J. Purcell, *A survey of general-purpose computation on graphics hardware*, Eurographics 2005, State of Art Report.

[Siu 1996] S. Siu, M. De Simone, D. Goswami, and A. Singh. *Design patterns for parallel programming*. Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96), pp. 230–240, 1996.

[Skaugen 2010] K. Skaugen, *HPC Technology - Scale Up and Scale Out*, Keynote, International Supercomputing Conference, 2010

[Skillicorn 1998] D. B. Skillicorn and D. Talia, *Models and languages for parallel computation*, ACM Comput. Surv., 30(2), 1998, pp.123-169.

[Tan 2003] K. Tan, D. Szafron, J. Schaeffer, J. Anvik and S. MacDonald, *Using generative design patterns to generate parallel code for a distributed memory environment*, PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, 2003, pp 203-215.

[Veldhuizen 1999] T. L. Veldhuizen, *C++ templates as partial evaluation*, In ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, 1999.

[Zhao 2005] Yuan Zhao, Ken Kennedy. *Scalarization on Short Vector Machines*. Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005. Pages: 187-196. ISBN:0-7803-8965-4