Abstract

Our ability to create systems with many cores 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 programs that operate on applications with ample data parallelism is essential to enabling the use of multi- and many-core processor architectures. However, those architectures and ISAs are also shifting and diversifying quickly, making it hard for a single binary to run best on all possible targets. Because of this, retargetability and dynamic compilation are of growing relevance.

This paper introduces 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 our forthcoming beta release product, 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.

1. Introduction

Parallelism, while only gaining significant attention in recent years, 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 diminishing returns, limited by power constraints and design complexity.

The recent attention directed towards parallelism, then, is the result of a major shift in responsibility from processor architects to software developers. This trend is manifest in several ways:

- The enrichment and widening of vector instructions, in which SIMD parallelism is explicit (SSE, AVX, Intel® Many-Integrated Core (MIC) Architecture [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-exposed and managed parallelism to effectively utilize the hardware. 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 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 new 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.

Moreover, 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 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 in reducing the cost of development over time. In order to satisfy software engineering practices, a programming model should support modularity without giving up performance. General parallelism is hard, but data parallelism often scales better than task parallelism. An abstract, structured form of data parallelism can avoid issues with race conditions and deadlock by design, enhancing productivity.

ArBB achieves these goals by combining several key technologies. First, it supports generic programming based on
dynamic code generation. This allows the code generation to adapt to the target architecture, while still allowing the programming model to be abstract. Second, the programming model itself is based on a structured form of data parallelism that avoids, by construction, common sources of error. Third, optimizations are included in ArBB that allow computations specified far apart in the source code to be fused together into arithmetically-intense 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[ArBB2010], 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 out of Intel Labs and the commercial RapidMind platform spun out of 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 serial ordering, simplifying 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 is 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.

2. 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 (integers, 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 processors. 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], itself with 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. Large amounts of latent parallelism can be expressed by use of both sequences of vector operations and application of elemental functions to collections. Vector operations act directly on collections, while elemental functions can be mapped over all the elements of a collection.

The VM is responsible for mapping this abstract, latent parallelism onto actual parallel mechanisms in the physical machine. Currently, ArBB supports both SIMD instruction and core parallelism. However, 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 more cores if available. Special features of these instruction sets, such as scatter/gather, are also supported. In addition, the VM supports an abstract model of memory that can transparently support remote memory, such as that in attached accelerator cards (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 “design out” 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 processors with radically different underlying architectures.

The details of the ArBB C++ interface are best presented using an example. Figures 1 through 3 give two implementations of the Mandelbrot set computation. The first implementation is in terms of elemental functions; the second is in terms 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. Figure 2 shows the code for setting up and invoking this elemental function.

```cpp
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++.

void mandelbrot1(const dense<i32, 2>& D, const std::complex<f32>& C) {
    // array output
    dense<f32, 2>&& dest,
    // array input
    dense<
        std::complex<f32>, 2>
    C
    // array input
} { map(mandel)(D, C);
}
void mandelbrot1(const
dense<i32, 2>&& dest,
const

    dense<f32, 2>& sR,
        // row scale (input)
    dense<f32, 2>& 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>
    tspcale;
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>(0.0f, 0.0f),
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 + tspcale+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 2: Wrapper code to invoke elemental function implementation of 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 range 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 is equivalent, so the software developer can choose whichever approach is more convenient.

```cpp
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>
    tspcale;
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>(0.0f, 0.0f),
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 + tspcale+offset, z);
    } _end_for;
}
```

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 construct and manipulate code objects, which we call "closures," explicitly. The "call" operation combines several steps: the first time it is called with a particular function pointer, it invokes it, but only to capture the sequence of operations over ArBB types. It 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, however, it uses the cached machine language.

```cpp
typedef void F(dense<i32, 2>&, dense<f32>, dense<f32>);
max_count = 100;
closure<F> mandelA = capture(mandelbrot2);
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 fully type safe.
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 can be invoked. 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 many 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.

3. 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, and shader languages. Similarly, 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. 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 compile the embedded language functions like call, map, and bind down to calls to the ArBB dynamic library entry points. When these functions are invoked (or capture is called), the ArBB library follows the control flow specified by the C++ code, and interprets just the ArBB core to build IR within the context at that time. 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 interpretation. 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.

![ compilers](image_url)

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.

4. Code Optimizations

The ArBB compiler optimizes the intermediate representation (IR) to improve the 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.

4.1 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 6.4 and 6.5 are shown in italics. 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).

4.1.1 Phase 1

The first HLO phase performs the unique Map-Function transformation and Sub-Primitive Decomposition, as well as classical compiler optimizations such as Static Single Assignment form (SSA) transformation, copy propagation, dead code elimination, and constant folding[Allen2002]. HLO performs these optimizations repeatedly as we transform the IR. The SIMDize Map transformation transforms a map function written in a scalar form to a vectorized call function in order to exploit data parallelism. 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. Once a high-level primitive is decomposed into sub-primitives, it does not retain high-level semantics any more. Figure 6(c) is the result of Sub-primitive Decomposition. ‘add’ is an element-wise operator and it is only decomposed into its local sub-primitive, while ‘addScan’ is decomposed into local, global and update sub-primitives.

4.1.2 Phase 2

The second HLO phase includes optimizations on the low-level primitives. It includes Parallel Loop Analysis, Fuse Mask, Fusion, Fuse Shape Ops, Fuse Different Types, Shift Fusion, and Bool Lowering.

Parallel Loop Analysis looks into the IR and marks loops without loop-carried dependences as parallelizable. This analysis results can help LLO decide the parallelization strategy of data parallelism or task parallelism later on.

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.

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. For now, fusible operators must not carry any dependences or require any synchronization between threads for simplicity of implementation in parallel tasks. By default, we fuse only operators on vectors with the same number of elements. We use the term ‘fused 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, element-wise operators have 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. Fuse Shape Ops fuses shape-related operators into fusion nodes.

The Fuse Different Types transformation fuses operations with the same number of elements but different element type widths, thanks to LLO’s support of loop unrolling for operations with a wider type. As a result, the granularity of parallelism is coarser and the threading overhead is reduced. More importantly, the result of one operator can be directly fed into the following operator without writing the result in memory. Figure 6(d) shows the result of Fusion applied to Figure 6(c). ‘add’ and ‘local’ can be fused together because they are both element-wise operators. ‘global’ cannot be included in the same fused node because it requires synchronization.

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 carry 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, which is different than other ArBB data types. It also tries to minimize the cast operators due to this disambiguation.

4.1.3 Phase 3

The third HLO phase mainly does 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. As a result, the IR becomes a non-SSA form, and it reduces the total memory usage. Figure 6(e) shows the result of Memory Optimization. <A> and <B> can share the same memory space, and <C> and <res> can, too. 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.

4.2 LLO

LLO performs parallelization and SIMDization. Its parallelization is implemented via function outlining and the outlined tasks each obtain their own portion of work after argument demarshalling, like the OpenMP implementation [Brunschchen2000]. Behind the scenes, the threading runtime (TRT) is invoked to spawn these outlined tasks. There are two types of parallelism that ArBB supports: data parallelism and "parallel for"-type task parallelism. So we partition either the common shape of a fused node as described in Section 4.1.2, or 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 not alive out of the _for. We use the 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 the 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 shouldn’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 [ArBB2010], is designed to be a point of the synchronization to combine the local results.
- The compiler supports fine-grained dependence tracking between tasks, enabling the use of synchronization between dependent tasks, rather than requiring a barrier, where that’s profitable.

ArBB’s SIMDization includes implementing 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 the last few elements of the user’s data set that can’t be filled into a vector register. There are two ways of handling residues: (1) using the masks to keep results only from valid elements; (2) generating the scalarized code to handle each instance repeatedly inside a scalar loop. We adopt the first approach in most cases, for 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 are 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, 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 be able to peel off the SIMD loop with aligned accesses.

5. 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 [Blloch 1990, 1993, 1996] (which are however not discussed at length in this paper). Nested data parallelism has also been supported in Haskell [Chakravartly 2001].

ArBB and [Chatterjee93] both support data parallelism and have the notation of flat and nested data types and the corresponding element-wise, permutation, reductions, scan operations for those data types, and even aggregate operations to form a bigger parallel loop executing on shared memory. But ArBB differs from [Chatterjee93] in several respects. The latter’s VCODE is based on LISP. ArBB borrows some features like map from functional languages, but is embedded in more mainstream languages like C++. It offers a richer syntax for operations on multi-dimensional arrays which are widely used by existing customers. [Chatterjee93] provides stronger nested operation support than ArBB. ArBB can fuse operations with control-flow statements like _for and _if, which [Chatterjee93] disallows for its loop fusion. ArBB employs task parallelism via parallelizing _for, employs threading, and performs SIMDization for SSE/SSE2, all in a unified framework that calls for more sophistication than the approach of [Chatterjee93].

patterns used in typical parallel workloads at Berkeley was also influential [Asanovic 2006]. Predecessor systems to ArBB include both RapidMind [Mccool 2006] and RapidMind’s predecessor, Sh [Mccool 2002, 2004a, 2004b] and Ct [Ghoulom 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].

6. Performance Results

There are three things that we demonstrate with performance results:

- The ArBB compiler is capable of generating high-quality code, and its performance compares favorably with other compilers.
- -

<table>
<thead>
<tr>
<th>Workload</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sinusoidal-tree</td>
<td>Numerical lattice for pricing European options</td>
</tr>
<tr>
<td>black-scholes</td>
<td>Analytical method for pricing European options. Optionally evaluates or approximates polynomials.</td>
</tr>
<tr>
<td>monte-carlo</td>
<td>Stochastic method for computing financial options using the Blacksholes formula given randomly varying prices. Can optionally generate the sequence of random numbers using a multiplicative congruential generator (MCG).</td>
</tr>
<tr>
<td>poisson-solver</td>
<td>Monte-Carlo method to solve Poisson functions (MCP solver). Uses a sequence of random numbers from a linear congruential generator (LCG).</td>
</tr>
<tr>
<td>raytracing 1</td>
<td>A kernel used 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.</td>
</tr>
<tr>
<td>raytracing 2</td>
<td>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.</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>Fractal data set generation with a quadratic polynomial map</td>
</tr>
<tr>
<td>convolve</td>
<td>Convolution of a 2D image with a discrete Gaussian function.</td>
</tr>
<tr>
<td>gauss-convolve</td>
<td>Similar to convolve. Uses different stencil sizes and does not assume odd stencil sizes.</td>
</tr>
<tr>
<td>sobel</td>
<td>An edge detection filter for a 2D image that uses the gradient (rate of change) of image intensities.</td>
</tr>
<tr>
<td>3D-dilate</td>
<td>A morphological operator for dilation applied to 3D grayscale images.</td>
</tr>
<tr>
<td>3D-erode</td>
<td>A morphological operator for erosion applied to 3D grayscale images. Similar to 3D-dilate, except that the smallest difference is output.</td>
</tr>
<tr>
<td>3D-gauss-convolve</td>
<td>Convolution of a 3D image with a discrete Gaussian function.</td>
</tr>
<tr>
<td>back-projection</td>
<td>A technique for image reconstruction used with inputs from computed axial tomography (CAT) scans.</td>
</tr>
<tr>
<td>3d-stencil</td>
<td>Convolution used in reverse time migration (RTM).</td>
</tr>
<tr>
<td>convolution</td>
<td>1D and 2D convolution for a seismic image.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methodology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>- It is able to gain significant speedups by harvesting SIMD and thread parallelism.</td>
<td></td>
</tr>
<tr>
<td>- Selected optimizations that are of greatest interest to the ArBB programming model can show significant gains where they apply.</td>
<td></td>
</tr>
</tbody>
</table>

We demonstrate the first two items by showing speedups of ArBB code over C, at different ArBB compiler optimization levels, in subsection 6.3. The impact of optimizations is shown in subsection 6.4.

6.1 Workloads

We used the workloads that are part of the ArBB beta release [ArBB2010] for our analysis. A description of these workloads may be found in Table 1. These workloads span a variety of different application domains, and were selected based on customer input and their suitability for harvesting data parallelism.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kirchhoff</td>
<td>Generic Kirchhoff migration assuming constant velocity of seismic waves through a sub-surface.</td>
</tr>
</tbody>
</table>

6.2 Methodology

The kernels in each sample are coded in both C and in one or more expressions 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 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 hyper-threading is disabled (4 single-threaded cores), and tasks are over-decomposed so as to have 8 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++ 2008 (VCC), the Intel C/C++ compiler version 11.1 072 (ICC) on and gcc 4.4.3 (GCC). The compiler switch settings correspond to those used in our development environment for reasonable build time and good performance: O2, Ob0 (no inlining), -msse2, and Os (compile for size). We are still investigating what the best-possible switches are for the baseline, and we plan to use those in the final paper version.

6.3 Overall Performance

The ArBB’s compiler has its own O2 and O3 optimization levels. Its use of threading is enabled with the environment variable ARBB_OPT_LEVEL=O3. Most other optimizations, including SIMDization, are enabled at O2. The speedup of each of the workloads in the samples that are part of the ArBB
beta are shown in Figure 7. The figure shows the data for VCC and ICC on Windows. ICC, and GCC results on Linux are similar. No special effort was made to make the C versions of the kernels more vectorizable. The geometric mean of the speedups, listed in Table 2, are over 3 for O2, suggesting good use of the SIMD hardware. As you can see from the S-curve in Figure 7, the O2 speedup is positive except for one benchmark. The O3 speedup is often linear, when not memory bound, showing good use of thread parallelism. Notice, for example, the great 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.

![Figure 7: Speedup relative to Intel and Microsoft C/C++ compilers, at ArBB optimization levels O2 (SIMD) and O3 (SIMD+threading), on a log2 scale. The S-curve is sorted by increasing ICC O3 speedup.](image)

<table>
<thead>
<tr>
<th>Workloads</th>
<th>Windows</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ICC O2</td>
<td>O3</td>
</tr>
<tr>
<td>Opt Prices</td>
<td>3.2</td>
<td>11.8</td>
</tr>
<tr>
<td>Geomean</td>
<td>86</td>
<td>937</td>
</tr>
<tr>
<td>Max</td>
<td>503</td>
<td>503</td>
</tr>
</tbody>
</table>

Table 2: Geomean and max speedups for ArBB O2 and O3, over the C baseline with the specified scalar compiler

6.4 Optimization Impact

As discussed in section 4, 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 % impact on performance (of optimization off time/base time)-1). The results are shown in Figure 8 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.
template<typename T>
void mapBinomialTree(T risk_free_rate,  
  T stock_price, T exe_price,  
  T volatility, T deltat, T& opt_price)  
{
  T deltat_n = T(deltat) / T(OPT_TIMESTEPS);  
  T u = exp(volatility * sqrt(deltat_n));  
  T d = exp(-volatility * sqrt(deltat_n));  
  T a = exp(risk_free_rate * deltat_n);  
  T multiplier =  
    exp(-risk_free_rate * deltat_n);  
  T p = (a - d) / (u - d);
  dense<T> opt_prices =  
    fill(T(0.0f), MAX_NTIMESTEPS);  
  dense<T> upow_tbl =  
    fill(T(0.0f), MAX_NTIMESTEPS);  
  dense<T> dpow_tbl =  
    fill(T(0.0f), MAX_NTIMESTEPS);
  upow_tbl =  
    replace(upow_tbl, u32(0), T(stock_price));
  dpow_tbl =  
    replace(dpow_tbl, u32(0), T(1.0f));
  i32 i;
  _for (i = 1, i <= OPT_TIMESTEPS, i++) {
    upow_tbl =  
      replace(upow_tbl, u32(i), u * upow_tbl[i - 1]);
    dpow_tbl =  
      replace(dpow_tbl, u32(i), d * dpow_tbl[i - 1]);
  }
  //_end_for;
  _for (i = 0, i <= OPT_TIMESTEPS, i++) {
    opt_prices =  
      replace(opt_prices,  
        u32(i), max(T(0.0f), (exe_price -  
          upow_tbl[OPT_TIMESTEPS - i]  
        ) * dpow_tbl[i])));
  }
  //_end_for;
  _for (i = OPT_TIMESTEPS - 1, i >= 0, i--) {
    i32 j;
    _for (j = 0, j <= i, j++) {
      T tmp_value = p * opt_prices[j]  
        + (T(1) - p) * opt_prices[j + 1]  
        * multiplier;
      T tl = upow_tbl[i - j] * dpow_tbl[j];
      T early_exe_value =  
        max(exe_price - tl, T(0.0f));  
      opt_prices =  
        replace(opt_prices, u32(j),  
          max(early_exe_value, tmp_value));
    }
    //_end_for;
  }
  //_end_for;
  opt_price =  
    opt_prices[uncaptured<usize>::type(0)];
}

In Figure 10: Elemental function for binomial tree option pricer. Figure 10, we show the code example of Monte-Carlo.

The _for loop in the second half of the figure is parallelized via task parallelization as each loop iteration is independent with other iterations. Thus, Parallel Loop Analysis improved the performance by nearly 2x, as shown in Figure 8.

Mandelbrot includes a big fusion code region, and it shows a large benefit, nearly 1400% improvement, or 15x.
6.6 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 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.

7. Summary and Conclusions

The Array Building Blocks compiler makes use of a language embedded within C++ or other such 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 is implicit, but well defined, upon access to data.
- Safety is ensured by using a logically-separate data space, and 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 3.2-3.6x, and a geomean speedup on 8 dual-threaded cores in the range of 11.6-14.3.

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 15x can occur from fusion, indicating that it’s a primary source of parallelization performance, while many others, like parallel loop analysis, fusing shape-related operators, and fusion of operations on different data types can nearly double performance.

8. 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 remains. Memory bandwidth appears to limit performance in many cases, and we’re investigating how to reduce memory traffic to gain scalability. We intend to provide a more detailed discussion of performance outliers in the final version.
