



Intel[®] Technology Journal

Multi-Core Software

Future-Proof Data Parallel Algorithms and Software on Intel[®] Multi-Core Architecture

Future-Proof Data Parallel Algorithms and Software on Intel[®] Multi-Core Architecture

Anwar Ghuloum, Corporate Technology Group, Intel Corporation
Terry Smith, Corporate Technology Group, Intel Corporation
Gansha Wu, Corporate Technology Group, Intel Corporation
Xin Zhou, Corporate Technology Group, Intel Corporation
Jesse Fang, Corporate Technology Group, Intel Corporation
Peng Guo, Corporate Technology Group, Intel Corporation
Byoungro So, Corporate Technology Group, Intel Corporation
Mohan Rajagopalan, Corporate Technology Group, Intel Corporation
Yongjian Chen, Corporate Technology Group, Intel Corporation
Biao Chen, Corporate Technology Group, Intel Corporation

Index words: parallel programming, data parallelism, forward scalability

ABSTRACT

Developers face new challenges with multi-core software development. The first of these challenges is a significant productivity burden particular to parallel programming. A big contributor to this burden is the relative difficulty of tracking down data races, which manifest non-deterministically. The second challenge is parallelizing applications so that they effectively scale with new core counts and the inevitable enhancement and evolution of the instruction set. This is a new and subtle change to the benefit of backwards compatibility inherent in Intel[®] Architecture (IA): performance may not scale forward with new micro-architectures and, in some cases, may regress. We assert that *forward-scaling* is an essential requirement for new programming models, tools, and methodologies intended for multi-core software development.

We are implementing a programming model called the *Ct* API that leverages the strengths of data parallel programming to help address these challenges of multi-core software development. In this paper we describe how *Ct* is designed for minimal effort by the developer, while providing forward scaling on multi-core IA. We describe how *Ct*'s design and implementation evolved from the initial prototype, based on co-traveler feedback, and we provide examples of how *Ct* can be used. We demonstrate how a sampling of key application spaces can be easily written using *Ct* to achieve high performance. Finally, we

discuss how these ideas can be transitioned into mainstream software development tools.

INTRODUCTION

The data parallel style of programming [3][9][10][15] is best encapsulated in programming models in which collections of data elements are operated on en masse using various operators. For example, if a programmer wishes to sum the elements of two vectors (or matrices, trees, or sets) together, she simply writes an expression that adds these collections free of the bookkeeping and overhead typically associated with threaded programming (i.e., $A = B + C$).

Lately, data parallelism has (re-)emerged as an important topic in multi-core application development for a number of important technical reasons. First, many algorithms, including much of what is considered “low-hanging fruit,” are appropriately characterized as data parallel in nature. Second, data parallel programming models offer the elusive, yet highly desirable, property of determinism, which effectively eliminates data races as a class of programmer errors. Put simply, this means that the programmer writes code that behaves the same way regardless of the number of cores on which it is executed. Third, data parallel programming models are generally highly portable, offering the possibility of building parallel applications that adapt to new micro-architectures.

Another highly prized characteristic of data parallel programming models is a predictable and relatively simple performance model. This allows the programmer to consider performance in software design without focusing on the specifics of the underlying architecture. A related consequence of this characteristic is that data parallel algorithms provide a means to future-proof applications. As previously mentioned, a significant challenge to programming for multi-core architectures is *forward-scaling* performance in applications on evolving multi-core architecture. The performance of parallel applications is very sensitive to core count, vector ISA width (e.g., SSE), core-to-core latencies, memory hierarchy design, and synchronization costs¹. Software development tools must abstract these variations so that software performance continues to reap the benefits of Moore's law. The built-in performance model of data parallel programming naturally accomplishes this. Figure 1 illustrates how a compiled Ct binary can dynamically be reoptimized for these changing parameters.

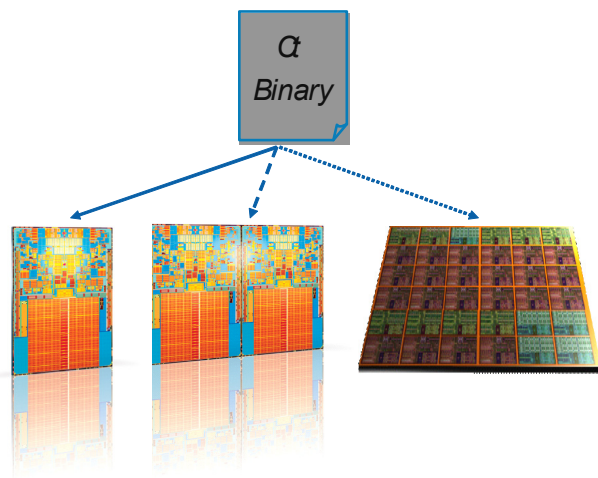


Figure 1: Forward scaling with Ct

An important goal of Ct is to extend the benefits of data parallel programming to less structured task parallel programming. We also aim to address highly object-oriented application designs. Because of this, we have developed a set of technologies to go well beyond basic data parallelism. For example, the underlying model of parallelism used by Ct is a sophisticated implementation of fine-grained concurrency and synchronization that we can progressively expose through the evolving Ct API.

In the next section of our paper we describe key factors and trends driving modern software development and how they are impacted by multi-core programming. Following that, we describe parallel programming models and show

where data parallelism lives from a taxonomic point of view. We then describe the Ct API and its implementation in detail and conclude with examples of Ct in action for typical algorithms.

SOFTWARE DEVELOPMENT DRIVERS

Software development takes fundamentally different processes and paths in different market segments. We believe that it is essential to understand these variations to adequately solve multi-core development challenges.

These are the key factors driving the adoption of parallel programming for multi-core architecture:

- *Productivity*: In most market segments, programmer productivity is a major factor in adopting new methodologies for programming, regardless of the benefits. Programmer productivity directly impacts cost and time-to-market, the latter of which is often driven by seasonal milestones. Productivity is adversely impacted by (newly introduced) parallelism-related bugs, performance tuning, and porting to increasingly parallel architectures.
- *Performance*. Raw performance, as measured directly by frame rate in a game or indirectly as new features enabled, is a first-order concern for most ISVs. However, there is frequently (though not always) tradeoffs against productivity-driven metrics like time-to-market.
- *Incremental adoption*. It is probably unreasonable to expect a company with an investment in several hundreds of thousands (or even millions) of lines of code to rewrite this code completely for parallelism. Rather, incremental adoption of parallelism features is the most likely scenario for the typical software developer. This carries with it several interoperability burdens: legacy binary libraries, existing code, and legacy threading APIs. For example, many developers use OpenMP or MPI to parallelize their code. It is paramount that new bridging technologies for parallel computing work well with these components.
- *Object-oriented design methodologies*: These can be viewed as another legacy interoperability issue, but their uniqueness and pervasiveness warrants separate consideration. In the last couple of decades, highly abstracted, objected-oriented programming styles have prevailed in the general software engineering community. The reasons are obvious: increasing abstraction levels facilitate more generic programming methods that increase code reusability. In many instances (C++, for example), programmers have found unexpected ways to use highly abstracted libraries through template meta-programming (Ct

¹ These are not necessarily orthogonal parameters.

itself leverages this!). This trend, however, runs counter to what the compiler and performance optimizer needs to see to generate high-performance parallel code: i.e., well-defined regions (typically, loops) of compute intensive execution. Any mainstream parallelism features must integrate smoothly into these programming methodologies.

This all boils down to the following seemingly untenable requirement: *Developers want a useful high-level programming model that introduces no parallelism-related bugs, yields high performance, and interoperates smoothly within the designs of their existing code base.*

In the next section, we describe how to characterize programming models in a way that serves this requirement.

DATA PARALLELISM BASICS

Parallel programming takes on many flavors. Traditionally, parallel programming models have been compared using dimensions such as message passing versus shared memory, or task (or control) versus data parallelism. However, the portability and expressive power of a particular manifestation of a programming model can transcend these issues. For example, some programming models are amenable to implementation on both shared memory and message passing systems. Also, many algorithms can be equally expressed using either task or data parallelism.

Despite the numerous formal and informal attempts to classify parallel programming models in this vein, we have chosen to measure success by specifically addressing the issues we raised in the previous sections. Our goal is to demonstrate all of these characteristics in our design:

- *Expressive power.* This is the ability to succinctly express different parallel algorithms in a model. For example, task parallel models support data parallel algorithms, though data parallel models cannot easily express some forms of task parallel algorithms. For a given application class, one style of programming model is likely to be prevalent.
- *Determinism.* A deterministic model has no possibility of data races introduced by the programmer, eliminating this new class of bugs. This directly impacts programmer productivity, though tools may mitigate this.
- *Performance transparency.* At the lexical level, it is possible to predict performance to varying degrees of accuracy. This often has a greater impact on programmer productivity, as it requires significant effort and low-level architectural understanding to tune performance on highly parallel architectures.

- *Portability.* Architectural portability is closely related to the requirements for forward-scaling multi-core applications. As the core count is scaled in multi-core architectures and new ISA enhancements are introduced, portable models are necessary to reliably leverage these features.

Expressive Power

Data parallel programming models allow the programmer to specify parallelism implicitly as operators on collections of data. For example, if a programmer wants to add to arrays of data in element-wise fashion, a data parallel programming model would be able to find parallelism roughly proportional to the amount of data in each array. So, if the arrays have 1,000 elements each, this comprises 1,000 independent (and potentially parallel) operations. To perform this computation, the data parallel model's implementation may choose to use parallel threads or tasks and vector instructions at its discretion.²

In the early days (the 60s and 70s) of parallel computing, this style of data parallelism was prevalent in languages like APL [3][15] and in the loop-y programming styles of Fortran (where the compiler did the heavy lifting with little guidance from the programmer).

The typical base data type in a data parallel programming model is an array or vector. Sometimes, these can be multi-dimensional. This has been the cornerstone of most models, but it can limit expressiveness. For example, flat or multi-dimensional vector-based models were most readily useful for dense linear algebra and signal or image processing applications. Moreover, complex computation patterns, like recursive subdivision or divide-and-conquer, were severely constrained in these models. Still, a large swath of applications found these models useful.

The key to broadening the applicability of data parallel models is to become more generically "collection-oriented." That is, by adding more types of collections that are supportable, the model becomes more expressive. For example, in the late 80s and early 90s, APL2 [4][14] and Nesl [11][18] added support for *segmented* vectors (see also [17] for a latter day example), which allowed the programmer to represent both irregular data structures and control flow. Per the former, sparse linear algebra was productively programmed using Nesl. Per the latter, divide-and-conquer algorithms like *quicksort* and *quickhull* were easily programmed. Paralation Lisp [19] and CM-Lisp [12][13] added support for indexed vectors,

² This computation can also be expressed as a task parallel computation, where we would "spawn" tasks for each of the 1,000 additions, followed by a synchronization to ensure that the computation is completed.

allowing even more complex data structures (including additional sparse representations) to be represented. Ct builds on these algorithms.

There are limitations to the applicability of the data parallel model. For example, applications that require tasks that make asynchronous updates to shared data will generally not map well onto this model. A Web server is a very good example of such an application. It is important to note that most applications require a variety of parallel programming models, so despite the prevalence of data parallelism for these applications, other flavors of parallelism are often required.

Determinism

The data parallel model generally relies on a compiler and/or runtime to manage task creation and usage of vector instruction; there is no explicit thread spawning or synchronization necessary, so data races are non-existent as far as the programmer is concerned. Though the data parallel model can provide fairly sophisticated data movement and communication primitives, it preserves this model.

For example, Ct provides many *collective communication* primitives, including the ability to perform a sum reduction on a vector. This entails summing all elements of the vector in parallel, which requires re-associating the computation. However, the programmer need only specify the reduction operator and leave the necessary threading and synchronization to the runtime. When considering nested or indexed vectors, the semantics of the operator are much more complex, but the programmer's view is as simple as a flat vector reduction.

Performance Transparency

Though the data parallel model constrains expressiveness somewhat, this property and its high-level abstraction bespeak a relatively predictable performance model. When programming with threads and lower-level synchronization constructs, it is difficult to predict when serialization (intended and unintended) will happen. Moreover, it is extremely difficult to predict memory-related performance issues, since predicting the volume of data accessed and any potential conflicts between threads is often rendered intractable by the high level of abstraction used in modern software.

Operations on collections have the desirable properties that the programmer can predict relative performance behaviors based on collection size and operation complexity. For example, a 1,000 by 1,000 element 2 dimensional matrix generally introduces up to 1,000,000-way parallelism, meaning that for up to thousands of hardware threads, the computation is likely to be able to profitably scale. Furthermore, a collective communication

primitive is likely to engender more synchronization than an element-wise operation (which often optimizes away to no synchronization). Though the exact performance is still difficult to predict, these higher-level tradeoffs allow the programmer to make good algorithmic choices.

Portability

Data parallel models have been mapped to a wide range of architectures, from massively parallel distributed memory architectures, to shared memory multi-processors, to deeply pipelined vector supercomputers, to GPUs. This portability is critical to the matching software requirements for evolving multi-core architecture.

This evolution is following several paths. First, the core count will increase, requiring ever increasing amounts of parallelism. Second, non-uniformity of memory access time between cores is increasing, meaning that typical memory access latencies will exhibit high variance to predict unless data partitioning is done carefully. Somewhat related to these two considerations, relative core-to-core synchronization costs will change, requiring re-optimization of code to make the best hide-related latencies. Third, we expect the instruction set improvements to continue, requiring quick adaptation to these enhancements.

The resiliency of data parallel models in many different operating environments is evidence of its ability to adapt to these changes. In particular, the programmer can expect that an algorithm written in a data parallel style will scale across generations of multi-core architectures, using ever-more cores and leveraging newer and wider vector ISAs while avoiding the pitfalls of unintended serialization through the memory hierarchy.

CT

Brief Ct Overview

Ct is a data parallel programming environment with predictable syntax based on C++ that provides distinct semantics and performance [6].

Unique among commercial data parallel programming models, Ct implements a *nested data parallel* model based on work on Nesl [18] and Paralation Lisp [19]. Ct's nested data parallelism enables a far broader set of collections to be represented. For example, sparse matrices and trees are very difficult to represent in flat data parallel or streaming models. However, these fall out naturally in a nested data parallel model. Also, common divide-and-conquer algorithms, for example, KD-tree construction and sorting, are very difficult to express using flat data parallel and streaming models. These are readily expressed using nested data parallelism. Nested data parallel computations

generally do not port efficiently to GPUs and streaming architectures, but they run efficiently on multi-core IA.

Unlike many of its data-parallel brethren, Ct also supports *deterministic task parallelism* on multi-core IA (inspired by [16]). Determinism guarantees that program behavior is identical, on one core or many cores. This essentially eliminates an entire class of programmer errors—data races.

TVECs

The basic type in Ct is a generic vector type, called TVEC. TVECs are allocated and managed in a segregated memory space that is accessible only by Ct operators, to ensure the safety of parallel operation on vectors. TVEC is polymorphic in terms of its base types and shapes.

The base types of TVECs are drawn from a set of typical pre-defined scalar (or value) types. Examples of base types include `I32` (32-bit integer), `I64` (64-bit integer), `F32` (Float), `F64` (Double), and `Bool` (Boolean). In future, Ct will also support the `Bit` type and user-defined base types, for example, C struct-like base type, `TSTRUCT`, and the C array-like base type, `TUPLE`, for more complicated application scenarios.

A TVEC may be declared as follows:

```
TVEC<F32> temp;
TVEC<F32> prices(option_prices, num_options);
TVEC<I8> red(image, length, 4/*stride*/);
```

The TVEC constructor copies data explicitly from the unmanaged C/C++ memory to the managed vector space, in the form of either plain element-wise copy, or the strided memory copy (`red` in the example above takes one byte from every four of the data stream). There are also exceptional cases when it is not preferable to copy the data all at once (because of long latency) or we do not want to copy at all. Thus, there are several TVEC traits that may be applied, including `Stream` for copying data in a streaming fashion, or `Direct` for not copying.

```
TVEC<F32, Stream> stream;
TVEC<F32, Direct> mapped_file;
```

Constant TVECs may also be constructed by factory methods. For example, an identity matrix, in the form of `TVEC2D` (a TVEC derivative for matrices), may be created as follows:

```
TVEC2D<F32> id = TVEC2D<F32>::identity(dim);
```

Nested data parallelism is a distinguished property for programming irregular data structures and algorithms. TVECs assume a number of shapes, including flat, multi-dimensional, irregular nested, and indexed forms. For example, a matrix TVEC could be constructed as follows:

```
TVEC2D<F32> matrix(data, width, height);
```

TVECs may also be associated with certain accuracy attributes, which may allow experienced programmers to influence the compiler's code generation. For example:

```
TVEC<F32, Default, 2/*ulp*/> data;
... = sqrt(data);
```

The above TVEC declaration specifies 2 ulp (*units-in-the-last-place*) as the tolerable accuracy threshold, which gives a hint to the compiler that the square root operator may be translated into a simpler code sequence with lower-order polynomials and less fix-up code. However, if 0.5 ulp is specified, the compiler may generate a more complicated code sequence that might be up to 60+% slower on some architectures.

When the computation on TVECs is completed, the computed results may be transferred back to the unmanaged space through the `copyOut` primitive.

Ct Operators

The only operators allowed on TVECs are Ct operators. Ct operators are functionally pure (free of side effects). That is, TVECs are passed around by value, and each Ct operator logically returns a new TVEC. For example:

```
scaled_red = red * 0.5; //a new TVEC is born
```

This property guarantees the safety of parallelism and the aggressive optimizations that make parallelism efficient.

The Ct API provides a broad range of Ct operators with rich functionalities. Operator overloading is used extensively to support a programming style, based on C++, particularly for the arithmetic, bitwise, and logical/comparison operators. For example, the `*` operator in the above example is overloaded to the TVEC multiply operator.

Basically Ct operators can be categorized into element-wise/vector-scalar, collective communication, and permutation operators.

Element-wise/vector-scalar operators are typically referred to as “embarrassingly” parallel, requiring no interactions between the computations on each vector element. An example of an element-wise operation is the addition of two vectors:

```
TVEC<F32> A = B + C; //“+” resolves to ctAdd
```

Note that this code generically performs an element-wise addition of two vectors, regardless of the “shape” of the two vectors (i.e., their length, dimensionality, irregularity).

Collective communication operators tend to provide distilled computations over entire vectors and are highly

coordinated³. While they have a high degree of interference, they can be structured so that there is parallelism in colliding writes, and they typically scale in performance linearly with processor count, with little or no hardware support.

There are two kinds of collective communication primitives in general, namely reductions and prefix-sums (also called scans). Reductions apply an operator over an entire vector to compute a distilled value (or values, depending on the type of vector). Prefix-sums perform a similar operation, but return a partial result for each vector element. For example, an `addReduce` sums over all the elements of a vector if the vector is flat. More concretely, `addReduce([1 0 2 -1 4])` yields $1+0+2+(-1)+4=6$. Likewise, `addScan([1 0 2 -1 4])` yields $[1\ 1+0\ 1+0+2\ 1+0+2+(-1)\ 1+0+2+(-1)+4]$.

A *permutation* operator in Ct is any operator that requires moving data from its original position to a different position. An example is the `gather` operation, which uses an index array to collect values of a vector in a particular order; and the `scatter` operator does the reverse. Permutations run the gamut, from arbitrary permutations with arbitrary collisions (occurring when two values want to reside in the same location) to well-structured and predictable permutations where no collisions can occur. For collisions, it is recommended that programmers make use of the collective communication operators. An example of a well-structured (and efficient) permutation operator is `pack`, which uses a flag vector to select values from a vector in the source vector order. With proper hardware support on multi-core IA, these operators can be implemented fairly efficiently. In contrast, these operators could not be implemented efficiently on constrained architectures (for example, most GPUs do not efficiently support scatter operations).

Besides these built-in operators, Ct also supports generic user-defined operators through Ct functions. As implied by their name, Ct functions define a block of code that is applicable to a collection of vectors, which allows programmers to define new generic operators or functions for repeated application (mitigating compilation overhead). The following code defines a Ct function that performs a fused multiply-add:

```
F32 fma(F32 a, F32 b, F32 c)
    return a + b * c;
```

³ These operators are called collective communication operators in MPI and reductions in OpenMP, though neither provides the rich set of operations that Ct does. In functional languages, these are termed fold operations or list homomorphisms.

We use a map operator that takes as arguments the Ct function pointer and three vector arguments to apply this function in an element-wise manner:

```
map(fma, ta, tb, tc)
```

The implementation of the map operator employs compile-time type inference to prevent programmers from specifying improper arguments, such as `TVEC<I32>` (which is not conformant to this function's definition), or wrong numbers of arguments. Just like C/C++ routines, Ct functions are composable, greatly extending Ct's expressiveness.

Nested Vectors

Ct's support for nested vectors is a generalization that allows a greater degree of flexibility than is otherwise found in most data parallel models. TVECs may be flat vectors or regular multi-dimensional vectors. They also may be nested vectors of varying length, which allows for very expressive coding of irregular algorithms, such as other variants of sparse matrix representations, or byproducts of divide-and-conquer algorithms.

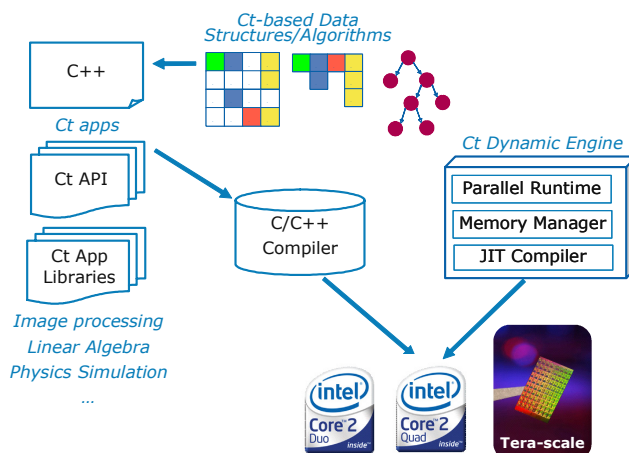


Figure 2: The usage and implementation of Ct

The vector value $[a\ b\ c\ d\ e\ f]$ is a flat (or 1-dimensional) vector. The vector $[[a\ b][c\ d\ e\ f]]$ holds the same element values, but is a vector of two vectors of lengths 2 and 4. The second vector might represent a partitioning of the first vector's data based on certain criteria (e.g., the relationship to a pivot value in a quicksort). Practically, the nested format enables a lot of irregular data structures and algorithms. Figure 2 gives a few such examples.

All Ct operators work on nested TVECs generically. The behavior of element-wise operators is the same for nested TVECs as for flat vectors. For example, $[[a\ b][c\ d\ e\ f]] + [[g\ h][i\ j\ k\ l]]$ yields $[[a+g\ b+h][c+i\ d+j\ e+k\ f+l]]$.

The power of nested versus flat TVECs is primarily differentiated through the behavior of collective communication and permutation primitives. Collective communication primitives applied to nested TVECs “respect the boundaries” of the subvectors by applying the operator to each subvector independently. For example, `addReduce([a b c d e f])` yields the singleton vector `[a+b+c+d+e+f]`, while `addReduce([[a b][c d e f])` yields the two-element vector `[a+b c+d+e+f]`.

IMPLEMENTING CT FOR FORWARD-SCALING

Figure 2 also illustrates the Ct execution model. The core of Ct-enabled applications is the use of Ct-based data structures and algorithms. In addition, Ct Application Libraries are a set of well-optimized higher-level APIs aiming to boost programmers’ productivity for Tera-scale applications such as image processing, linear algebra, and physics simulation. The Ct libraries can be compiled by stock C++ compilers, such as Visual C++, Intel® C/C++, and Gnu C/C++ compilers, into an IA binary that is able to run on all multi-core IA platforms. This binary comprises of either IA32 or Intel® 64 Architecture instructions, which also include calls to the Ct Dynamic Engine. During the execution of the binary, the Ct Dynamic Engine is launched and provides the services essential to performance and forward-scaling. More specifically, the three major services are the Threading Runtime (TRT), Memory Manager (MM), and Just-In-Time (JIT) compiler. In particular, the TRT and JIT (especially the vector abstraction we will introduce called VIP) provide the basis for forward scaling across IA.

The Threading Runtime

The first key to forward-scaling is to *dynamically adapt to new architectural characteristics*. Threading and synchronization overhead is likely to change between processor generations, necessitating an ability to both select the task granularity and synchronization method dynamically. In fact, our approach is to isolate the architecture-dependent components of the Ct runtime to dynamically loaded libraries. Another aspect of forward-scaling is that data set sizes are likely to scale in the long run, but are generally unpredictable in phases of computation, especially for client applications such as games. In this case, the amount of data being processed is highly scene and gameplay dependent. As such, the runtime must be able to adapt its threading strategy to variable data sets.

The TRT provides a fine-grained threading model that is used to implement both data parallel and task parallel constructs. The underlying building block for this model is a *future*, which under the runtime semantics may represent

a suspended closure or *thunk* (i.e., a function pointer and an argument list representing a potentially parallel function application), a *thunk* computation *in flight*, or a *computed* value (representing a successfully evaluated *thunk*). A handle to a *future* essentially represents a dependency on that suspended *thunk*’s evaluation. This is inspired by the techniques for expressing and managing parallelism presented in [7][8]. Using this mechanism, many complex fine-grained synchronization patterns may be expressed; however, the TRT facilitates fine-grained synchronizations via a building block called a *join*. A *join* can be used to express a range of logical combinations of synchronization dependencies.

The TRT uses additional primitives called *bulkspawns* and *bulkjoins*, which essentially represent mapped future spawns and joins on collection-oriented arguments. Bulkspawn operations dynamically partition the collection into the *right number* of fine-grained tasks interlinked with fine-grained synchronizations. This is key to adapting to the core count and utilization, as well as cache footprint.

The Memory Manager

The Ct MM automatically manages the segregated Ct vector space. As such, it provides a set of lock-free memory allocation interfaces, as well as a reference-counting-based garbage collector to reclaim dead vectors automatically. The MM is responsible for allocated data format and, in conjunction with the TRT, partitions vectors for parallel operations (i.e., the TRT bulkspawn operations).

The Compiler

The Ct compiler has a slightly unconventional structure, notably in its dynamic nature. When executing Ct API calls, the dynamic engine constructs intermediate representations of the computation, deferring actual execution (and further optimization) until *later*. “Later” is bounded by the necessity to copy values back into native C/C++ space, though the engine may decide to compile code at intermediate steps, such as when back edges in control flow (i.e., loops) are detected. This intermediate representation (IR) building is the default mode of Ct code execution for new paths in the program. Otherwise, cached code is executed if the path followed is in the “Code Cache,” or a cached IR is augmented.

Once the compiler is invoked, several phases with distinct objectives are invoked: the High-Level Optimizer (HLO), the Low-Level Optimizer (LLO), and the VIP Code Generator (VCG).

The HLO phase [5] performs architecture- and runtime-independent optimizations, such as sub-primitive decomposition (breaking up data parallel operators into

more primitive patterns of parallelism), fusion (essential to coarsening the fine-grained concurrency of the Ct model as much as possible), scalarization, common sub-expression elimination, and copy propagation. These optimizations are all possible without introducing the details of run-time memory allocation and shape checks. This is left to LLO.

The LLO phase is still architecture independent, but unlike HLO, LLO does runtime-dependent optimizations. It has three primary objectives: 1) generate parallelized kernels using the TRT; 2) translate the optimized kernels (spawned in the TRT) into proper vectorized code; 3) generate architecture-independent representations, which we call the Virtual Intel[®] Platform, or VIP. Much of the difficulty of implementing and optimizing collective communication [1] and segmented operators [2] is deferred to this phase. VIP is an abstract instruction set that is based on IA32/Intel 64 Architecture, but that uses a generalized vector ISA to defer binding to a particular generation of SSE as late as possible.

One of the challenges we face for forward-scaling is the *near certainty of SSE extensions and enhancements*. Figure 3 shows the evolution of Intel SIMD ISA. The number of instructions has been increasing at the pace of 30 instructions per year on average since the introduction of MMX[™] technology in 1996. Meanwhile, the data width of vector registers was also increased from 64 bits to 128 bits, and it can be reasonably expected to increase further at some point in the future. This is driven by the need to increase performance in the most power efficient way and extending SIMD ISA is one such mechanism. VIP, as a virtual ISA of the Ct Dynamic Engine, is designed to hide changes in SIMD ISA, and, via VCG, provide future-proof performance to Ct applications.

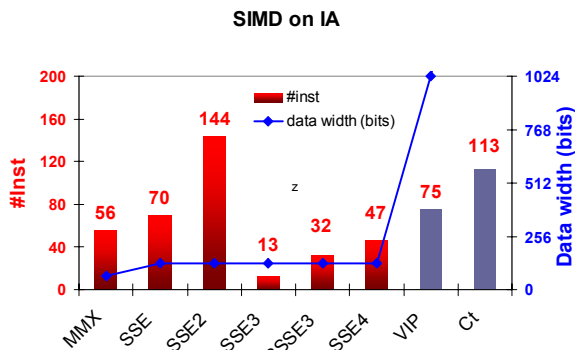


Figure 3: The evolution of IA SIMD ISA vs. Ct

The VCG phase is a state-of-the-art backend for VIP that dynamically selects the appropriate target ISA. When new SSE extensions are introduced, a new dynamically linked library can be made available that supports both legacy

and new SSE extensions. No recompilation with a static compiler is necessary. In this way, applications can forward scale through vector ISA with the adaptivity of the VCG backend. VCG does classic loop-based optimizations, such as loop fusion, loop interchange, and array contraction. VCG also does architecture-dependent optimizations, such as register allocation and instruction scheduling.

By carefully layering architecture and run-time dependent optimizations in the Ct compiler, we can retarget the entire dynamic engine with great agility, including for the purposes of evaluating new micro-architectures (i.e., considering in-order architectures and evaluating new, throughput-oriented ISA extensions). This was done deliberately with forward-scaling in mind.

The dynamic compilation approach, especially the VIP layer, provides smooth migration paths to future SSE and IA-based SIMD instruction sets.

CT IN ACTION

In this section, we walk through some examples to demonstrate how Ct boosts the productivity and performance for a variety of application domains. We take a step-by-step approach, to make clear some guidelines for porting to Ct. In particular, we provide rules of thumb for translating sequential code to Ct code.

Black-Scholes

Option pricing is a computation-hungry, important application in modern financial engineering. Black-Scholes is a well-accepted analytical model for European option pricing. We use it here as an exemplar for C/C++ to Ct migration. The code below shows the sequential C code.

```

1 float s[N], x[N], r[N], v[N], t[N];
2 float result[N];

3 for(int i = 0; i < N; i++) {
4   float d1 = s[i] / ln(x[i]);
5   d1 += (r[i] + v[i] * v[i] * 0.5f) * t[i];
6   d1 /= sqrt(t[i]);
7   float d2 = d1 - sqrt(t[i]);

8   result[i] = x[i] * exp(r[i] * t[i]) *
9     ( 1.0f - CND(d2)) + (-s[i]) * (1.0f -
10     CND(d1));
10 }

```

The code below shows its Ct counterpart. The two pieces of code are very similar (lines 4-9).

```

0 #include <ct.h>

1 T s[N], x[N], r[N], v[N], t[N];
2 T result[N];
3 TVEC<T> S(s, N), X(x, N), R(r, N), V(v, N),
4 T(t, N);

```

```

4 TVEC<T> d1 = S / ln(X);
5 d1 += (R + V * V * 0.5f) * T;
6 d1 /= sqrt(T);
7 TVEC<T> d2 = d1 - sqrt(T);

8 TVEC<T> result = X * exp(R * T) *
9 ( 1.0f - CND(d2)) + (-S) * (1.0f -
                                CND(d1));

```

The only differences are these:

- Ct needs to include the `ct.h` header file (line 0).
- Ct adds the TVEC declarations (line 3).
- Ct exempts programmers from having to manipulate arrays with loops and subscripts (lines 3-9).
- The Ct version co-exists well with C++’s parametric polymorphism, allowing the code to be instantiated with different types `T`.

The desirable tradeoff here is that the coding overhead is small when migrating to Ct, but you get highly efficient vectorized, parallelized, and forward-scaling code. In contrast, a manually vectorized version using MMX/SSE intrinsics has 51 lines of code, and the manual parallelization using threads requires an additional 20+ lines of code. When the underlying hardware or OS changes, you may need to modify the code to use new intrinsics and change the number of threads or the threading primitives (e.g., `pthread`).

The code below shows how a Cumulative Normal Distribution function, CND, is implemented in C and Ct, respectively. Though the C code can be translated into Ct easily, we use a Ct Application Library function, `Ct::Polynomial::eval`, to accelerate the polynomial evaluation. Our data show that for a 5th-order polynomial, the optimized Ct library yields ~3X speedup over the naïve implementation with negligible precision loss.

```

float CND(float d) { // The C version
    ... ..
    w = 0.31938153f * k -
        0.356563782f * k * k +
        1.781477937f * k * k * k -
        1.821255978f * k * k * k * k +
        1.330274429f * k * k * k * k * k;
    ... ..
}

template <typename T>
TVEC<T> CND(TVEC<T> d) { // The Ct version
    ... ..
    static T coefficients[] = {
        0.0f,
        0.31938153f,
        0.356563782f,
        1.781477937f,
        1.821255978f,
        1.330274429f};

    w = Ct::Polynomial::eval(k, 5/*order*/,
        coefficients);
    ... ..
}

```

Rule of Thumb I:

```

for(int i = 0; i < N; i++)
    ... data[i] ...
→
... data ...

```

Convolution

Convolution is a widely used function in many application domains ranging from signal/image processing to statistics, to geophysics. Compared with the very parallel Black-Scholes, the computation pattern of Convolution is slightly trickier. In particular, programmers have several mechanisms at their disposal and we will explore the tradeoffs between these approaches.

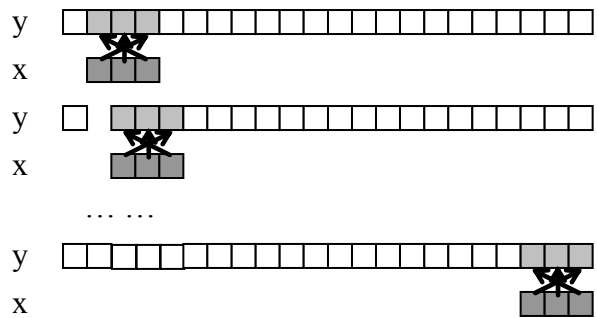


Figure 4: Convolution algorithm illustration (1 dimensional)

Figure 4 is a typical 1D convolution algorithm, where `y` is a data set, and `x` is a kernel sliding through the data set. The code below gives the C implementation⁴. As compared to Figure 4, you may find this loop structure is more complicated and the array access pattern is more irregular (particularly `y[i - j]` on Line 5).

```

float x[M], y[N], z[N];
for (int i = 0; i < N; i++) {
    z[i] = 0.0f;
    for (int j = 0; j < M; j++)
        z[i] += x[j] * y[i-j];
}

```

Our first question is how to map the two-level loops to TVECs. Obviously we want to abstract the data set, `y`, to be a TVEC. In this regard, we peel the outer loop and change all the occurrences of `y` to `Y`, as shown below.

```

T x[M], Y[N], z[N];
TVEC<T> Y(y, N), I = TVEC<T>::index(0, N);
TVEC<T> Z = TVEC<T>::constant(0.0f, N);
for (int j = 0; j < M; j++)
    Z += x[j] * Y[I-j];

```

⁴ The example is just for illustration purposes, and we omitted some code for checking boundary conditions.

The second question is how to represent $y[i - j]$ in Ct. Because i is a loop induction variable incremented by a step value 1, we map it to an identity vector, \mathbb{I} , which results in $y[\mathbb{I} - j]$. Ct's C++ front-end reinterprets the \mathbb{I} operator into a gather operator, gathering values from y according to an index vector, $\mathbb{I} - j$.

This porting is straightforward but we can't claim this solution is ideal for performance, because the gather operator is expensive on most architectures. Experienced Ct programmers, when understanding the algorithm, may resort to a more lightweight operator, `shiftPermute`. If you look at Figure 4 from a different perspective, that is, the kernel x is fixed while the data set y is sliding leftward, the result is equivalent. The optimized implementation is shown in the code below.

```
T x[M], y[N], z[N];
TVEC<T> Y(y, N);
TVEC<T> Z = TVEC<T>::constant(0.0f, N);
for (int j = 0; j < M; j++)
    Z += x[j]*Y.shiftPermute(1);
```

It is worthwhile to mention that the Ct implementation can be extended to 2D convolutions with minimal effort.

Rule of Thumb II:

```
for(int i = 0; i < M; i++)
    for(int j = 0; j < N; j++)
        ... data[i + a, j + b] ...
→
... data.shiftPermute(a, b) ...
```

Sparse Matrix Vector Product (SMVP)

Linear algebra, particularly matrix operations, is quite common in high-performance computing, physics simulation, aspects of machine learning, and many Recognition, Mining, and Synthesis (RMS) applications. Sparse matrices are extremely useful for cases where the particular algebraic formulation of a problem sparsely populates elements in the matrix with meaningful values.

An example is large-scale physics simulations. In such cases, the logical size of a dense matrix might be 100s of megabytes, where a sparse matrix representation that only stores non-zero matrix elements would perhaps only hold 1 megabyte of data. Unlike dense matrices, whose control paths and data access patterns are highly predictable, sparse matrices are much more hard in terms of the diversity of data structures and the irregularity of algorithms. In this section, we use a common kernel in gaming and RMS applications, Sparse Matrix Vector Product (SMVP), to demonstrate how a sparse matrix multiplied by a vector is implemented with Ct.

We use a Compressed Sparse Column (CSC) format. The basic idea of CSC is to only store the non-zero elements of

the matrix in the column order, and with each non-zero element, the programmer also stores the row index. Consider the sparse matrix below.

```
A = [[0 1 0 0 0]
      [2 0 3 4 0]
      [0 5 0 0 6]
      [0 7 0 0 8]
      [0 0 9 0 0]]
```

The matrix is stored as three arrays:

- `nz_mval`: the nonzero values, in column major order.
- `row_idx`: the row indices for nonzero values.
- `col_ptr`: the column pointers. `col_ptr[i]` tell the values of the i -th column start from which index into the `nz_mval` array).

```
mval = [2 1 5 7 3 9 4 6 8]
row_idx = [1 0 2 3 1 4 1 2 3]
col_ptr = [0 1 4 6 8 9]
vval = [1 2 3 4 5] // the vector values
```

The schema for computing the SMVP is shown below.

```
1 for (c = 0; c < col_num; c++) {
2   for (e = col_ptr[c]; e < col_ptr[c + 1]; e++) {
3     int r = row_idx[e];
4     product[r] += mval[e] /* A[r][c] */
5       * vval[c];
6   }
```

It is worth observing some of the computing patterns to comprehend the implications for porting to Ct:

- The two-level loops are more irregular than the aforementioned examples. Typically this kind of loop structure can be mapped to a two-level nested vector, as shown below


```
[
  [...], //col_ptr[1]-col_ptr[0] elems
  [...], //col_ptr[2]-col_ptr[1] elems
  ...
  [...], //col_ptr[col_num+1]-
        col_ptr[col_num] elems
]
```
- In the inner loop, `vval[c]` is used for `col_ptr[c+1] - col_ptr[c]` times, which can be viewed as a special kind of *gather* operation. In Ct, we have a dedicated operator, `distribute`, to replicate values of a vector for certain numbers of times specified by another vector.
- The expression `product[row_idx[e]] += ...` implies that we are performing what is called a *combining-send*, or alternatively a *multi-reduction* or *combining-scatter*.

By comprehending the patterns, we have the Ct implementation presented below:

```

1 vval = vval.distribute(col_ptr);
  //=> [1 2 2 2 3 3 4 5 5]
2 TVEC<T> product = mval * vval;
  //=> [2 2 10 14 9 27 16 30 40]
3 product.applyNesting(row_idx,
  ctNestedIndex);
  //=> [[2] [2 9 16] [10 30] [14 40] [27]]
4 product = addReduce(product);
  //=> [2 27 40 54 27]

```

Rule of Thumb III:

```

for(int i = 0; i < num_segs; i++)
  for(int j = seg[i]; j < seg[i+1]; j++)
    ... data[j] ...
→
data: [ [seg1], [seg2], ..., [segnum_segs] ]

```

Rule of Thumb IV:

```

for(int i = 0; i < M; i++)
  data[j] += ...
→
addReduce(data);

```

Experimental Results

We measured the performance of Ct and sequential C implementations of representative data parallel operators and a set of real-world applications on an Intel® Xeon® processor E5345⁵ platform (two 2.33GHz quad-core processors, 4GB memory), and plotted the speedup of Ct over C in Figure 5 and Figure 6. To present the benefits from the JIT compiler optimizations and the TRT separately, we configured Ct to run with different numbers of cores. The C implementations are compiled with Visual C++ 2005 compiler.

Figure 5 compares Ct's performance vs. C (compiled with O3-level optimizations on the Intel C Compiler) for a few key operators. These are common building blocks in many-core applications, though their use and mix is varied. It provides a reasonable baseline for assessing scalability based on the mix of Ct operators in your application. These operators can be categorized into two classes: `add`, `max`, `sqrt`, and `exp` belong to element-wise operators, while `addReduce` and `addScan` are collective communication operators.

The Ct implementations of almost all element-wise operators achieve 7-8X scalability when adding the

number of cores from 1 to 8. When considering only one core, the Ct Compiler's aggressive vectorization also makes significant difference:

- For `add`, Ct generated code achieves 2X speedup against a scalar implementation. Given SSE's vector width is 4, and the vectorized code is mixed with a lot of scalar code, the speedup is quite reasonable.
- For `max`, the Ct implementation takes advantage of SSE's `max` instructions, however, the C version uses control flow (e.g., `a > b ? a : b`) that are more challenging to vectorize. Thus the speedup reaches up to 11X.
- For `sqrt`, Ct generated code leverages SSE's `sqrt` instruction, while the C code relies on slow C runtime library implementation. As such, the Ct implementation achieves a significant speedup of 42X.
- SSE doesn't have direct support for `exp`. However, Ct still generates highly efficient SSE code sequence, based on a look-up table and interpolation-based method that outperforms the C runtime library-based implementation by 15X.

Unlike element-wise operators that are embarrassingly parallel, collective communication operators have more complicated inter-thread communication patterns. In the meantime, the collective operators also impose challenges on local code vectorization.

- The `addReduce` operator performs the summation of all elements of a vector. The Ct implementation achieves totally 93X speedup over the scalar implementation, where 12X comes from vectorization.
- The `addScan` operator requires more complicated communication patterns. Again, the speedup achieved by our optimized code is as high as 31X, where 5X is from code vectorization.

In the future, Ct's adaptive compilation strategy will play an even more important role when new, throughput-oriented ISA extensions emerge (such as `mask`, `cast/conversion`, `swizzle/shuffle`, and `gather/scatter`).

⁵ Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See www.intel.com/products/processor_number for details.

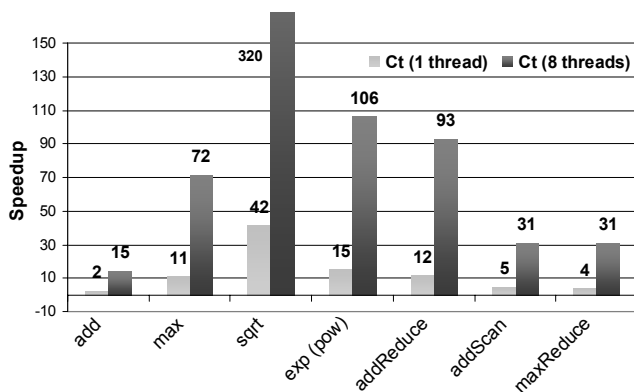


Figure 5: Performance of select Ct primitives vs. C compiled with optimizations

The applications being studied are listed in the table below. These span applications from high-performance computing, financial computing, image processing, through physics simulation. Black-Scholes, Binomial Tree, and Monte Carlo Simulation are three widely used option pricing models, 2D Convolution is a typical signal processing kernel, and the Narrow-Phase Collision Detection is a compute-intensive component of gaming physics.

Table 1: Applications characterized under Ct

Program	Description
Black Scholes	European Option Pricing (financial analytics)
Binomial Tree	American Option Pricing (financial analytics)
Monte Carlo	Asian Option Pricing (financial analytics)
Convolution	2D Convolution kernel (signal processing)
Collision Detection	Narrow-phase collision detection (game physics)

Figure 6 shows Ct’s performance on Core 2 Quad machines, as compared to plain C code and hand-compiler tuned SSE code (using SSE intrinsics). The figure also indicates that Ct code has good scalability when increasing the number of cores from 1 to 8.

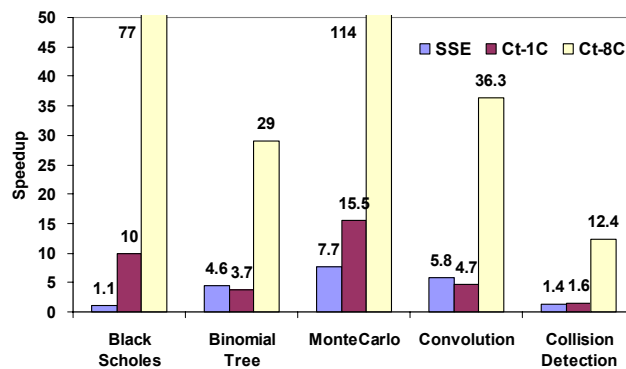


Figure 6: Application scaling with Ct

For single-thread performance, Ct achieves a speedup as high as 10X for Black-Scholes. Black-Scholes relies heavily on the performance of transcendental functions, namely *exp*, *log*, *rcp* and *sqrt*. SSE does provide efficient support for *rcp* and *sqrt*, while lacking support for *exp* and *log*. Typically, programmers fall back to a scalar loop and call corresponding C runtime functions for each element. Even though the rest of the program is well vectorized, the overall speedup of SSE over C is only 1.1X. Ct’s 10X speedup is mainly attributed to JIT’s use of vectorized implementation of such transcendental functions.

Monte Carlo Simulation has a 15.5X Ct-over-C speedup. Two factors contribute to the 8X speedup: first, Ct has a very fast, vectorized implementation of random number generator, while C has to resort to the C runtime function, *rand*; second, Monte Carlo Simulation heavily uses two transcendental functions, *sin* and *cos*, where Ct also has very efficient SSE-based implementations. Consequently, the speedup achieved by SSE is only 7.7X.

Single threaded Ct for Binomial Tree and Convolution achieve only 3.7X and 4.7X speedup, respectively, which is not surprising given that the two applications are not arithmetic intensive. Their SSE versions are slightly faster because the Visual C++ compiler uses a static compilation strategy that makes more aggressive optimizations affordable. An interesting note is that Binomial Tree suffers from many floating point underflow exceptions. Ct allows specifying lower numerical precision requirements. This enables the Ct Compiler to generate code under “flush-to-zero” mode, which speeds up the performance further by 3-4X. For cases when lower accuracy is not tolerable, we may specify *F64* (namely *double*) as the base type of *TVEC*. Although the SIMD data width is reduced to half, the underflow exceptions are totally removed, which speeds up the performance of the *TVEC<F32>* version by 2.5X. It is trivial for Ct programmers to get the speedup because only *TVEC* declarations are changed (i.e., they do not have to change a single line of code).

Note that this graph shows the performance when running the *same* Ct binary with different hardware configurations. Looking forward, Ct provides nice forward scalability. Note that relatively inexperienced C/C++ programmers can get these performance benefits (nearly) for free on stock machines. Porting C implementations to their Ct counterparts takes minor effort, and the Lines-of-Code of the two implementations are almost 1:1.

CONCLUSION

Future-proofing algorithms for multi-core architectures is an important way to continue to reap the performance benefits of Moore's Law scaling. Data parallel programming models offer a promising abstraction to use for forward-scaling, but it is often limited by too-narrowly defined types and operators. This severely limits the scope of applicability for such models. In Ct, we are attempting to build a system that delivers a more general data parallel (indeed, a deterministic task parallel) model while providing the essential framework for forward-scaling.

A serious challenge is acknowledging the realities of modern software development methods and assuring compatibility with legacy code and programming methodologies. We believe that using the Ct Dynamic Engine's particular flavor of adaptive compilation and run-time is the most effective way to extract chains of performance code without seriously compromising the language design or the large software investment by developers. More radical language redesigns are likely to appear at some point, but we view this more incremental approach as a flexible and highly productive way to leverage multi-core architectures while developing the basic parallel algorithms and design methods. In fact, we expect that future languages will almost certainly encompass some form (if not the exact form) of the ideas in Ct.

ACKNOWLEDGEMENTS

We thank other members in the Ct team, namely Zhanglin Liu, Dan Zhang, Jane Liu, Zhaohui Du, Josh Fryman, and Zhigang Wang for their **significant and indispensable** contributions to the work presented in this paper. We also wish to thank Joe Schutz, Shekhar Borkar, Jerry Bautista, and Justin Rattner for their support in this work.

REFERENCES

- [1] Allan L. Fisher, Anwar M. Ghuloum, "Parallelizing complex scans and reductions" in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pp. 135–146, June 20-24, 1994, Orlando, Florida, United States.
- [2] Anwar M. Ghuloum, Allan L. Fisher, "Flattening and parallelizing irregular, recurrent loop nests," *ACM SIGPLAN Notices*, v.30 n.8, pp. 58–67, August 1995.
- [3] "APL: A Programming Language," at <http://www.users.cloud9.net/~bradmcc/APL.html>
- [4] "APL2," at <http://www.ibm.com/software/awdtools/apl/>
- [5] Byoungro So, Anwar Ghuloum, Youfeng Wu, "Optimizing data parallel operations on many-core platforms," *First Workshop on Software Tools for Multi-Core Systems (STMCS)*, Manhattan, NY, 2006, pp. 66–70.
- [6] "Ct: A Flexible Parallel Programming Model for Tera-scale Architectures," at <http://techresearch.intel.com/UserFiles/en-us/File/terascale/Whitepaper-Ct.pdf>
- [7] Daniel P. Friedman and David S. Wise, "Aspects of applicative programming for parallel processing," *IEEE Transactions on Computers*, C-27(4):289–296, April 1978.
- [8] David A. Krantz, Robert H. Halstead, Jr., and Eric Mohr, "Mul-T: a high-performance parallel lisp," in *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pp. 81–90, 1989.
- [9] Guy Blelloch, "Vector Models for Data-Parallel Computing," *MIT Press*. ISBN 0-262-02313-X. 1990.
- [10] Guy E. Blelloch and Gary W. Sabot, "Compiling Collection-oriented Languages onto Massively Parallel Computers," *Journal of Parallel and Distributed Computing*, Vol. 8, Issue 2, pp. 119–134, 1990.
- [11] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zaghera, "Implementation of a Portable Nested Data-Parallel Language," *Journal of Parallel and Distributed Computing (JPDC)*, 21(1), April 1994.
- [12] Guy L. Steele and W. Daniel Hillis, "Connection Machine LISP: Fine-grained Parallel Symbolic Processing," in *Proceedings 1986 ACM Conference on Lisp and Functional Programming*, Cambridge, MA, August 1986.
- [13] Guy L. Steele, "CM-Lisp Technical Report," *Thinking Machines Corporation*, 1986.
- [14] "IBM, APL2 Programming: Language Reference," first ed., August 1984.

- [15] Kenneth E. Iverson, *A Programming Language*, John Wiley & Sons, Inc., New York, 1962.
- [16] Leslie G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, 33(8):103–111, August 1990.
- [17] Manuel M. T. Chakravarty, Gabriele Keller, Roman Lechtchinsky and Wolf Pfannenstiel, "Nepal – nested data parallelism in Haskell," in *Proceedings 7th International Euro-Par Conference*, volume 2150 of Lecture Notes in *Computer Science*, pp. 524–534, Springer-Verlag, Manchester, UK, 2001.
- [18] "NESL: A Parallel Programming Language," at <http://www.cs.cmu.edu/~scandal/nsl.html>
- [19] "Paralation LISP—Embeds the paralation model in Common LISP," Available from *MIT Press*, (800)356-0343.

AUTHORS' BIOGRAPHIES

Anwar Ghuloum is a Principal Engineer with Intel's Microprocessor Technology Lab, working on diverse topics such as parallel language and compiler design, parallel architecture evaluation, optimizing memory system performance, and multimedia applications. Anwar received a B.S. degree in Computer Science and Engineering from the University of California, Los Angeles and a Ph.D. degree in Computer Science from Carnegie Mellon University's School of Computer Science in 1996. Before joining Intel, he co-founded and was the CTO of a fab-less semiconductor startup that designed parallel image and video processors for the consumer electronics market. Prior to that, Anwar developed novel predictive drug design software for early lead optimization using 3D surface pattern recognition techniques for a biotech startup. A recurring theme in Anwar's work has been to bridge high-level application knowledge and low-level parallel architecture constraints with careful parallel language and compiler design to achieve the optimal tradeoffs in productivity and performance. His e-mail is anwar.ghuloum@intel.com.

Terry Smith is a Business Development Manager within Intel's Corporate Technology Group. In his ten years with Intel he has focused on the management of emerging technologies, strategic marketing, and business development. His background includes the Executive MBA Program at the University of Texas-Austin and a B.S. degree in Math/CS from the University of Illinois.

Gansha Wu is a researcher with Intel's Corporate Technology Group. He leads a team researching advanced compiler and runtime technology for future Intel architectures. Gansha has been with Intel for seven years. His e-mail address is gansha.wu@intel.com.

Xin Zhou is a researcher with Intel's Corporate Technology Group. He leads a Ct programmability and workload study. Xin has been with Intel for five years. His e-mail address is xin.zhou@intel.com.

Jesse Fang is the Director and Chief Scientist of the Programming System Lab at Intel/CTG/MTL (Corp. Technology Group/Microprocessor Technology Lab). Before joining Intel in 1995, Jesse was manager of the Hewlett-Packard Research Lab compiler team for Itanium® Architecture. Before that, he was the manager of parallel/vector compilers at Convex and Concurrent Computer Corporation in 1989 and 1986, respectively. Jesse received his Ph.D. degree in Computer Science from the University of Nebraska-Lincoln before he did a post-Doctorate at the University of Illinois, Urbana-Champaign. Jesse received his B.S. degree in Math from Fudan University in Shanghai.

Peng Guo is an engineer in Intel's Corporate Technology Group and works on dynamic compilers. His research interests include dynamic compiler optimizations, and compiler/runtime interactions. He received his Masters degree in Computer Science from the Beijing University of Aeronautics and Astronautics. His e-mail address is peng.guo@intel.com.

Byoungro So is a Senior Research Scientist in Intel's Corporate Technology Group. His research interests include program analysis, high-performance computing, adaptive computing, parallelizing compilers, and performance optimizations. Before joining Intel, he worked for IBM T.J. Watson research center where he developed the Cell compiler and runtime. He received both his M.S. and Ph.D. degrees in Computer Science from the University of Southern California in 1998 and 2003, respectively. His email address is byoungro.so@intel.com.

Mohan Rajagopalan is a Research Scientist in Intel's Programming Systems Lab and leads the parallel runtime research for Ct. His current interests include runtime technologies for forward-scaling on emerging multi-core platforms, new programming models such as for reusable and incremental computation, and whole system optimization. Mohan received his Ph.D. degree from the University of Arizona in 2006. He was the recipient of the 2005 IEEE/IFIP Willam C. Carter Dissertation Award. His e-mail is mohan.rajagopalan@intel.com.

Yongjian Chen is an Engineer in Intel's Corporate Technology Group and works on dynamic compilers. His research interests include parallel language design, compiler/runtime technology to support parallel computation, and parallel architectures. He received his Ph.D. degree from Tsinghua University. His e-mail is yongjian.chen@intel.com.

Biao Chen is an Engineer in Intel's Corporate Technology Group and works on Ct memory management and Ct workload study. His research interests include emerging workloads and memory management. He received his Masters degree in Computer Science from the Beijing University of Aeronautics and Astronautics. His e-mail is biao.chen at intel.com.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel Leap ahead., Intel Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skool, Sound Mark, The Journey Inside, VTune, Xeon, and Xeon Inside are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Intel's trademarks may be used publicly with permission only from Intel. Fair use of Intel's trademarks in advertising and promotion of Intel products requires proper acknowledgement.

*Other names and brands may be claimed as the property of others.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Bluetooth is a trademark owned by its proprietor and used by Intel Corporation under license.

Intel Corporation uses the Palm OS[®] Ready mark under license from Palm, Inc.

Copyright © 2007 Intel Corporation. All rights reserved.

This publication was downloaded from
<http://www.intel.com>.

Additional legal notices at:
<http://www.intel.com/sites/corporate/tradmarx.htm>

THIS PAGE INTENTIONALLY LEFT BLANK

For further information visit:

developer.intel.com/technology/itj/index.htm