Flexible Parallel Programming for Tera-scale Architectures with Ct

Last Revision Date:

April 26, 2007

Primary Authors:
Anwar Ghuloum, Eric Sprangle, Jesse Fang

Key Contributors:
Biao Chen, Yongjian Chen, Zhao Hui Du, Zhenying Liu, Mohan Rajagopalan, Byoungro So, Zhi Gang Wang, Gansha Wu, Xin Zhou

Table of Contents

1.0 New Opportunities, New Challenges

Processors architecture is evolving towards more software-exposed parallelism through two features: more cores and wider SIMD ISA. At the same time, graphics processors (GPUs) are gradually adding more general purpose programming features. Several software development challenges arise from these trends. First, how do we mitigate the increased software development complexity that comes with exposing parallelism to the developer? Second, how do we provide portability across (increasing) core counts and SIMD ISA? Ct is a *deterministic* parallel programming model intended to leverage the best features of emerging general-purpose GPU (GPGPU) programming models while fully exploiting CPU flexibility. A key distinction of Ct is that it comprises a top-down design of a *complete* data parallel programming model, rather than being driven bottom-up by architectural limitations, a flaw in many GPGPU programming models.

1.1 Process & Architecture Trends

CPU designs are increasingly power-constrained. Every silicon process step shrinks linear dimensions by 30%, which has the following implications:

- The area to hold a constant number of transistors is cut in half ($0.7^2 = 0.5$), or alternatively we can build twice the transistors in the same die area
- The capacitance of each transistor shrinks by 30%
- The maximum voltage decreases by approximately 10%
- The switching time of a transistor shrinks by 30% (at maximum voltage)

$\Rightarrow$ Power scales in proportion to the number of transistors * capacitance per transistor * voltage^2 * frequency, which is $2 * 0.7 * 0.9^2 * 1/.7 = 1.62$x per generation

While voltage may drop by slightly more than 10% per generation or capacitance may drop by slightly more than 30%, this does not substantially affect the power scaling trends. As a consequence, power efficiency is the ultimate goal because power consumption is the ultimate limiter to improving computational performance in silicon technology: silicon scaling improves transistor density by 50% per generation, but only reduces power 20% per generation. The first order design concern for Tera-scale architectures is to improve computational power efficiency (MIPS/Watt) over traditional processors on parallel workloads.

While increasing core count as a means of performance scaling will continue, the hardware and associated power involved with uncovering parallelism in traditional out-of-order processors must be addressed. There are several techniques for exposing the parallelism to software explicitly, including long vector ISA and simultaneous multi-threading.

Each of these techniques has strengths and weaknesses depending on the type of code that are intended to be run. Given the type of codes that we believe will be important in the future (graphics rendering, media, etc, which have streaming memory behavior) architectures will have to strike the right balance of these techniques. Simultaneous multi-threading is powerful technique for hiding memory latency in applications with poor locality. Future architectures will rely on longer SIMD vectors (e.g. 4, 8, 16 elements per register) to improve power efficiency but there is generally a "harder" limit on the effectiveness of increasing vector length. Increasing vector length creates inefficiencies for algorithms that inherently use shorter or unaligned vectors. For example, we expect it is appropriate to scale the number of cores at about 2x per process generation (tracking Moore's Law), but the SIMD width will scale much more slowly. Nevertheless, this presents the software developer another architectural variable to adapt to.

1.2 Programming Tera-scale Architectures

ISVs are excited by the peak throughput benefits of future Intel architectures, but at the same time they are concerned about the burden on the application developer to explicitly exposing the parallelism to the hardware. Using threads and vector intrinsics gives maximum flexibility to the programmer, but at great expense of programmer productivity, application portability and scalability. Moreover, evolving vector lengths and core counts are going to create performance scaling problems for software developers. Backwards compatibility in CPUs guarantees functional correctness, but does not address performance scaling, which may regress as architecture evolves!

GPU hardware and software vendors have taken steps toward solving this problem. Languages for GPUs relieve the programmer of having to think about threads or SIMD width by supporting, for example, the DirectX programming model. In this model, a single data element (pixel) of a collection (display surface) is processed independent of neighboring data elements (pixels). This simpler model of computation reflects the underlying GPU architecture and is often called *streaming data parallelism*.

GPU programming models are constrained in a way that the compiler and run-time can reason about the application and extract the parallelism automatically. Examples of this include DirectX, CUDA, and Cg. If the programmer can reformulate the application to work under GPU constraints, the compiler/run-time can do the rest automatically. However, reformulating the application to fit under these constraints often requires considerable programmer effort, and can result in significantly less efficient software algorithms. For example, it is difficult to efficiently operate on linked lists or compressed data structures, so applications that would naturally like to use these types of algorithms need to be reformulated to use algorithms more consistent with (GP)GPU models.

Intel architecture is more general purpose than GPU and other coprocessor architecture. Unlike GPUs, Intel architectures have

1. Inter-core communication through substantial, coherent cache hierarchies
2. Efficient, low latency thread synchronizations across the entire processor array
3. Narrower effective SIMD width

More general purpose hardware allows Intel architecture to run more general purpose software algorithms (for example, algorithms that employ linked lists). So while Intel architecture can run applications written to use a GPU programming model, these applications are more constrained than necessary. That is, it makes sense to define a constrained programming model so the compiler and run-time can extract the parallelism, but one less constrained than most GPGPU models is highly desirable so applications do not have to be reformulated as drastically. At a high level, this is the goal of Ct: define a constrained programming model that efficiently and portably targets highly parallel general purpose cores, like Intel multi-core and Tera-scale systems. To ease incremental adoption, Ct seamlessly extends C/C++ and can be used with legacy threading APIs.

1.3 Ct: Nested Data Parallelism

It often is convenient for the programmer to think of the computing resources provided by a multi-core CPU as an engine for data-parallel computation. The basic idea is that applications *exhibit a lot of parallelism through operations over collections of data*. Abstracting the underlying hardware threads, cores, and vector ISA as computation over collections of data greatly simplifies the task of expressing parallelism in an architecture independent fashion. Ct provides a nested data-parallel abstraction that is initially familiar to most programmers, but provides extended functionality to address irregular algorithms.

Irregular algorithms are broadly defined as those that require:

- Dynamic data types, such as sparse matrices, linked lists, or trees;
- A high likelihood of contended synchronization, such as reductions and prefix-sums in which elements of a collection are “summed” using a combining operator;
- Moderate control flow, such as well-structured conditional nests, nested loops, and recursive functions.

Intel multi-core architecture (including Tera-scale architecture) addresses these requirements efficiently, whereas GPUs generally do not. Ct aims to address the software stack by providing a data parallel model that supports irregular algorithms, whereas GPGPU programming models do not. An important benefit of Ct for the software developer is that it scales *forward* with increasing core count and vector ISA width. For example, a Ct application will scale from dual- and quad-core systems to Tera-scale systems.

1.3.1 The Importance of Determinism

Like many of its flat data-parallel brethren, Ct is *deterministic*. Determinism guarantees that program behavior is identical on 1 core and many cores. This essentially eliminates an entire class of programmer errors, namely data races. Ct also provides a predictable high-level programming model. This means that the average programmer can use Ct operators with a basic understanding of the cost and scalability of its use. This is difficult to achieve for unconstrained threading models.

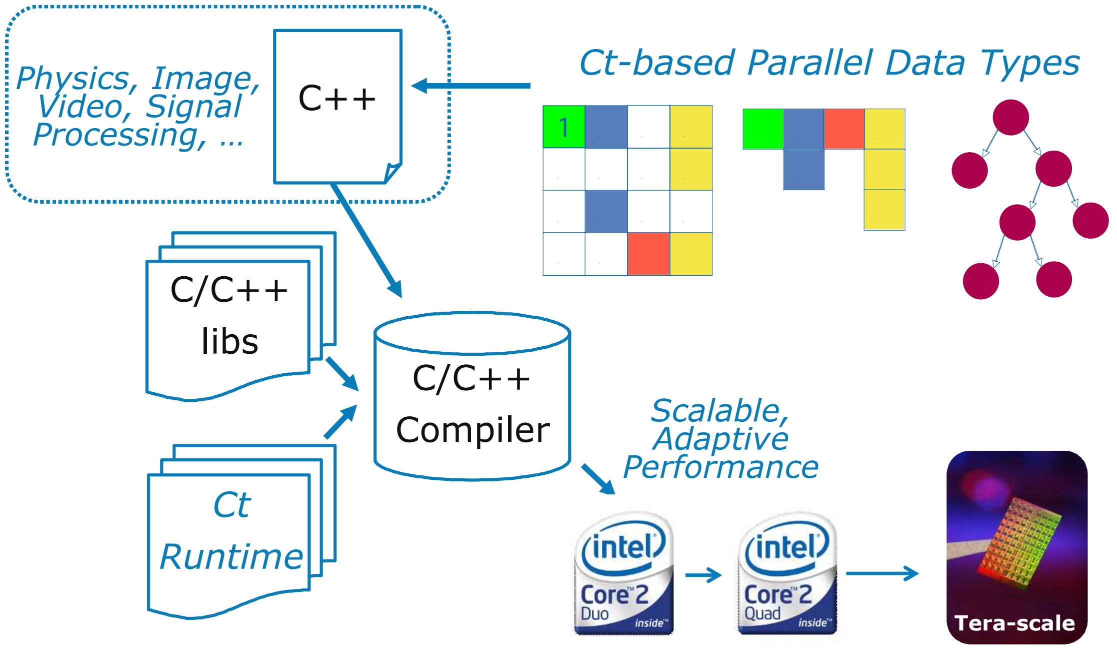

Physics, Image, Video, Signal Processing, …
C++
Ct-based Parallel Data Types
1
C/C++ libs
C/C++ Compiler
Ct Runtime
Scalable, Adaptive Performance
intel Core 2 Duo inside
intel Core 2 Quad inside
Tera-scale

Figure 1-1 The Ct API in the software development process.

An important theme for Tera-scale programming models will be combining deterministic semantics with powerful programming abstractions. While this is a difficult task, Ct represents a design point that combines powerful, high-performance data parallel constructs with completely deterministic behavior.

2.0 An Example: Sparse Matrices

2.1 Why Sparse Matrices are Hard

Many useful data structures have unusual (or irregular) organization because of efficiency of such representations. For example, dense (or fully populated) matrices are required when the problem formulation dictates that most elements of the matrix are significant. However, it is often the case that the particular algebraic formulation of a problem sparsely populates elements in the matrix with meaningful values. An example is large scale physics simulation. 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. This type of data compression is essential in desktops, mobile, and console gaming platforms.

Dense representations simplify the parallelization of such code. Walking through the elements in a dense representation is often performed in some regular patterns, such as column or row order. This means that the control paths and data access patterns are very predictable (one feature of regularity in an algorithm). For example, the code in Example 2-1 can be used to walk over a dense matrix A in row-order.

Example 2-1: Dense matrix traversal

```
for (row = 0; row < row_num; row++) {
   for (col = 0; col < col_num; col++) {
   …touch elements of A[row][col]…
   }
}
```

There are generally two things that make this a relatively manageable exercise in parallelization:

1. Depending on what algorithm is being implemented there may be parallelism in one or both of the enclosing loops. For large matrices, these loops provide sufficient parallelism that can be trivially decomposed into parallel sub-loops.
2. The data accessed within one or both of these loops is distinct (or *independent*, in compiler terms).

Sparse matrices are much more difficult. There are many forms of sparse matrix, we will touch on two: Compressed Sparse Column (CSC) and Compressed Sparse Row (CSR). Both pose unique challenges. The basic idea of CSC and CSR is to only store the non-zero elements of the matrix in either column or row order, respectively. With each non-zero element, the programmer will also store the row or column index.

Consider the sparse matrix in Example 2-2. In CSR and CSC formats, the matrix would be stored as three vectors, the nonzero values, the row or column pointers, and the column and row indices, respectively. Schemas for traversing the two representations are shown in Example 2-3.

Example 2-2: Sparse matrix representation and traversal

```
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 CSR Representation of A:

```
Values  = [1 2 3 4 5 6 7 8 9]
ColIdx  = [1 0 2 3 1 4 1 4 2]
RowP    = [0 1 4 6 8 9]
for (row = 0; row < row_num; row++) {
   for (elt = RowP[row]; elt < RowP[row+1];
elt++) {
      int col = ColIdx[elt];
      …touch elements of A[row][col]…
   }
}
```

The CSC Representation of A:

```
Values  = [2 1 5 7 3 9 4 6 8]
RowIdx  = [2 1 3 4 2 5 2 3 4]
ColP    = [0 1 4 5 6 7 9]


for (col = 0; col < col_num; col++) {
   for (elt = ColP[col]; elt < ColP[col+1];
elt++) {
      int row = RowIdx[elt];
```

```
        …touch elements of A[row][col]…
    }
 }
```

There are two things that make this difficult to parallelize efficiently.

1. The inner loop in both cases has a varying and unpredictable trip count. For example, this might be determined by where a game player is looking in a particular scene in a game. This makes it difficult to predict the workload for each inner loop invocation, thus it is difficult to balance the workload among threads.
2. There is an indirection through the Column or Row index array that will create aliases and dependences in most computations that might use sparse matrices.

Consider sparse matrix vector product (SMVP), a common kernel in gaming and RMS applications. In SMVP, a sparse matrix is multiplied by a vector. Like dense matrix vector multiplication, sparse matrix vector product requires taking the inner product of each row of the matrix with the vector. For the CSR sparse matrix, the loop traverses the data in row order, similar to the dense computation. For the CSC sparse matrix, it is simpler to traverse the data in column order, updating the result vector when non-zero row elements occur.

It is worth observing some of the broader patterns of these computations to comprehend the implications for parallelism:

- In CSR, the expression *vec[ColIdx[elt]]* denotes that *vec* is permuted by the contents of *ColIdx*. This looks like a *gather* operation in GPUs.
- In CSC, the expression *vec[col]* implied that each element of *vec* must be replicated (*ColP[col+1]-ColP[col]*) times. This can be viewed as a special kind of *gather* operation, but more complex than what is typically supported in GPU hardware.
- In CSR, the left-hand side expression "*product[row]*+= …" denotes that we are summing together (or *reducing*) the all the elements computed in the right-hand side of that expression for the inner loop.
- In CSC, the expression "*product[RowIdx[elt]]* +=…" implies that we are performing a what is called a *combining-send,* or alternatively a *multi-reduction* or *combining-scatter*.

Note that observations 3 & 4 in this list are similar, except that the data is effectively pre-sorted by destination for the CSR form. That is, the inner product to compute an element of the result for CSR occurs entirely within one invocation of the inner loop, whereas it takes place across many invocation of the inner loop for CSC.

We can now create a conceptual parallel pattern for computing these particular sparse matrix computations: *first, permute the vector*; *second, perform an element-wise product, then finally, perform some flavor of reduction.*

2.2 Why Sparse Matrices are Particularly Hard for GPUs

Special purpose processors often lack many of the mechanisms required for efficient implementation of sparse matrix kernels. For example, GPUs lack the basic facilities required to support efficient, low-latency cross-chip inter-thread communication and synchronization. Cache coherence provides a vehicle for core-to-core communication, while high-performance interconnects provide low-latency.

Inter-thread synchronization and communication of partially computed results is essential to support collective communication primitives such as reductions and prefix-sums. For example, to reduce the values of an array, the standard algorithm is to partition the elements of the array among threads, reduce the values locally, and then combine the partial results from each thread. To combine the partial results from each thread, they must be exchanged among the threads. For a general purpose multi-core architecture, this requires the use of coherent memory locations and locks. No external memory bandwidth is consumed.

For a GPU, the algorithm is quite costly in terms of memory bandwidth. The only mechanism for inter-thread synchronization and communication is through access to external memory. The GPU reduction algorithm must then repeatedly write and then read partial results to and from external memory. For an array of length *n*, using a progressive combining scheme whereby adjacent pairs of elements are combined together with the result written out in each phase. The first phase reduces to *n/2* elements, the second to *n/4* elements, and so on. In all, this requires *O(log n)* passes through memory, consuming *O(n*log n)* memory bandwidth.

Sparse-matrix operations require more complex flavors of collective communication primitives, like *segmented reductions*, *multi-reduce*, and *prefix-sum*. Operations like these are significantly more complex on special purpose processors, requiring additional passes through memory. This is a principal reason why such primitives are historically restricted or unavailable on GPGPU programming models.

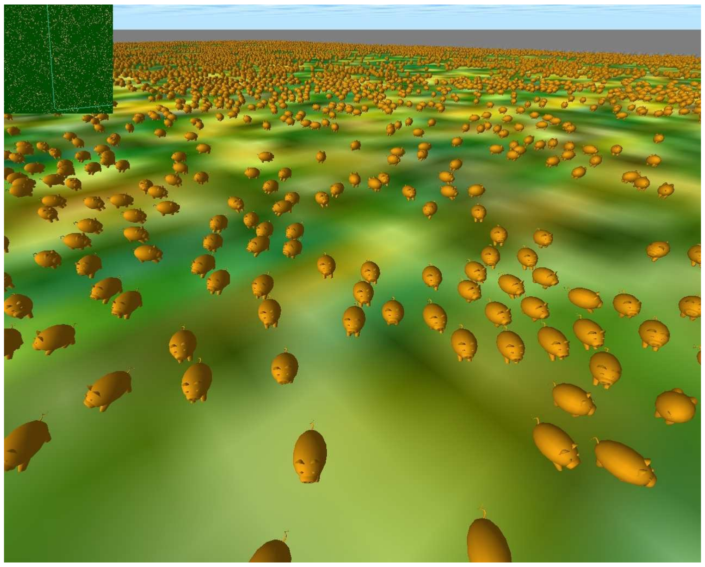


Figure 2-1 Ct-based physics simulation (requires sparse matrices): "Golden Pigs"

2.3 Sparse Matrices with Ct

Ct introduces a new (template-style) polymorphic type, called a TVEC. TVECs are write-once vectors that reside in a vector space segregated from native C/C++ types. For example, a vector of non-zeros for the stiffness matrix in a cloth simulation may be declared as:

```
CCtVEC<F64> nonzeros;
```
[1]

[1] For cloth simulation, we are likely to use a block sparse symmetric matrix, but we elide these details for simplicity of presentation.

The column indices for each nonzero and row sizes for the sparse matrix (assuming a Compressed Sparse Column representation) can be represented similarly:

```
CCtVEC<I32> RowIdx;
CCtVEC<I32> ColP;
```

The type of primitives used for this particular sparse matrix-vector product falls into three of the Ct operator categories:

- *Element-wise* operators that support simple unary, binary and n-ary operators, such as addition, multiplication, etc. For example, to perform an element-wise multiplication of two vectors:
  ```
  CCtVEC product = ctMul(nonzeros, expv);
  ```
  …or, using operator overloading:
  ```
  CCtVEC product = nonzeros*expv;
  ```
- Collective communication operations like reduction, prefix-sum, or combining-send. Multi-core CPUs supports these very efficiently, though many accelerators (including GPUs) support only a few efficiently. For example, to perform an indexed reduction, or a combining-send, of a vector, we use a multi-reduce primitive:
  ```
  CCtVEC innerproduct =
  ctAddMultiReduce(product,RowIdx);
  ```
- Permutation operations which allow both structured and unstructured reordering and replication of data. For example, to create a new vector expv comprised of a variable number of copies (denoted by each element of cols) of each element of v:
  ```
  CCtVEC expv = ctDistribute(v,ColP);
  ```

Putting it all together, we can write a sparse matrix-vector product as in Example 2-3.

Example 2-3: Sparse matrix vector product with Ct

```
 CCtVEC<F64> ctSparseMatrixVectorProductCSC(
  CCtVEC<F64> Values,
  CCtVEC<I32> RowIdx,
  CCtVEC<I32> ColP,
  CCtVEC<F64> v) {
    CCtVEC expv = ctDistribute(v,ColP);
    CCtVEC product = Values*expv;
    CCtVEC result =
ctAddMultiReduce(product,RowIdx);
    return result;
```

```
}
```

Ct provides several levels of abstraction below the high-level API for increasing levels of sophistication/expertise in the programmer. Lower layer levels expose task granularity and machine-width independent vector ISA intrinsics and optimizations.

3.0 The Ct API

3.1 Ct Vectors: TVECs

The basic type in the Ct API is the TVEC, a managed parallel vector. TVECs are allocated and managed in a segregated memory pool to ensure the safety of parallel operation on TVECs. Normal C and C++ data structures are generally unsafe because of unrestricted effects and use of aliases.

Data must be explicitly copied into and out of vector space and the only operators allowed on TVECs are Ct operators, which are functionally pure. It is important to note that TVECs are logically passed around by value. This particularly property guarantees the safety of parallelism and the aggressive optimizations that make parallelism efficient.

The base types of TVECs are drawn from a set of typical pre-defined scalar types. A TVEC variable may be declared without an instantiated base type, but the compiler must be able to infer the type. Examples of base types include I32 (32-bit integer), I64 (64-bit integer), F32 (Float), F64 (Double), and Bool (Boolean). Ct also includes a C struct-like base type for user-defined base types (such as color pixels) and fixed size arrays.

The internal representation of TVECs is opaque to the programmer and may include meta-data that is useful to the runtime. A TVEC may be declared as:

```
CCtVEC<I8> Red;
CCtVEC<F32> Xes;
```

Values in native C/C++ memory space are explicitly copied into and out of Ct managed vector space using ctCopyin and ctCopyout operators:

```
Red = ctCopyin(CRed, Height*Width, I8);  // Red
← CRed
ctCopyout((void*)CXes,Xes);  // CXes ← Xes
```

There are several flavors of each of these operators for different data types and shapes.

3.2 Ct Operators

Ct operators are logically side-effect free from the programmer's perspective. As such, each Ct operator logically returns a new TVEC. (Note that the C++ operator overloading is used to write "cleaner" or more readable code.)

```
ScaledRed = Red*0.5; // ScaledRed←a new TVEC
```

The API encompasses a broad range of rich functionality. Within each of the classes of facility, element-wise, collective communication, and permutation operators, there are many subclasses of operations, each defined over all TVEC types. Each Ct operator generally has the form ctOpClass, where Op is the particular flavor of operator and Class is the operator class. For example, ctAddReduce is the reduction operator using addition.

3.2.1 Element-wise Operators

Element-wise operators are typically referred to as "embarrassingly" parallel, requiring no interactions between the computations on each vector element. In functional languages, these can be implemented with map operations, while in OpenMP they can be implemented as parallel for loops. An example of an element-wise operation is the addition of two vectors:

```
CCtVEC 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).

3.2.2 Collective Communication Operators

Collective communication operators tend to provide distilled computations over entire vectors and are very coordinated. While they have a high degree of interference, they can be structured so that there is parallelism in colliding writes and typically scale in performance linearly with processor count with little or no hardware support. 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 are implemented via list homomorphisms.

There are two kinds of collective communication primitives in general, though there are several flavors of each depending on the type of vector being operated on. Broadly, the two fundamental types of operations are 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, a ctAddReduce sums over all the elements of a vector if the vector is flat. More concretely, `ctAddReduce([1 0 2 -1 4])` yields 6. Likewise, `ctAddPrefix([1 0 2 -1 4])` yields `[0 1 1 3 2]`. If the vector is nested, the behavior is described later in this document.

3.2.3 Permutation

A permutation operator in Ct is any operator that requires moving data from its original position to a different position. An example of this is a gather operation, which uses an index array to collect values of a vector in a particular order. Permutations run the gamut from arbitrary permutations with arbitrary collisions (where 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 ctPack, which uses a flag vector to select values from a vector in the source vector order. Hardware can typically support this fairly efficiently.

3.2.4 Nested Vectors

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

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 some attributes (like a threshold value).

Ct operators work on nested TVECs seamlessly. Element-wise operators behave the same as they do for flat vectors. For example, `ctAdd([[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 primitives.

Collective communication primitives applied to nested TVECs "respect the boundaries" of the subvectors by applying the operator to each subvector independently. For example, `ctAddReduce([a b c d e f])` yields the singleton vector `[a+b+c+d+e+f]`, while `ctAddReduce([[a b][c d e f])` yields the two element vector `[a+b c+d+e+f]`.

The internal representation of a nested vector is opaque to the programmer. The nesting structure is accessible through several facility routines, but this generally should not be necessary.

Other vector types supported by Ct are indexed vectors, where there are indices associated with each value. This is related to the irregular nested vectors

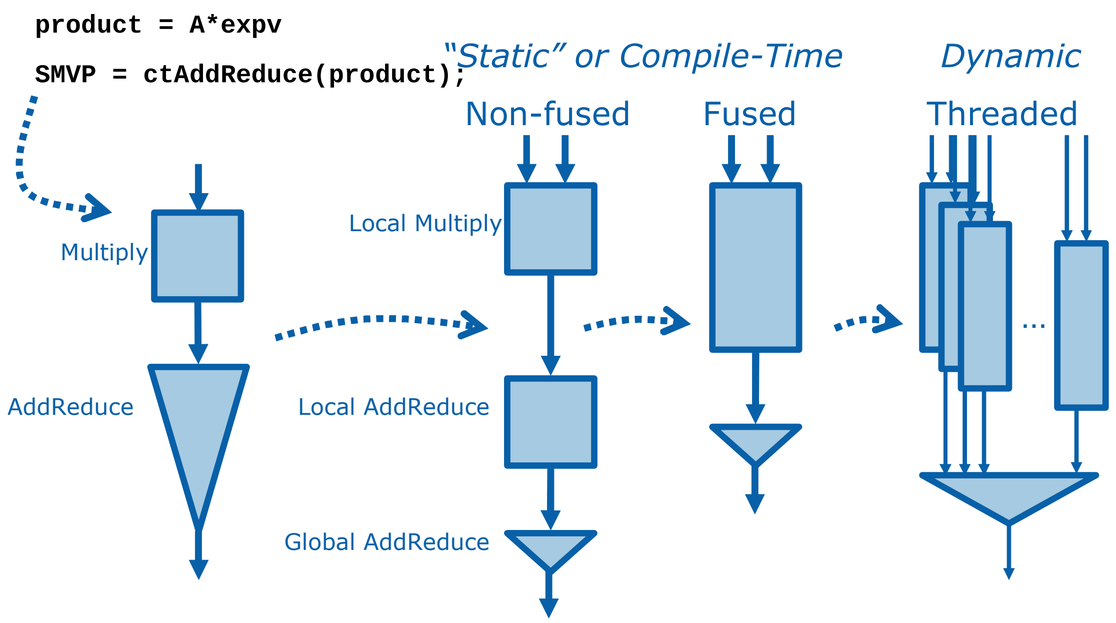
3.3 How Ct is Implemented

Earlier generations of data parallel programming models and languages benefited from the extraordinarily large data sets and highly parallel (high bandwidth) memory systems. This meant that the optimization pressure on task granularity was not as critical as simply identifying parallel regions of code and the basic operator implementations. For Tera-scale architecture, such implementations would yield code that is memory bandwidth limited and overly burdened with threading overhead. The computation power of Tera-scale must be coupled to intelligent optimization that maximized the amount of calculation per memory operation.

The top priority of the Ct compiler and runtime is to minimize threading overhead and make effective use of memory bandwidth. To accomplish this, Ct utilizes a fine-grained, data-flow threading model. Essentially, the Ct computation is decomposed into a task dependence graph that is optimized by the compiler by merging similar tasks into coarser-grained tasks.

The task dependence graph is comprised of data parallel *sub-primitives*. These sub-primitives are essentially the building blocks of data parallel computing, comprising *local* phases of computation that entail no inter-processor synchronization and *global* phases that perform structured write combining and synchronization. Similar sub-primitives can usually be fused together into coarser grained tasks. This simultaneously increases task granularity and locality of data access, minimizing off-chip memory accesses.

Another distinction of Ct's approach is that threading decisions are made dynamically. Each task is represented by a spawn point, but the precise number of sub-tasks created is dependent on both the number of cores available and the size of the underlying vector being processed. The Ct runtime is highly adaptive to varying data sizes and core loads.



```
product = A*expv
SMVP = ctAddReduce(product);
```

Figure 3-1 How Ct is compiled.

4.0 Concluding Remarks: The Future of Ct

Ct implements many more features than described here and enables several other key features that will be implemented in the coming months. An important future feature is *deterministic task parallelism*. As previously observed, determinism is a critical property for programmer productivity, guaranteeing that program execution is functionally predictable. Through determinism, data races are entirely eliminated as a class of programmer errors.

The underlying threading model supports a fine-grained, adaptive dependent tasking model. While this is used to implement the higher-level data parallel constructs, revealing implicit task parallelism, Ct will include constructs for deterministic task parallelism.

Ct supports a high-level performance model that can be used by the average programmer to guide algorithmic choice. For example, element-wise operators generally scale linearly in performance on Intel multi-core and Tera-scale architecture for large vectors. If the operations are used for small vectors, the underlying runtime knows not to use as many cores for the computation, mitigating threading overhead. Similarly, collective communication operators generally scale linearly with core count, but have an addition cost associated with synchronization. The synchronization patterns vary by architecture, but are generally asymptotic in core count, not in vector size. So, for large vectors, the linear scaling component tends to dominate.

Ct provides a virtual laboratory with which to experiment with more exotic Tera-scale programming features, such as lossy, real-time, and adaptive computation. Some of these ideas are used on an *ad hoc* basis in many high performance algorithms and applications, notably those with real-time constraints, like media and gaming. A key objective of Ct is to create a framework in which such performance breakthroughs (sometimes pejoratively and unfairly misunderstood as *hacks*) are supported systematically in a framework

5.0 Selected Bibliography

Guy Blelloch. Vector Models for Data-Parallel Computing. MIT Press. ISBN 0-262-02313-X. 1990.

Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a Portable Nested Data-Parallel Language. Journal of Parallel and Distributed Computing (JPDC), 21(1), April 1994.

Manuel M. T. Chakravarty and Gabriele Keller and Roman Lechtchinsky and Wolf Pfannenstiel. Nepal --- nested data parallelism in Haskell. In R. Sakellariou, J. Keane, J. R. Gurd, and L. Freeman, editors, Proc. 7th International Euro-Par Conference, volume 2150 of Lecture Notes in Comput. Sci., pages 524--534. Springer-Verlag, 2001.

Allan L. Fisher , Anwar M. Ghuloum, Parallelizing complex scans and reductions, Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, p.135-146, June 20-24, 1994, Orlando, Florida, United States.

Daniel P. Friedman and David S. Wise, Aspects of applicative programming for parallel processing. IEEE Transactions on Computers, C-27(4):289–296, Apr. 1978.

Anwar M. Ghuloum, Allan L. Fisher, Flattening and parallelizing irregular, recurrent loop nests, ACM SIGPLAN Notices, v.30 n.8, p.58-67, Aug. 1995.

Matthew Hammer, Umut A. Acar, Mohan Rajagopalan, Anwar Ghuloum, A Proposal for Parallel Self-Adjusting Computation, In Proceedings of the Workshop on Declarative Aspects of Multicore Programming (DAMP 2007), January 2007

Robert M. Keller, The Future of Meta-Programming in Parallel Languages Proceedings of the Forum on Parallel Computing Curricula, Wellesley College, 1995.

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, pages 81–90, 1989.

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.

Leslie G. Valiant. A bridging model for parallel computation. Communications of the ACM, 33(8):103--111, August 1990.

6.0 Appendix: Ct Coding Examples

6.1 Image Processing

Image processing is generally fairly straightforward in data parallel programming models, even those that are flat. There is a slight difference in coding style between the "streaming" style of local kernel specification and the global data parallel approach. In the kernel style, only the local computation is specified and is assumed to be applied at every pixel (similar to the graphics model). In the data parallel style, the entire image is operated on at once, though this can be relaxed through the use of Ct's generic operators (e.g., ctGenBinary). However, the code is not generally substantially more or less compact with either approach, as illustrated in the color conversion code below.

```
CCtVEC<CT_F32> ctColorConvert(CCtVEC<CT_F32> rchannel, CCtVEC<CT_F32> gchannel,
                              CCtVEC<CT_F32> bchannel, CCtVEC<CT_F32> achannel,
                              F32 a0, F32 a1, F32 a2, F32 a3)
{
    return (rchannel * a0 + gchannel * a1 + bchannel * a2 + achannel * a3);
}
```

Convolutions are somewhat more substantial and broadly useful in both image processing and more general signal processing domains. Because convolutions, require a neighborhood of pixels to compute the filter, the source image is "shifted" about to place the required pixel at the computed pixel. These shifts are logically creating new values, though in this case, the compiler and optimizer trivially optimize away any copying and simply refer to a single source image.

```
CCtVEC<F32> Convolve2D3x3(CCtVEC<F32> pixels, I32 channels, CCtVEC<F32> kernel) {
  CCtVec respixels;

    // directions[m][n] is a constant TVEC of size 2 with values {m-1, n-1}
  respixels += ctShiftPermute(pixels, directions[0][0]) * kernel[0][0];
  respixels += ctShiftPermute(psixels, directions[0][1]) * kernel[0][1];
  respixels += ctShiftPermute(pixels, directions[0][2]) * kernel[0][2];
  respixels += ctShiftPermute(pixels, directions[1][0]) * kernel[1][0];
  respixels = pixels * kernel[1][1];
  respixels += ctShiftPermute(pixels, directions[1][2]) * kernel[1][2];
  respixels += ctShiftPermute(pixels, directions[2][0]) * kernel[2][0];
  respixels += ctShiftPermute(pixels, directions[2][1]) * kernel[2][1];
  respixels += ctShiftPermute(pixels, directions[2][2]) * kernel[2][2];

  return respixels
}
```

6.2 Sparse Linear Solvers

Sparse linear solvers are quite common in high performance code in physics simulation, aspects of machine learning, and many RMS applications. A common technique used is the preconditioned conjugate gradient method, illustrated in Ct below. Note that the key kernel in use here is a CSR sparse matrix vector product.

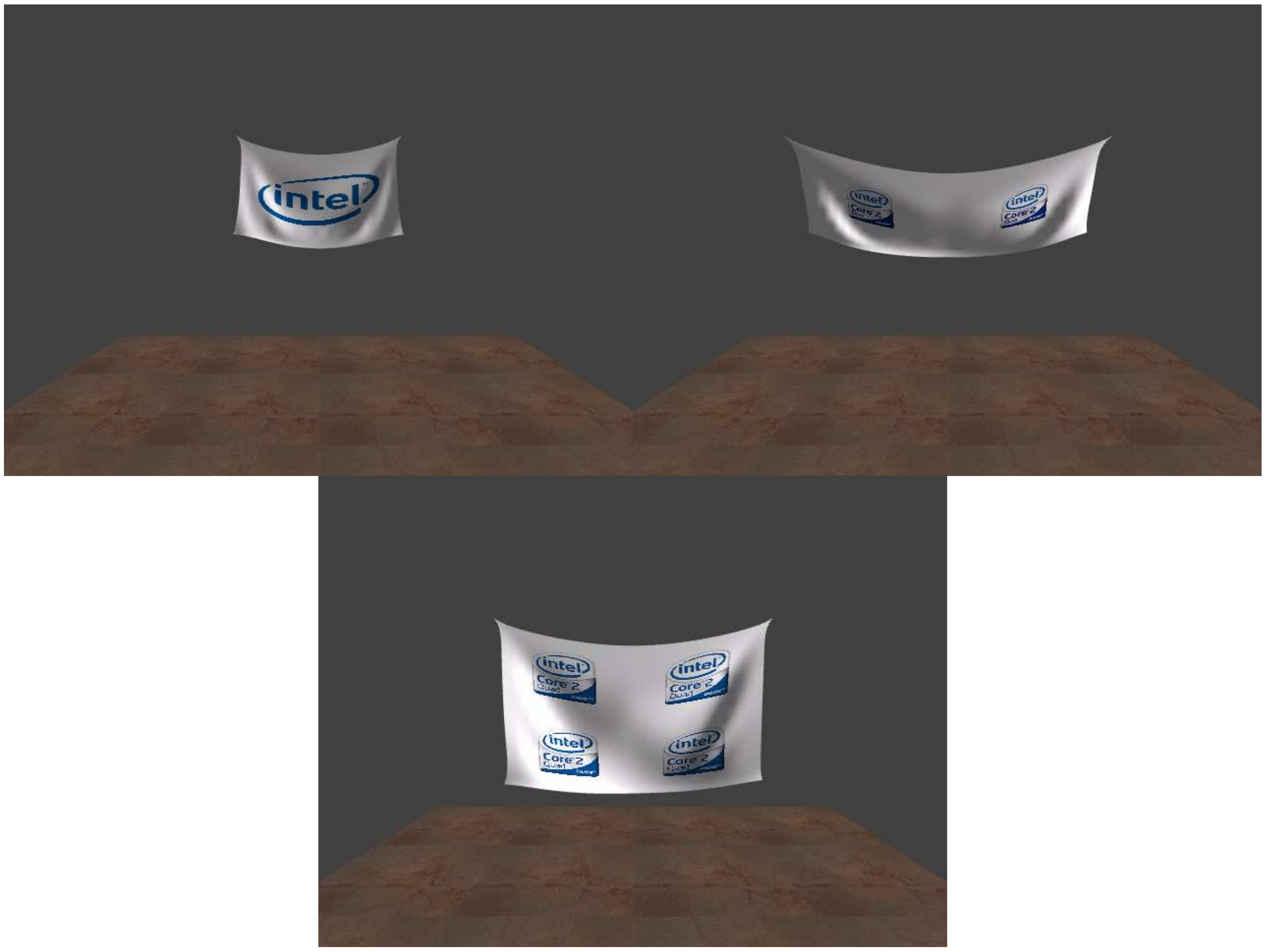


Figure 6-1 Ct-based Cloth simulation on Core 2, Core 2 Duo, Core 2 Quad

First, we define a C structure to contain the linear system.

```
typedef struct {
    CTSparseMatrix          A;
    CCtVEC                  b;
    Float                   e;          // Epsilon

    CTSparseMatrix          pM;       //Preconditioning matrix.
    CTSparseMatrix          pMinv;    //Inverse preconditioning matrix.
} CTLinearSystem;
```

The solver uses the sparse matrix vector product kernel (called ctSmvMul below).

```
CCtVEC<F64> ctPCG(CTLinearSystem &lsys, CCtVEC x0){
    int i = 0;
    F64 alpha, delta0=0, delta1;
    CCtVEC v_x  = x0;
    CCtVEC v_r  = lsys.b - ctSmvMul(lsys.A, v_x);
    CCtVEC v_pr = ctSmvMul(lsys.pMinv, v_r);
    delta1 = ctAddReduce(v_r * v_pr);
    while( (delta > delta0 * lsys.e) && (i < IterationMax)){
        CCtVEC v_q   = ctSmvMul(lsys.A, v_pr);
        alpha        = ctAddReduce((v_pr * v_q));
        alpha        = delta/alpha;
        v_x          = v_x + (v_pr * alpha);
        v_r          = v_r - (v_q * alpha);
        CCtVEC v_s   = ctSmvMul(lsys.pMinv, v_r);
        delta0       = delta1;
        delta1       = ctAddReduce(v_r * v_s);
        v_pr         = v_s + (v_pr * (delta1/delta0));
        i++;
    }
    return v_x;
}
```

The cloth simulation requires a bit more code to integrate forces in the system and detect collisions. Collision detection, in particular, is extremely challenging to parallelize for flat data parallel systems, but simplified greatly through nested data parallelism.

6.3 Sorting

While sorting is typically used as an illustrative algorithm, the kernel below forms part of the implementation for quick KD-tree construction in Ct. The quicksort is simpler to use to illustrate the tradeoffs in implementation.

The problem with recursive sorting (and divide and conquer algorithms, in general) is that the (superficially) data parallelism is maximized at the root of the algorithm's call graph and minimized at the leaves. Similarly, task parallelism is (superficially) minimized at the root and maximized at the leaves. Taking the task versis data parallelism view of quicksort, it appears to be difficult to fit in a single programming model. The approach is illustrated in the code and illustration below.

```
CCtVEC<F64> ctQsort(CCtVEC<F64> Keys) {
  CCtVEC <F64> pivot, lowerKeys, pivotKeys, upperKeys;
  CCtVEC<Bool> pivotFlags;
  I32 pivot;

  if (ctLength(Keys) == 0)
    return Keys

    pivot = ctExtract(Keys, 0)

    pivotFlags = ctLessThan(Keys, Pivot);
    lowerKeys = ctPack(Keys, pivotFlags);
    pivotFlags = ctEqual(Keys, Pivot);
    pivotKeys = ctPack(Keys, pivotFlags);
    pivotFlags = ctGreaterThan(Keys, Pivot);
    UpperKeys = ctPack(Keys, pivotFlags);

    return ctCat(ctQsort(lowerKeys),
      ctCat(pivotKeys,ctQsort(upperKeys));
}
```

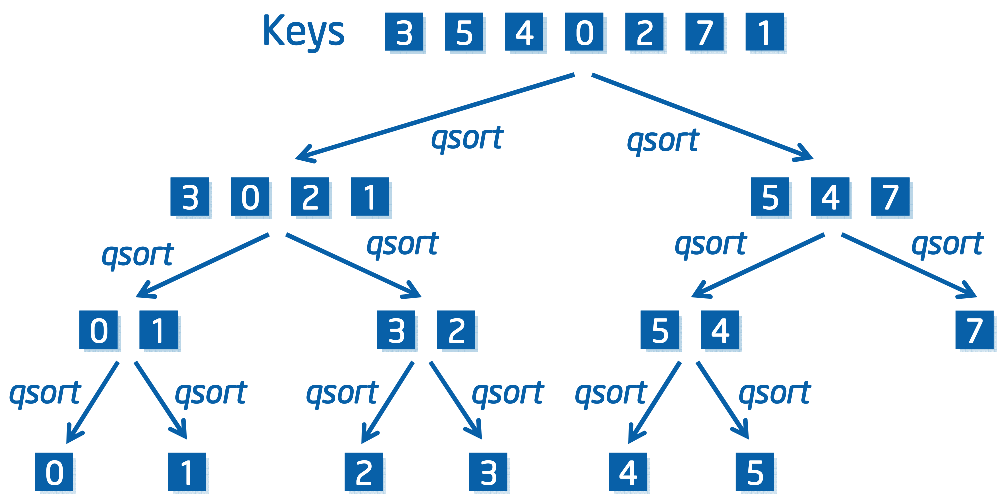


Figure 6-2 Decreasing data parallelism and increasing task parallelism in Quicksort.

Nested data parallelism unifies data parallelism and divide-and-conquer task parallelism. Sub-tasks are not created as parallel threads, but rather are manifest as partitions in a nested vector. We use ctPartition to create these sub-partitions, and then recurse on the entire vector. The power of nested data parallelism for this type of irregular control flow is clearly illustrated through the resulting unification of data and task parallelism.

```
CCtVEC<F64> ctQsort(CCtVEC<F64> Keys, CCtVEC<Bool> Mask) {
  CCtVEC<F64> pivot, partitionedKeys,
  CCtVEC<Bool> newMask;
  CCtVec<I32> pivotPartitions
  if (ctEquals(ctOrReduce(ctOrReduce(Mask)), 0)) // since mask is nested, have to do this twice
    return ctFlatten(Keys);

    pivot = ctExtract(Keys, ctNewVector(ctNumPartitions(Keys),0,I32));

    pivotPartitions = ctCompare(Keys, Pivot, Mask);
    partitionedKeys = ctPartition(Keys, pivotPartitions);
    partitionedMask = ctPartition((Mask && pivotPartitions), pivotPartitions);

    return ctQsort(partitionedKeys, partitionedMask);
}
```

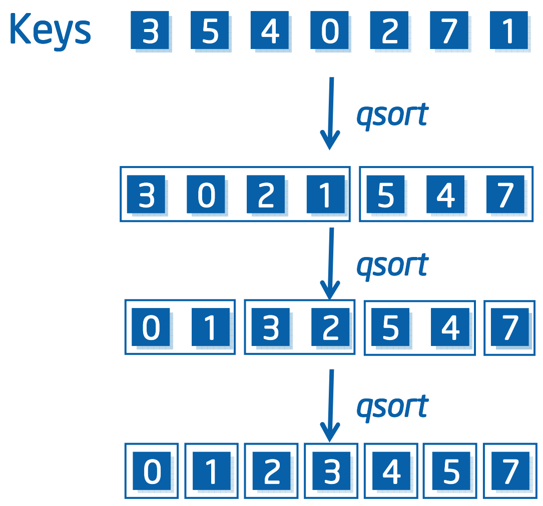


Figure 6-3 Unifying Quicksort's data and task parallelism in Ct via nested data parallelism.

7.0 Appendix: Typical Ct Operators

Table 7-1 A selection of typical Ct Operators

Facilities

Managed Vector/Native Space Copying

ctCopyIn, ctCopyin2D, ctCopyin3D, ctCopyout

Vector Generators

ctCat, ctRepeat, ctReplicate, ctReplace, ctIndex, ctCopy, ctNewVector

Vector Utilities

ctExtract, ctCopy, ctLength

Nested Vectors

ctNewNestedVector, ctApplyNesting, ctCopyNesting, ctSetRegular2DNesting, ctSetRegular3DNesting, ctGetNesting, ctGetNestAsVec

Element-wise

Vector-Vector

ctAdd, ctSub, ctMul, ctDiv, ctEqual, ctMin, ctMax, ctMod, ctLsh, ctRsh, ctGreater, ctLess, ctGeg, ctLeq, ctNeq, ctIor, ctAnd, ctXor, ctPower, ctDivTan, ctSelect, ctGenBinary

Vector-Scalar

ctAddVectorScalar, ctSubVectorScalar, ctSubScalarVector, ctMulVectorScalar, ctDivVectorScalar, ctDivScalarVector, ctEqualVectorScalar, ctMinVectorScalar, ctMaxVectorScalar, ctModVectorScalar, ctLshVectorScalar, ctRshVectorScalar, ctGreaterVectorScalar, ctLessVectorScalar, ctGeqVectorScalar, ctLeqVectorScalar, ctNeqVectorScalar, ctIorVectorScalar, ctAndVectorScalar, ctXorVectorScalar, ctGenVectorScalar

Unary

ctAbs, ctNot, ctLog, ctExp, ctSqrt, ctRsqrt, ctSin, ctCos, ctTan, ctAsin, ctAcos, ctAtan, ctSinh, ctCosh, ctTanh, ctFloor, ctCeiling, ctRound, ctGenUnary

Collective Communication

Reduction

ctAddReduce, ctMulReduce, ctMinReduce, ctMaxReduce, ctAndReduce, ctIorReduce, ctXorReduce, ctGenReduce

Prefix-Sum

ctAddPrefix, ctMulPrefix, ctMinPrefix, ctMaxPrefix, ctAndPrefix, ctIorPrefix, ctXorPrefix, ctGenPrefix

Multi-Reduce

ctAddMultiReduce, ctMulMultiReduce, ctMinMultiReduce, ctMaxMultiReduce, ctAndMultiReduce, ctIorMultiReduce, ctXorMultiReduce, ctGenMultiReduce

Multi-Prefix

ctAddMultiPrefix, ctMulMultiPrefix, ctMinMultiPrefix, ctMaxMultiPrefix, ctAndMultiPrefix, ctIorMultiPrefix, ctXorMultiPrefix, ctGenMultiPrefix

Permutation

Pack/Unpack

ctPack, ctUnpack

Scatter/Gather

ctScatter, ctGather

Shift/Rotate

ctLeftShiftPermute, ctRightShiftPermute, ctLeftRotatePermute, ctRightRotatePermute, ctShiftDefaultPermute, ctRotateDefaultPermute

Partition

ctPartition, ctUnpartition