CuNesl: Compiling Nested Data-Parallel Languages for SIMT Architectures

Yongpeng Zhang, Frank Mueller
North Carolina State University, Raleigh, NC, USA, mueller@csc.ncsu.edu

***Abstract*—Data-parallel languages feature fine-grained parallel primitives that can be supported by compilers targeting modern many-core architectures where data parallelism must be exploited to fully utilize the hardware. Previous research has focused on converting data-parallel languages for SIMD (single instruction multiple data) architectures. However, directly applying them to today's SIMT (single instruction multiple thread) architectures does not guarantee competitive performance.**

We propose cuNesl, a compiler framework to translate and optimize NESL into parallel CUDA programs for SIMT architectures. By converting recursive calls into while loops, we ensure that the hierarchical execution model in GPUs can be exploited on the "flattened" code. The performance gap between our auto-generated CUDA code and hand-crafted CUDA code thus narrows while programmability is greatly increased. Our compiler outperforms handwritten parallel code running on CPUs in terms of both execution time and programmability.

I. Introduction

Exploiting data parallelism is crucial for programming on many-core architectures because data parallelism exposes a much higher degree of parallelism than task or pipeline parallelism. This high degree of parallelism is necessary to keep up with the ever-increasing instruction throughput provided by hardware. However, popular languages, such as C/C++, do not treat data parallelism as a first-class citizen. This gap between the front-end language and hardware is exacerbated by the fact that compilers are struggling to extract data parallelism from language abstractions. Therefore, human assistance is often necessary to increase performance, which adversely affects the programmer's productivity.

Although adding new ways to pass critical data parallelism information to the compiler (*e.g.,* via pragmas) for task-oriented languages maybe a viable method, we take a completely different approach. We investigate what compiling techniques are needed to efficiently map data-parallel languages to state-of-the-art GPU architectures. One of the advantages of this approach is to improve programming productivity because data-parallel languages are often found to be more concise and elegant to express parallel algorithms.

Among the various data parallel languages, NESL [1] is of particular interest. It is based on the concept of nested data parallel abstractions, which are very common in divide-and-conquer parallel algorithms. An apply-to-each construct encourages programmers to think about algorithms in a parallel fashion at the finest data granularity, thus removing the burden for compilers to engage in complicated data dependence analysis. Recursive calls are widely used in NESL, an elegant way to express nested parallelism.

This work was supported in part by NSF grants 1058779, 0958311, 0937908, and DOE grant DE-FG02-08ER25837.

Previous research has studied how to compile data-parallel languages for SIMD vector machines as well as MIMD parallel machines ([2], [3]). Our work, in contrast, targets the SIMT paradigm, an increasingly popular programming model advocated by modern GPU architectures. There is a fundamental difference between SIMD and SIMT. In SIMD, only one flow of control exists. The width of vector data that an instruction operates on can swiftly vary from one instruction to another. SIMT, in contrast, has more control flow resources to support many independent threads, each of which can execute instructions asynchronously. The loosely-coupled threading model in SIMT gives programmers more flexibility and removes the lock-step synchronization of SIMD. Generally speaking, it is more challenging to fully utilize the hardware resources in SIMT. SIMT requires coarser grained data parallelism with many, preferably independent entities to achieve good performance.

An important vectorizing compiler technique is the transformation of nested data parallel languages to SIMD code. The transformed code can be ideally mapped into the *Parallel Vector Model*, which contains a vector processor and a "flattened" vector memory [2]. Unfortunately, today's SIMT is not truly "flattened". In particular, CUDA-enabled GPUs consist of hierarchical levels of threading models with different synchronization properties ([4]): (level-1) a host level CPU thread; (level-2) massive numbers of asynchronous threads in CUDA kernels; (level-3) moderate numbers of synchronizable threads in a CUDA kernel block and (level-4) a relatively small number of lock-step synchronized threads in a warp. A naïve execution of the transformed code uses the level-1 CPU thread as flow control and treats the set of level-2 threads as a unified vector processor. Under the CUDA model, explicit barriers are required for each nesting level but are not supported in hardware at level 2. This lack of support results in many unnecessary and expensive global barriers (at level 1) between explicit kernel calls issued by the CPU, which adversely affects performance.

Therefore, it is desirable to delve into the threading model hierarchy and take advantage of low-overhead local synchronizations. To that end, we spawn the control flow at level-3/4. But recursive functions in NESL pose a performance hurdle. In previous approaches, there was no motivation to remove recursions during code transformations. Yet, invoking recursive functions at level-2/3 will cause overhead such as branch penalties and results in imbalanced computation load. Such overhead cannot be neglected and may consume the

benefits of faster local synchronizations.

As we can see, previous code transformations no longer suffice to generate code suitable for today's hierarchical SIMT architectures. In this work, we design a source-to-source compiler to directly convert NESL to CUDA code that can be efficiently executed on contemporary NVIDIA GPUs. We focus on recursive NESL functions. In addition to the vectorization transformations, we restructure control flow to remove recursion and provide fine-grained data granularity suitable for SIMT architectures. A recursion-free control flow allows us to dynamically switch between hierarchical threading models and then to choose the best one under different scenarios.

The current cuNesl compiler targets CUDA C++, a vendor proprietary programming model from NVIDIA. However, the proposed compiler techniques can be extended to other data-parallel languages, such as data-parallel Haskell [5].

II. NESL Language

In this section, we give a brief introduction to NESL. NESL is an example of a data-parallel language, also known as a collection-oriented language [6]. It is strongly typed and declarative (free of side-effects).

Like other data-parallel languages, NESL consists of standard apply-to-each (map) constructs. The apply-to-each construct applies a certain operation to all elements of a sequence. For example, the expression

```
{negate(a): a in [3, -4, -9, 5]};
```

negates the sequence of numbers in an element-wise fashion in *parallel*, resulting in a sequence of values [-3, 4, 9, 5]. NESL ensures that apply-to-each constructs can be executed independently per element. Therefore, they can easily be mapped onto data-parallel execution models.

A set of primitive parallel functions that can operate on sequences are pre-defined in NESL as well. These functions are not necessarily embarrassingly parallel but still represent efficient parallel algorithms. An example is "b = permute(a,i)", where sequence b is formed in such a way that the *j*th elements in sequence a is permuted to position i[*j*] for all *j*s.

Support for nested parallelism is one of the key ideas behind NESL. Elements in a sequence in NESL can itself be a sequence, which supports recursively nested sequences. Such nested parallelism comes from NESL's ability to apply any function in parallel over the elements of a nested sequence. For example, a sum applied to a nested sequence forms a set of parallel sum calls in a nested fashion.

```
{sum(a) : a in [[2,3], [8,3,9], [7]]};
=> it = [5, 20, 7] : [int]
```

NESL defines several functions to support nesting and unnesting of a sequence, including flattening (reducing the nesting by one level) and bottop (splitting a sequence in two halves and returning them as a nested sequence). NESL is very powerful in expressing divide-and-conquer parallel algorithms with nested recursive calls. Quicksort written in NESL is depicted in Figure 1. The expression

```
result = {qsort(v): v in [less, greater]}
```

applies the recursive calls to qsort on a nested sequence formed by less and greater sequences. Nested parallelism, in this case, means that both the two qsorts and the generation of three intermediate arrays inside qsort can be performed in parallel.

```
1 function qsort(a) =
2 if (#a < 2) then a
3 else
4   let pivot = a[#a/2];
5       less = {e in a| e < pivot};
6       equal = {e in a| e == pivot};
7       greater = {e in a| e > pivot};
8       result = {qsort(v): v in [less,greater]};
9   in result[0] ++ equal ++ result[1] $
```

Fig. 1. Quicksort in NESL

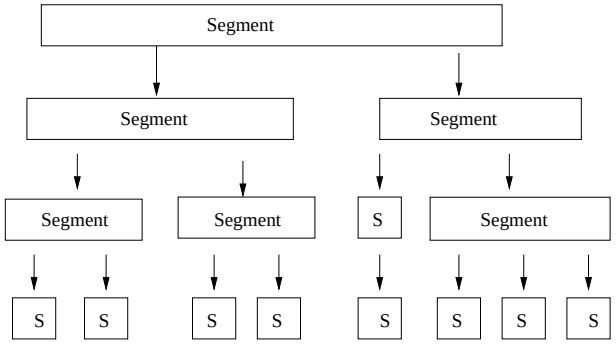


Fig. 2. Segmented Array in Quicksort: Each row is a segmented array.

A. Segmented Array

Previous research translates data-parallel languages (*e.g.*, NESL) into a stack-based intermediate language called VCODE ([7]), which is tailored to SIMD machines. This transformation is called *flattening of nested parallelism* [8]. The basic data type of VCODE is a flattened segmented array. Unlike the nested sequence in NESL, it allows only one level of partitions.

Figure 2 depicts the dynamic partitioning of segmented arrays for quicksort. Each row in the figure is a segmented array. Initially, a single segmented array with just one segment exists. As more and more partitions are formed, the segmented array breaks into many smaller segments.

CuNesl adopts this concept and provides an efficient implementation for pre-defined parallel operations on segmented arrays. We will provide more details in Section V-A.

III. Related Work

Programming on SIMT architectures has quickly become mainstream since the launch of CUDA and has changed the GPU's image from that of a purely graphics-specific accelerator to a general-purpose co-processor. While a tremendous numbers of applications can benefit from manually rewriting legacy code for CUDA, many researchers strive to improve the programmability without sacrificing performance.

One approach is to provide handwritten, highly-efficient implementations for well-defined APIs so that they can directly be used by other programs. CUDPP [9], Jacket [10] and Thrust [11] are examples of this approach. In fact, cuNesl's

implementation directly uses CUDPP's parallel scan/reduce APIs. However, this only applies to certain areas where the interface can be clearly defined or standards exist.

By restricting problems into specific domains, compilers can aggressively exploit domain-specific knowledge to auto-generate efficient CUDA code. Domains like stencil computation [12], [13], [14], streaming [15] and PDE solvers [16] are already benefiting from this approach.

For general-purpose languages, a common method is to add directives (*e.g.*, pragmas) to enable code generation by the CUDA back-end. They can be either extending existing directives like OpenMP [17] or introduce new sets of pragmas [18], [19]. There are also source-to-source compilers that translate a naive CUDA kernel into an optimized highly efficient version [20].

In terms of data-parallel languages, the PGI CUDA Fortran Compiler [21] directly compiles HPF into CUDA source code. The compilation of other data-parallel languages, such as Haskell and Python, into CUDA code is still an active research topic [22], [23], [24].

CuNesl shares the same philosophy as Copperhead [24] in that a hierarchical execution model should be exploited in today's architectures to achieve good performance for nested parallelism. CuNesl also extends the applicability of this concept to recursive calls, which cannot easily be statically mapped to finite execution hierarchies and are thus beyond Copperhead. In addition, we show that a *nested* flattening transformation, if coupled with data-flow analysis on the transformed code, matches the hierarchical execution model for SIMT architectures and results in additional performance benefits.

IV. cuNesl Compiler

A. Removing Recursive Calls

As discussed in Section II-A, removing the recursive calls in NESL is important for efficient compilation in SIMT architectures. In this section, we will use quicksort as an example to show how cuNesl maps a recursive function into a while loop, even for some non-tail recursion cases.

For a recursive function to terminate, there are always conditional branches inside the recursive function. At least one of the branches does not make further recursive calls. A simplified control flow for a recursive function is illustrated in Figure 3(a). The P2() branch is the exit path for the recursive call. In the recursive path, if P4() is empty and this path directly returns after issuing a single recursive call, then it is a tail-recursive function. Most of the NESL examples do not fall into the category of tail-recursion, for they either have multiple recursive calls or have a non-empty P4() block. Fortunately, NESL's syntax guarantees that such multiple recursive calls, if they exist, can be executed in parallel. And it is often the case that P4() is a simple operation that, if positioned prior to the recursive calls, does not affect the final output. Examples of such operations are concatenation, flatten and bottop. Figure 3(b) shows the recursion-free transformation based on the above assumptions. The recursive function is now replaced by a while loop, which exits once all segments

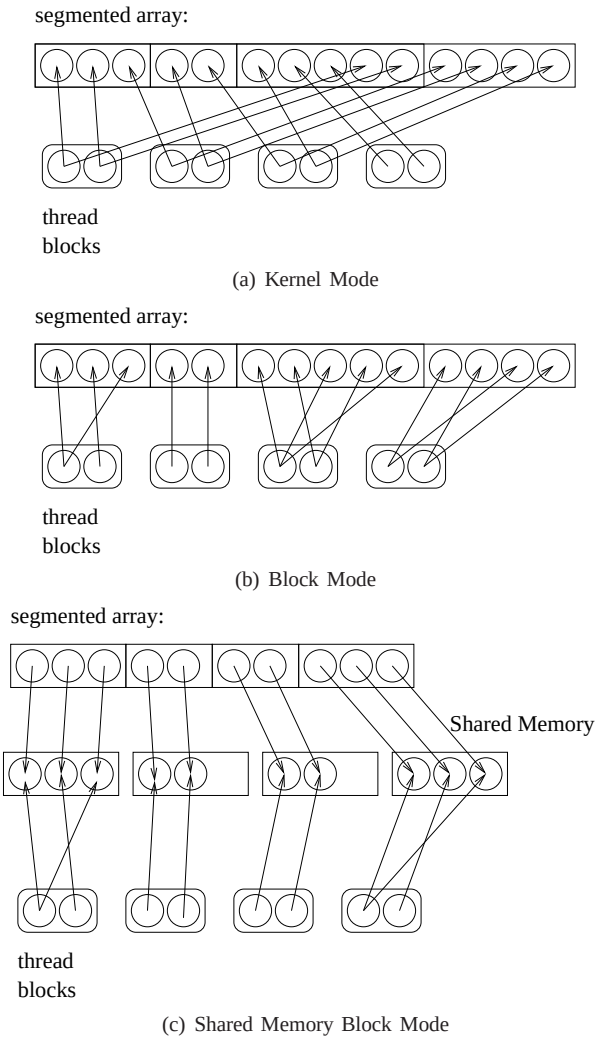
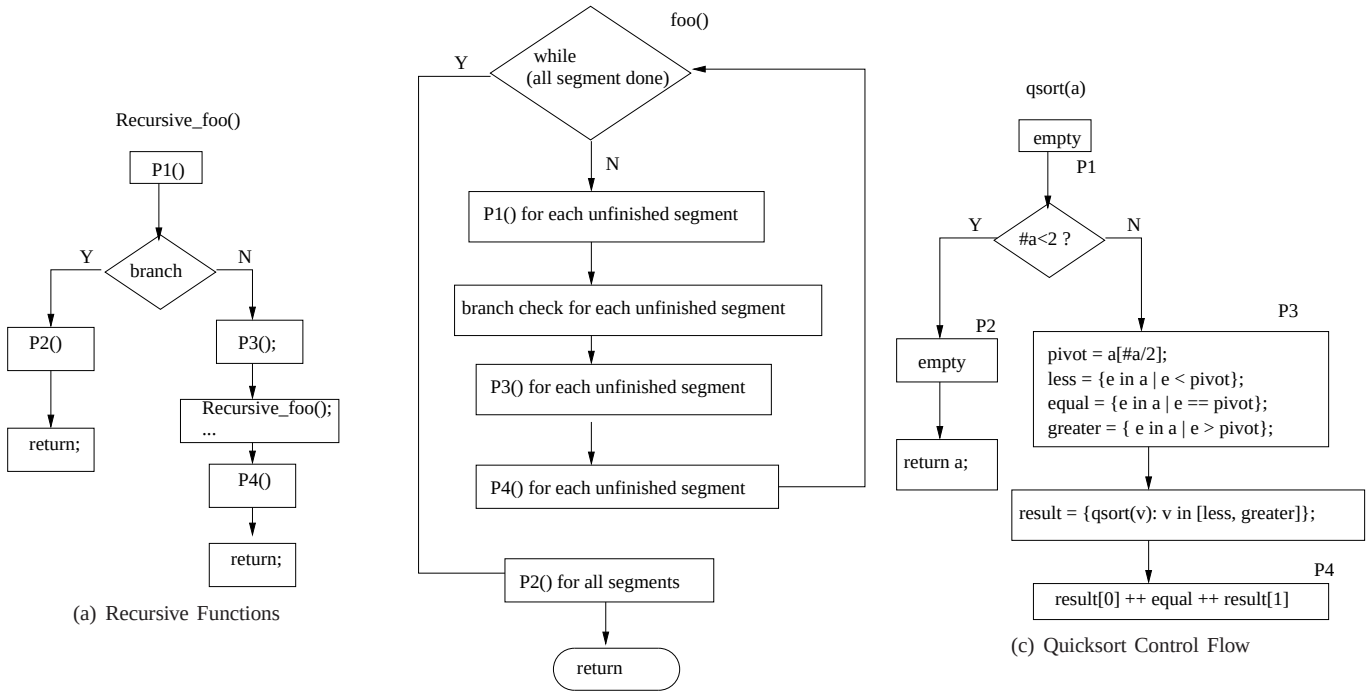


Fig. 4. Different Execution Modes. In kernel mode, threads process elements in the array globally. In block mode, one block is assigned to a segment. Shared Memory block mode is an optimized version of block mode. It utilizes the on-chip Shared Memory to reduce global memory accesses.

terminate (have reached the exit branch). Inside the loop, all operations are applied to segments that have not been marked as finished. P2() is executed when the loop exits. Here, we also assume that P2()'s execution can be safely moved to the end. Otherwise, this step need would need to be moved inside the while loop so that it is applied to every segment that has *just* terminated.

Quicksort in NESL is an example of an algorithm that can be transformed into a parallel tail-recursive function. The mapping from NESL source code into the recursive control flow is shown in 3(c). P4() is a simple concatenation operation. Therefore, the compiler can perform code motion to place it before issuing the recursive calls. This is done by inserting the "equal" sequence in in between the "less" and "greater" sequences and marking this as a non-recursive segment.

Figure 5 lists the resulting code generated for quicksort. The compiler fuses operations that share the same input into one operation. For instance, all three intermediate flag arrays for "less", "equal" and "greater" are generated from the same



(a) Recursive Functions

(b) Segmented Version

(c) Quicksort Control Flow

Fig. 3. Convert (a) Recursive_foo() into (b) a recursion-free while loop with (c) an example for Quicksort.

```
1  void qsort(SegmentArray<T> &array) {
2    while(!array.isRecursiveAllDone()) {
3      /* branch 0, check segment length */
4      array.setRecDoneByLength(1);
5      /* branch 1, */
6      MirroredArray<T> pivots(array.getNumSegments())
           ;
7      gen_pivots(array, pivots);
8
9      MirroredArray<uint> less_flag(array.getSize());
10     ...;
11     gen_flags_from_pivot(array, pivots, less_flag, ...);
12
13     FlagSubIrregularSegmentArray<T> less();
14     FlagSubIrregularSegmentArray<T> equal();
15     FlagSubIrregularSegmentArray<T> greater();
16
17     FlagSubIrregularSegmentArray<T> *children[3];
18     children[0] = &lesser;
19     children[1] = &equal;
20     children[2] = &greater;
21
22     /* reshuffle each segment in array into 3 segments,
23        a built-in function in segmented array */
24     array.reshuffle(&children[0], 3);
25   }
26 }
```

Fig. 5. Generated Code for Quicksort

kernel function. The concatenation of three segments is a built-in function of our segmented array (reshuffle() method).

B. Hybrid Execution Mode

After converting recursive routines into iterative while loops, we have successfully flattened the program and made it suitable for SIMT architectures. Threads can now start from the bottom and work at the finest data granularity during the entire execution. But in practice, this transformation alone does not usually deliver competitive performance. The reason is that today's SIMT architectures consist of a hierarchy of execution modes, each of which has its own characteristics. Consider CUDA, which has the following execution levels:

Kernel level: This is similar to the bulk synchronization model [25]. Control flow is driven by one or a few host threads on the CPU side. Concurrent computation is performed by launching massively-threaded CUDA kernels. Global synchronization is feasible (between CUDA kernel launches) but relatively expensive.

Block level: This level operates inside CUDA kernels on a GPU. Threads in the same block execute the same program, but do not necessarily proceed at the same rate. Sharing between threads can be realized via Shared Memory. Synchronization at block level is relatively cheap. In the CUDA context, it is supported trough the $\_\_syncthreads()$ API call.

Warp level: This level is similar to SIMD in the sense that threads in the same warp execute programs at the same pace on a GPU. There can be one or more warps at the upper block level. Branches are more efficiently executed if threads in the warp all agree to take the same path. Synchronization between warps is zero-overhead because it is enforced by the hardware via lock-step execution.

Experienced CUDA programmers often choose particular execution levels to solve different problems, or even a problem at different stages, based on various factors. If global synchronization is only occasionally required, a kernel level program should be designed. If a problem can be divided into at least a moderate number of independent smaller problems (a divide-and-conquer approach), it is generally more efficient to work

```
1  template <class T> __global__ void
2  quicksort_block(IrregularSegmentGpuArrayC<T> *array) {
3     __shared__ FlagSubIrregularSegmentGpuArray<T> less;
4     __shared__ IrregularSegmentGpuArrayC<T> s_array;
5     __shared__ GpuArray<uint> less_flag; ...
6     __shared__ GpuArray<T> pivots;
7     // temporary buffer for parallel scan/reduce
8     __shared__ uint mSharedBuffer[...];
9     __shared__ FlagSubIrregularSegmentGpuArray<T> *
            children[3];
10
11    __syncthreads();
12    // copy the segment info locally
13    if (threadIdx.x == 0)
14          array→clone(&s_array);
15
16    __syncthreads();
17    ...
18    int segid = blockIdx.x;
19    if (array→isRecursiveDone(segid))
20       return;
21    // prepare for the assigned segment
22    s_array.convertToLocal(bid);
23    __syncthreads();
24
25    // the while loop from the recursive call
26    while (!s_array.isRecursiveAllDone()) {
27          s_array.setRecDoneByLength(1);
28          __syncthreads();
29          if (s_array.isRecursiveAllDone())
30             break;
31          __syncthreads();
32          ...;
33          gen_pivots_block(s_array, pivots);
34          __syncthreads();
35          gen_flags_block(s_array, pivots, less_flag,..);
36          __syncthreads();
37          ...;
38          s_array.reshuffle(children, 3, mSharedBuffer);
39          __syncthreads();
40       }
41    // copy the data back to the global array
42    s_array.copyFromLocal();
43 }
```

Fig. 6. Generated Code for Quicksort in Block Mode

at the block level because kernel launch overhead and global synchronization are reduced. It is not uncommon to utilize the lock-step synchronization property at the warp level for small but communication-rich operations. Such examples can be found in efficient CUDA implementations of parallel reduce or scan [26].

Lessons learned from coding styles of real-world applications lead us to believe that a hybrid execution mode is necessary to achieve good performance in cuNesl. A truly flattened hardware is not likely to be available due to the unavoidable tradeoff between hardware resources and performance. Relying on a flattened execution mode will only underutilize the hardware, which would make such a method inferior to other approaches.

Therefore, we define several execution modes in cuNesl corresponding to the hierarchical levels of hardware abstractions. This is best explained in the context of how to access and manipulate elements in a segmented array for a massive number of independent SIMT threads. Right now, cuNesl defines the following three execution modes:

Kernel Mode:This mode corresponds to the kernel level abstraction above. When the segmented array consists of only a few large segments, it does not make sense to assign a large segment to only one thread block. Instead, it is more efficient to spawn as many threads as possible and allow multiple blocks to work on the same segment (Figure 4(a)). The drawback of this mode is that synchronizations across a segment can only be performed between disjoint CUDA kernels, which is relatively expensive. From the recursive routine's point of view, this mode is usually advocated at the beginning of a recursive call where the number of partitions is small. The foreach operations on a segmented array are translated into kernel pseudo code like the following:

```
stepsize = blockDim.x * gridDim.x;
for (id = threadIdx.x; id < size; id += stepsize)
{
   segid = getSegId(id);
   seglen = getSegLen(segid);
   ...
}
```

Block Mode: When the segment array contains a moderate number of segments, we can assign each segment to an exclusive thread block (Figure 4(b)). This corresponds to the block level abstraction. Because a barrier is supported within a thread block, many operations on segments, though not embarrassingly parallel, can be performed without leaving the kernel, thus reducing kernel launch overheads. This mode can often be applied during the mid-phase of a recursive call when enough partitions are produced to fully utilize the many-cores of SIMT architectures. The foreach operations on a segmented array, in this mode, is translated to the following pseudo-code inside the CUDA kernel:

```
stepsize = blockDim.x;
segid = blockIdx.x;
segsize = getSegLen(segid);
segoffset = getSegOffset(segid);
for (id = threadIdx.x; id < segsize; id += stepsize)
{
  my_global_id = segoffset + id;
  ...
}
```

Shared Memory Block Mode: One important and effective optimization opportunity arises when the size of each segment becomes small enough to fit in the on-chip Shared Memory. We can preload segments into Shared Memory first and work on them before storing them back to global memory (Figure 4(c)). This way, we can reduce memory bandwidth consumption. Because Shared Memory is limited in size, this mode is usually feasible and more efficient near the end of a recursive call. This mode can be regarded as an optimized version of *block mode*.

As of now, we have not explored the benefits of going down to the warp level in cuNesl because warp level programming is often found in low-level libraries that are *used* by cuNesl. This is not to say that the lockstep synchronization at warp level is unimportant. A study to assess if this mode is beneficial for

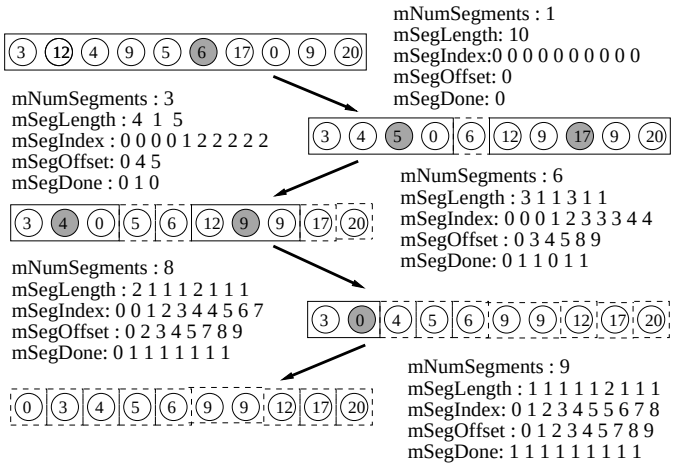


Fig. 7. The modification to the segmented array for the quicksort. Shadowed elements are quicksort pivots. Elements in the same segment are grouped by rectangular boxes. Dotted boxes indicate segments that are not subject to recursive calls.

more general cases is subject to future research.

Going back to quicksort, the pseudocode in Figure 5 is in fact generated for the *kernel mode*, only. To switch to other execution modes, cuNesl adds a counter check inside the while loop to exit the loop early, i.e., once the number of segments exceeds a threshold. It then calls a single CUDA kernel that executes the rest of the iterations in *block mode*. The pseudo-code for this single kernel is shown in Figure 6. It contains a similar while loop as in the *kernel mode* (Figure 5). Functions that are used as kernel calls in *kernel mode* are transformed into device functions in a segmented version. Barrier synchronization is provided by __syncthreads() between parallel regions.

V. Runtime

A. Segmented Array

The core of cuNesl's runtime system supports the necessary primitives for segmented arrays. Segmented arrays are encapsulated in various classes that can be included in the compiler-emitted code. They are further compiled by NVCC to generate binaries. To support the concept of a segmented array, to make it conveniently available to the programmer and to ensure efficiency for fine-grained SIMT threads to work on individual elements, we add several auxiliary arrays besides the raw data array to maintain the state of a segmented array (assuming its size is N):

mSegments: array of size N. Elements in this array are either 1 or 0. A 1 indicates the start of a new segmented array.

mNumSegments: size one. It stores the number of segments in a segmented array.

mSegIndex: array of size N. mSegIndex[i] returns which segment the i-th element belongs to.

mSegOffset: array of size mNumSegments. It stores the offset of each segment relative to the starting address of the data array.

mSegLength: array of size mNumSegments. mSegLength[i] returns the length of the i-th segment.

mSegDone: array of size mNumSegments. It is used for recursive calls. A "1" at position i means that the i-th segment has reached the exit condition of the recursive call.

In the quicksort example, such segment information also preserves the current state of the quicksort recursion. Figure 7 illustrates the status of a segmented array for quicksort. The six auxiliary arrays (32 bits each) come with a linear increase in the memory footprint for a total of 24 bytes per NESL data structure.

The layout of a segmented array can be dynamically modified by the user via storing a 1 in the mSegment array. The runtime is responsible for adjusting the remaining auxiliary arrays accordingly. We have developed an efficient data-parallel approach via Algorithm 1 to minimize the execution time of this operation. These auxiliary data structures together with the algorithm help reduce the overhead of the code generated by cuNesl.

Algorithm 1 Update auxiliary arrays from mSegments

```
mSegIndex := Inclusive_Scan(mSegments, N);
Barrier();
mNumSegments := mSegIndex[N-1];
Barrier();
for i = 1 → N do
  mSegIndex[i] := mSegIndex[i] - 1;
end for
Barrier();
for i = 1 → N do
  if mSegments[i] == 1 then
    mSegOffset[mSegIndex[i]] := i;
  end if
end for
Barrier();
for i = 1 → mNumSegments do
  if i == (mNumSegments - 1) then
    nextOffset := N;
  else
    nextOffset := mSegOffset[i+1];
  end if
  mSegLength[i] = nextOffset - mSegOffset[i];
end for
```

In this parallel algorithm, all for loops and the Inclusive_Scan() function can be efficiently and cooperatively (independently) executed by SIMT threads. We need to generate two versions of code based on this algorithm to fulfill the need for different execution modes discussed in Section IV-B, one for the *kernel mode*, the other for the other two modes. In the *kernel mode*, all for loops are transformed into separate CUDA kernels and Inclusive_Scan() is invoked by calling appropriate CUDPP library APIs [9]. A Barrier() is implicitly enforced by the CUDA runtime. In the *block mode* and the *shared memory block mode*, the entire algorithm becomes a device function called by other device or global functions.This also applies to the Inclusive_Scan() function, which only needs to perform a local scan at the block level. Barrier() needs to be

instantiated by __syncthreads() (provided as a CUDA device function) to ensure correctness.

This strategy to provide kernel-level and block-level support for an operation needs to be applied to either pre-defined NESL primitives in the runtime or emitted code by the cuNesl compiler. This allows us to exhaustively explore different combinations of execution modes to find the fastest combination. Fortunately, except for a few differences (*e.g.*, barriers in *block mode* are realized via __syncthreads()), these two versions are similar to each other.

The implementation of the CuNesl runtime takes advantages of existing hand-crafted CUDA libraries for many of the parallel primitives supported in NESL. For example, CUDPP's APIs at different layers are heavily used in our runtime system. We also provide implementations of other primitives, such as sum, concatenation and reverse.

B. Optimizations

We call a segmented array a *regular* segmented array when all its segments are of the same length. For such segmented arrays, we do not need to waste memory and time to maintain the aforementioned auxiliary arrays. Instead, only a single scalar is needed to keep track of the length of each segment in the array. All other information, such as segment offset and the corresponding segment id for an element, can be calculated on-the-fly and independently by SIMT threads. The runtime system will convert a *regular* segmented array to a non-regular one whenever necessary.

VI. Experimental Results

We conducted our experiments on a Quad-core Intel(R) Xeon(R) CPU E5507 machine with 6 GB memory. The GPU was a Geforce GTX 480 consisting of 15 Streaming Multiprocessors. The host code was compiled by Gcc 4.4.4. CUDA code was compiled by NVCC, CUDA release 4.0. Both Gcc and CUDA codes are compiled at optimization level -O3.

A. Quicksort

We present cuNesl's quicksort performance by comparing with three other implementations:

GPU-Quicksort: This is a hand-written CUDA sorting library using quicksort in the beginning and switching to bitonic sort in the end [27]. To the best of our knowledge, it is the fastest open-source GPU implementation involving quicksort. The total number of source code lines, including both the host-side C++ and CUDA, adds up to about 900 lines.

STL: This is also a hybrid sorting implementation: it first uses introsort, which is based on quicksort, followed by insertion sort. It is run on CPUs only.

OpenMP: We also wrote quicksort in OpenMP using the *parallel* pragma directives. This, too, is run on CPUs only. The maximal number of threads is 8. The same thread configuration applies to other experiments.

We use the number of lines of code (LOC) as a metric to assess the programmability, i.e., reflecting the effort of the programmer to write code.

TABLE I
Quicksort: Line of Code Comparison

Implementation	LOC
GPU-Quicksort	900
cuNesl	9
STL	100
OpenMP	130

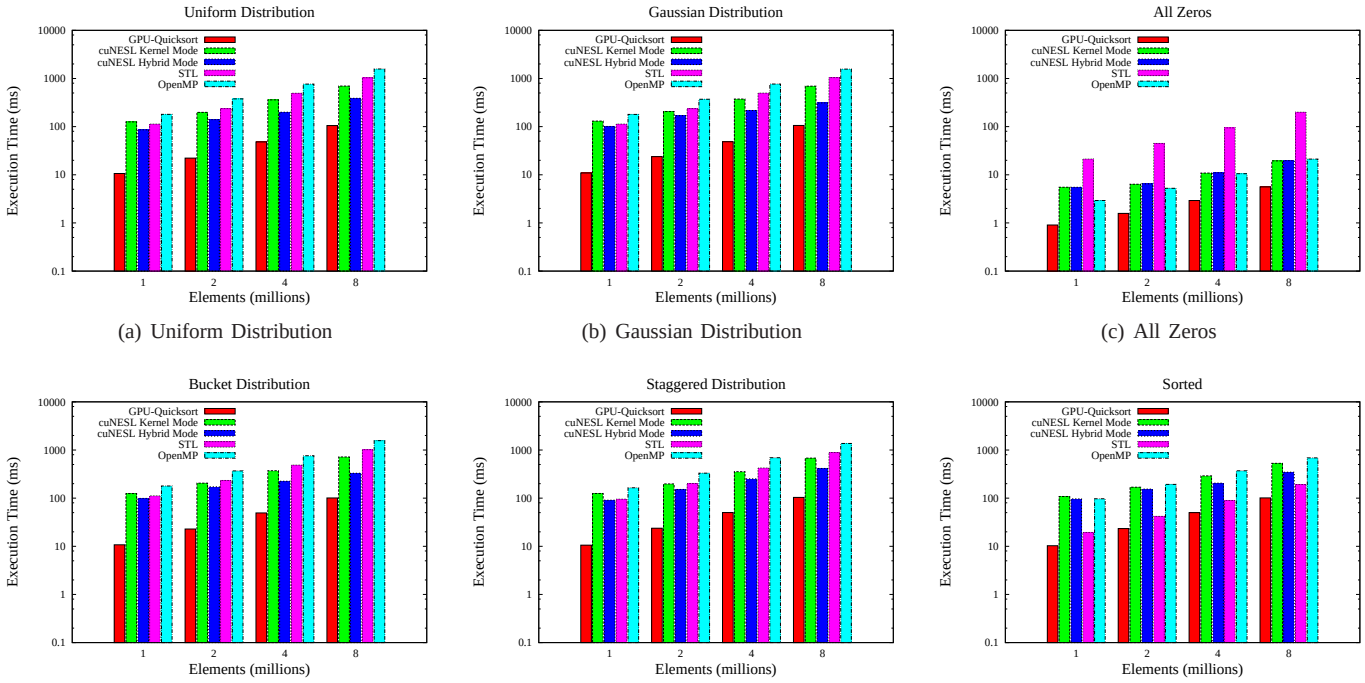
For STL constructs, we are counting the lines of code at the first major level, e.g., inside of std::sort(). In our OpenMP implementation, we use std::partition() to split arrays into halves, which is a central part of quicksort. This hand-written STL code is counted as just one LOC in the table. The LOC metric shows that NESL supports extremely concise expressions of such a recursive function: The LOC metric is one to two orders of magnitude less than for the other implementations.

We adopted the same testing strategy as in [27] by measuring the execution time under different input distributions, namely uniform, Gaussian, zero, bucket, staggered and sorted. The details of these distributions are explained in [27]. We slightly revised the original Quicksort NESL script to choose a better pivot element for each segment array. Instead of blindly taking the element at the middle index, we pick the pivot as the average of the max and min value in each segment.

The final performance is shown in Figure 8. The Y axis shows the execution time on a log scale. The X depicts shows the array size from one million to eight million elements (numbers). For cuNesl, we show two bars. The first is obtained by only generating code in the *kernel mode*. The second starts with *kernel mode* and then switches to *block mode* after producing enough segments (256). This is referred to as the "hybrid mode" in the figures. The switching point needs to be tuned (currently manually, could be automated) because it depends on the size of the sorting data types and the resource usage (register and Shared Memory). We can see that the hybrid mode usually takes about half of the time of the *kernel mode*. This demonstrates our previous hypothesis that different execution modes are suitable for different segment arrays. We also tried to add the *shared memory block mode* to the hybrid mode when segments are becoming small enough fit in the GPU's Shared Memory. But it provides no improvement over the two-stage hybrid mode. The extra barrier and bookkeeping between the *block mode* and *shared memory block mode* resulted in a net performance loss due to overheads. Therefore, the execution time in this case is not displayed in the Figure 8.

Figure 8 shows that our best compiled quicksort routine (hybrid mode) is about two to three times slower than the hand-written CUDA implementation (GPU-Quicksort). This is mainly due to three reasons:

- GPU-Quicksort uses bitonic sort at the end, i.e., after spawning a sequence of the quicksort recursions. Quicksort is well known to be less efficient than bitonic sort due to the partition imbalance problem.
- GPU-Quicksort is using problem-specific knowledge to reduce execution time. For this particular case, the pro-



(a) Uniform Distribution (b) Gaussian Distribution (c) All Zeros

(d) Bucket Distribution (e) Staggered Distribution (f) Sorted

Fig. 8. Quicksort Results

```
1  function bitonic_sort(a) =
2  if (#a == 1) then a
3  else
4      let
5      bot = subseq(a,0,#a/2);
6      top = subseq(a,#a/2,#a);
7      mins = {min(bot,top):bot;top};
8      maxs = {max(bot,top):bot;top};
9      in flatten({bitonic_sort(x) : x in [mins,maxs]}) $
10
11 function batcher_sort(a) =
12 if (#a == 1) then a
13 else
14     let b = {batcher_sort(x) : x in bottop(a)};
15     in bitonic_sort(b[0]++reverse(b[1]))
16     $
```

(a) Batcher Sort in NESL

```
1  void batchersort(SegmentArray<T> &array) {
2    while (!array.isRecursiveAllDone()) { // first while loop
3      array.setRecDoneByLength(1);
4      // nothing to do, just push the segment info
5      array.pushSegments(0); // line 14 in NESL
6      array.bottop();
7    }
8    array.popSegments(0); // no action for the finest granularity
9    while (array.popSegments(0)) {
10     SubSegmentArray<T> bs(&array, Sub_Bot);
11     bs.reverse(); // correspond to the reverse call in line 15
12     while (!array.isRecursiveAllDone()) {// seond while
13       array.setRecDoneByLength(1);
14       if (array.isRecursiveAllDone()) break;
15       genMinMax(array); // responsible for line 5 to 9
16       array.bottop(); // deduced from subseqs in line 5 and 6
17     }
18   }
19 }
```

(b) Generated CUDA C++ code for Kernel Mode

Fig. 9. Batcher Sort

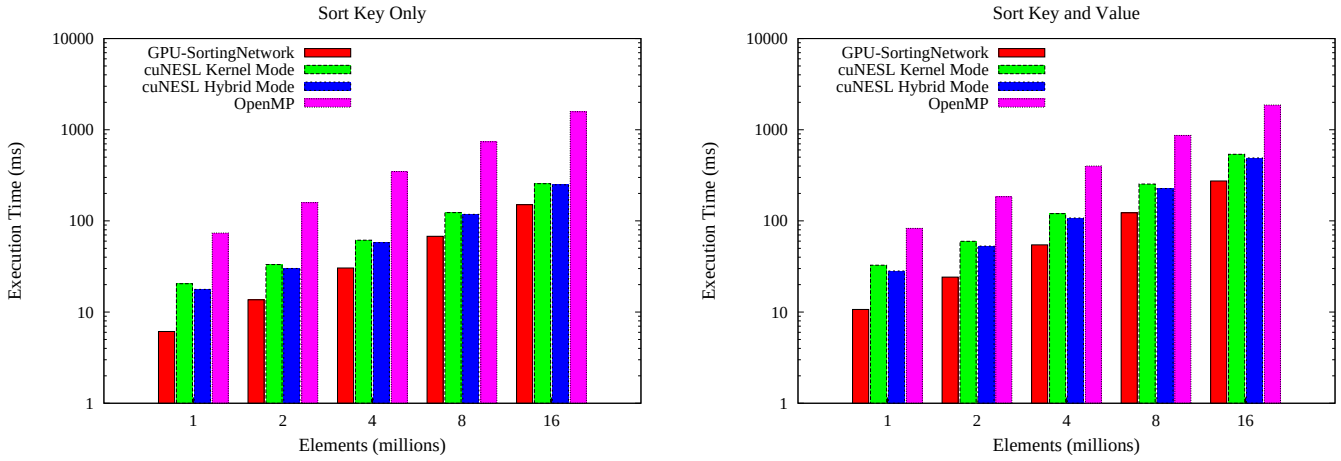
grammer knows that the concatenated total length from the less, equal and greater arras (partitions) are the same as the original array. This greatly increases the parallelization opportunity because the new offset for each element can be calculated independently inside a quicksort partition. Such information is difficult to deduce for the cuNesl compiler. Therefore, for safety reasons, a global scan needs to be performed to calculate the new offset in *"kernel mode"*. This enforces a barrier between different depths of recursion.

- For handwritten quicksort, programmers do not need to maintain auxiliary arrays for segmented arrays. They just need to keep record the sizes of each sub-array. (All other variants require these sub-arrays to support segmented arrays and incur overhead for maintaining these auxiliary data structures.)

The performance in all our cases is two to three times higher than STL, which is usually one third faster than our handwritten OpenMP implementation, except for the all-zero case. Considering the tremendous advantage in terms of programming effort, we believe that cuNesl is a viable way to realize data-parallelism for SIMT architecture.

B. Batcher Sort (Bitonic Sort)

We also evaluated the Batcher Sort benchmark, which recursively calls Bitonic sort in a depth-first manner. Bitonic sort itself is also a recursive call that keeps sorting symmetrical



(a) Bitonic Sort (Key Only)

(b) Bitonic Sort (Key and Value)

Fig. 10. Batcher Sort Results

partitions in the first and second halves at different granularity levels. The NESL source code depicted in Figure 9(a) is almost as concise as that of quicksort.

This benchmark represents a typical example of multiple recursions. Correspondingly, cuNesl generates one while loop for each recursion. The top-level C++ code for the *kernel mode* is shown in Figure 9(b). Because the first-level recursion is operating in a bottom-up manner, we need to push the segment information onto a stack and invoke the second-level recursive functions (bitonic sort) when segments are popped (see the while loop at line 9). The second-level recursion is transformed into the while loop the same way as for the quicksort routine (see lines 12-17).

We compare cuNesl with two other implementations of Batcher Sort:

GPU-SortingNetworks: This code is released as an example in NVIDIA's CUDA SDK. It features highly optimized hand-crafted CUDA code.

OpenMP: We also rewrote Batcher Sort in C++ utilizing the OpenMP *parallel for* pragma directive for parallelization. This version runs on CPUs only.

The LOC summary is listed below. Again, cuNesl (NESL) increases the programmer's productivity as an order of magnitude fewer LOCs are required.

TABLE II
BATCHER SORT: LINE OF CODE COMPARISON

Batcher Sort	LOC
SortingNetworks	250
cuNesl	15
OpenMP	120

We applied batcher sort on two kinds of arrays: one is just a key array with unsigned int type; the other is a (key, value) pair array with unsigned int type for both key and value. Observed execution times are shown in Figure 10. Similar to quicksort, we provide two bars for cuNesl. One is obtained by executing in *kernel mode* only. The "hybrid mode" in this case means *kernel mode* followed by *shared memory block mode*. As shown in the figure, the "hybrid mode" is about 10% faster than the *kernel mode*. Given the fact that the *shared memory block mode* saves global memory traffic, it indicates that batcher sort is memory bandwidth bound. The same conclusion can be drawn for the other two implementations as well because they both take roughly twice as much time to sort the (key, value) pair as just the key array.

A closer look at the source code of GPU-SortingNetworks reveals that this program also divides the execution into two phases, where in the later stage it puts small sub-arrays into Shared Memory to reduce bandwidth consumption. This is exactly what cuNesl does. The handwritten CUDA code does not need to keep track of changes in the segmented array, making it about $30 - 40\%$ faster than the best cuNesl code (hybrid mode).

Batcher sort is more friendly to parallelization than quicksort, even though it only works for arrays of certain sizes (power of two). Within the investigated input size range (one million to eight million elements), batcher sort is twice as fast as quicksort on C++ code. CuNesl achieves up to a 5X speedup over the parallel OpenMP implementation.

C. Discussions

NESL's conciseness comes along with sacrifices: it can only pass limited information to the compiler. A human programmer can exploit algorithm-specific knowledge that a compiler cannot easily deduce. Therefore, we do not expect cuNesl's performance to be at par with hand-optimized GPU code. After all, it is often argued that the performance of a language is proportional to the required programming effort, especially for GPUs. Our results in the above two sorting algorithms show that the performance gap is not as large as the programming effort saved. The results are even more compelling when comparing cuNesl with codes running on CPUs. Our compiler outperforms them in terms of both execution time and programmability. In addition, there is still much room for cuNesl to improve its performance. Adding directives (e.g., OpenMP pragmas) maybe a promising direction for future research.

VII. Future Work

CuNesl is under active development. There are many exciting directions we would like to pursue to make it more robust and efficient. Some are mentioned in previous sections. Additional ideas are listed below:

Auto-Tuning: At the current stage, the transition threshold between different execution modes is emitted as heuristic constants. Our reported result is obtained by manually tuning those constants. Our experience shows that changing those constants can sometimes make a significant difference in performance. It is thus desirable to auto-tune these parameters.

Non-Recursive Functions: This paper mainly focuses on how to transform recursive functions in NESL and optimize them. For non-recursive functions, we would like to show that cuNesl performs equally well by transforming independent code schemes into segments.

Scheduling of Execution Mode: Right now, the switching between different execution modes is hand-coded: a barrier exists that prevents two execution modes from overlapping in time. By aggressively scheduling modes in parallel, we may be able to obtain better performance for irregular algorithms, such as quicksort.

VIII. Conclusions

This paper presents translation techniques for a nested data parallel language to be efficiently executed on modern SIMT architectures. Previous approaches to convert nested parallelism into flattened segments failed to consider the hierarchy of execution modes of modern architectures. We show that by applying control-flow transformations on the flattened code, the new recursion-free control flow provides the freedom to dynamically transition between different threading models. The resulting CUDA code allows the user to enjoy both the conciseness of data-parallel languages and the computational power of SIMT accelerators.

References

[1] G. E. Blelloch and P. R. Model, "NESL: A Nested Data-Parallel Language," Tech. Rep., 1993.

[2] G. E. Blelloch, *Vector Models for Data-Parallel Computing*. Cambridge, MA, USA: MIT Press, 1990.

[3] S. Chatterjee, "Compiling Nested Data-Parallel Programs for Shared-Memory Multiprocessors," *ACM Trans. Program. Lang. Syst.*, vol. 15, pp. 400–462, July 1993. [Online]. Available: http://doi.acm.org/10.1145/169683.174152

[4] "NVIDIA Cooperation, CUDA Programming Guide."

[5] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow, "Data Parallel Haskell: a Status Report," in *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, ser. DAMP '07. New York, NY, USA: ACM, 2007, pp. 10–18. [Online]. Available: http://doi.acm.org/10.1145/1248648.1248652

[6] J. Sipelstein and G. E. Blelloch, "Collection-Oriented Languages," *Proceedings of the IEEE*, vol. 79, no. 4, pp. 504–523, 1991.

[7] G. E. Blelloch and S. Chatterjee, "VCODE: A Data-Parallel Intermediate Language," in *Proceedings Frontiers of Massively Parallel Computation*, 1990, pp. 471–480.

[8] G. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha, "Implementation of a Portable Nested Data-Parallel Language," *Journal of Parallel and Distributed Computing*, vol. 21, pp. 102–111, 1994.

[9] http://code.google.com/p/cudpp/, "CUDPP."

[10] http://www.accelereyes.com, "Jacket."

[11] J. Hoberock and N. Bell, "Thrust: A Parallel Template Library," 2010, version 1.3.0. [Online]. Available: http://www.meganewtons.com/

[12] D. Unat, X. Cai, and S. Baden, "Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C," in *Proceedings of the 25th International Conference on Supercomputing (ICS'11)*, 2011.

[13] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers," 2011.

[14] M. Christen, O. Schenk, and H. Burkhart, "PATUS: A Code Generation and Autotuning Framework For Parallel Iterative Stencil Computations on Modern Microarchitectures," *In IEEE Intl Parallel and Distributed Processing Symposium (IPDPS)*, May 2011.

[15] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, "Software pipelined execution of stream programs on gpus," in *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 200–209.

[16] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, "Liszt: A Domain Specific Language for Building Portable Mesh-Based PDE Solvers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 9:1–9:12. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063396

[17] S. Lee, S.-J. Min, and R. Eigenmann, "Openmp to gpgpu: a compiler framework for automatic translation and optimization," *SIGPLAN Not.*, vol. 44, pp. 101–110, February 2009. [Online]. Available: http://doi.acm.org/10.1145/1594835.1504194

[18] T. D. Han and T. S. Abdelrahman, "hicuda: a high-level directive-based language for gpu programming," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 52–61. [Online]. Available: http://doi.acm.org/10.1145/1513895.1513902

[19] S. zee Ueng, M. Lathara, S. S. Baghsorkhi, and W. mei W. Hwu, "CUDA-Lite: Reducing GPU Programming Complexity," in *LCPC08*, 2008.

[20] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU Compiler for Memory Optimization and Parallelism Management," in *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 86–97. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806606

[21] P. Group, "PGI CUDA Fortran Compiler." [Online]. Available: http://www.pgroup.com/resources/cudafortran.htm

[22] S. Lee, V. Grover, M. M. T. Chakravarty, and G. Keller, "Gpu kernels as data-parallel array computations in haskell," 2009.

[23] R. Garg and J. N. Amaral, "Compiling Python to a Hybrid Execution Environment," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010, pp. 19–30. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735695

[24] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: Compiling an Embedded Data Parallel Language," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 47–56. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941562

[25] L. G. Valiant, "A Bridging Model for Parallel Computation," *Commun. ACM*, vol. 33, pp. 103–111, August 1990. [Online]. Available: http://doi.acm.org/10.1145/79173.79181

[26] M. Harris, S. Sengupta, and J. D. Owens, "Parallel Prefix Sum (Scan) with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, August 2007, ch. 39, pp. 851–876.

[27] D. Cederman and P. Tsigas, "A Practical Quicksort Algorithm for Graphics Processors," in *Proceedings of the 16th annual European symposium on Algorithms*, ser. ESA '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 246–258.