PREPRINT: To be presented at ICFP '12, September 9–15, 2012, Copenhagen, Denmark.

Nested Data-Parallelism on the GPU

Lars Bergstrom
University of Chicago
larsberg@cs.uchicago.edu

John Reppy
University of Chicago
jhr@cs.uchicago.edu

Abstract

Graphics processing units (GPUs) provide both memory bandwidth and arithmetic performance far greater than that available on CPUs but, because of their *Single-Instruction-Multiple-Data* (SIMD) architecture, they are hard to program. Most of the programs ported to GPUs thus far use traditional data-level parallelism, performing only operations that operate uniformly over vectors.

NESL is a first-order functional language that was designed to allow programmers to write irregular-parallel programs — such as parallel divide-and-conquer algorithms — for wide-vector parallel computers. This paper presents our port of the NESL implementation to work on GPUs and provides empirical evidence that nested data-parallelism (NDP) on GPUs significantly outperforms CPU-based implementations and matches or beats newer GPU languages that support only flat parallelism. While our performance does not match that of hand-tuned CUDA programs, we argue that the notational conciseness of NESL is worth the loss in performance. This work provides the first language implementation that directly supports NDP on a GPU.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (Functional) Programming, Concurrent, distributed, and parallel languages; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Languages, Performance

Keywords GPU, GPGPU, NESL, nested data parallelism

1. Introduction

Graphics processing units (GPUs) provide large numbers of parallel processors. For example, the NVIDIA Tesla C2050 has 14 multiprocessors, each with 32 cores, for 448 total cores. This card provides over 1200 GFLOPS, far more than the approximately 50 GFLOPS available from a typical Intel quad-core i7 processor. While the GPU cores provide very good integer and floating point throughput, they are very limited compared to a general-purpose CPU. They only achieve peak performance when all cores are executing the same instructions at the same time. This model works well for a wide variety of arithmetically intense, regular parallel problems, but it does not support *irregular* parallel problems — problems characterized by abundant parallelism but that have non-uniform problem subdivisions and non-uniform memory access, such as divide-and-conquer algorithms.

Most GPU programming is done with the CUDA [NVI11b] and OpenCL [Khr11] languages, which provide the illusion of C-style general-purpose programming, but which actually impose restrictions. There have been a number of efforts to support GPU programming from higher-level languages, usually by embedding a data-parallel DSL into the host language, but these efforts have been limited to regular parallelism [CBS11, MM10, CKL+11].

The current best practice for irregular parallelism on a GPU is for skilled programmers to laboriously hand code applications. The literature is rife with implementations of specific irregular-parallel algorithms for GPUs [BP11, DR11, MLBP12, MGG12]. These efforts typically require many programmer-months of effort to even meet the performance of the original optimized sequential C program.

GPUs have some common characteristics with the wide-vector supercomputers of the 1980's, which similarly provided high-performance SIMD computations. NESL is a first-order functional language developed by Guy Blelloch in the early 1990's that was designed to support irregular parallelism on wide-vector machines. NESL generalizes the concept of data parallelism to *nested data-parallelism* (NDP), where subcomputations of a data-parallel computation may themselves be data parallel [BS90, Ble96, PPW95, CKLP01]. For example, the dot product of a sparse vector (represented by index/value pairs) and a dense vector is a data-parallel computation that is expressed in NESL using a parallel map comprehension (curly braces) and a parallel summation reduction function:

```
function svxv (sv, v) =
    sum ({x * v[i] : (x, i) in sv});
```

Using this function, we can define the product of a sparse matrix (represented as a vector of sparse vectors) with a dense vector as:

```
function smxv (sm, v) =
    { svxv(row, v) : row in sm }
```

This function is an example of a nested data-parallel computation, since its subcomputations are themselves data-parallel computations of irregular size.

As described, NDP is not well-suited to execution on SIMD architectures, such as wide-vector supercomputers or GPUs, since it has irregular problem decomposition and memory access. Blelloch's solution to this problem was the *flattening* transformation, which vectorizes an NDP program so that the nested array structures are flat and the operations are SIMD [BS90, Kel99, PPW95, Les05]. This compilation technique allows NDP codes to be run on vector hardware, but it has not yet been applied to GPUs.

In this paper, we describe a port of the NESL language to run on GPUs. Our implementation relies on the NESL compiler to apply the flattening transformation to the program, which produces a vectorized stack-machine code, called VCODE. We use a series

```
function quicksort(a) =
  if (#a < 2) then a
  else let
    p   = a[#a/2];
    lt  = {e in a | e < p};
    eq  = {e in a | e == p};
    gt  = {e in a | e > p};
    r = {quicksort(v) : v in [lt, eq, gt]};
    in r[0] ++ r[1] ++ r[2];
```

Figure 1. NESL implementation of Quicksort, demonstrating irregular parallelism in a divide-and-conquer algorithm.

```
function quicksort' (as) =
  if all(#as < 2) then as
  else let
    ps  = {a[#a/2] : a in as};
    lts = {{e in es | e < p} : p in ps; es in as};
    eqs = {{e in es | e == p} : p in ps; es in as};
    gts = {{e in es | e > p} : p in ps; es in as};
    rs  = quicksort' (flatten ([lts, eqs, gts]));
    in rs;
```

Figure 2. Flattening-inspired implementation of Quicksort, providing a high-level overview of the effect of the code and data transformations.

of code optimizers and libraries to transform this intermediate language for efficient execution on GPUs. This paper makes the following contributions:

1. We demonstrate that a general purpose NDP language, such as NESL, can be implemented efficiently on GPUs. By allowing irregular parallel applications to be programmed for a GPU using NESL, we effectively move the requirement for highly-skilled GPU programmers from the application space to the language-implementation space.
2. We explain the performance requirements of modern GPU hardware and describe the techniques and data structures that we developed to tune the performance of the underlying vector primitives required to implement NDP on a GPU.
3. We demonstrate that our preliminary implementation provides performance better than many of the flat data-parallel languages, which illustrates the potential of this compilation approach.

The remainder of the paper is organized as follows. In the next section, we describe the programming and execution model using quicksort as an example. Then, we provide an overview of GPU hardware and the CUDA language. Section 4 is a detailed description of our implementation and the match between the requirements of NESL on the vector machine and the CUDA language and its associated libraries, focusing on the design decisions required for high performance on GPUs. Since directly implementing the vector hardware model was not sufficient for our performance targets, Section 5 describes additional optimizations we perform. After we introduce several related systems in some depth in Section 6.2, we compare the performance of our system to these systems. Finally, we cover some related work and conclude.

The source code for our implementation and all benchmarks described in this paper are available at: `http://smlnj-gforge.cs.uchicago.edu/projects/neslgpu`.

2. NESL

NESL is a first-order dialect of ML that supports nested data-parallelism [BCH+94]. It provides the ability to make data-parallel function calls across arbitrarily nested data sequences. A standard example of NDP computation in NESL is the quicksort algorithm [Ble96], which is given in Figure 1. The recursive calls to the `quicksort` function lead to irregular parallel execution in general. That is, the compiler cannot know how the data will be partitioned, so it cannot statically allocate and balance the workload.

The NESL compiler supports such irregular NDP computations by transforming the code and data into a representation where nested arrays have been *flattened*. The result of this transformation is *VCODE*, which is code for a stack-based virtual vector machine [BC90]. This language executes in an interpreted environment on the host machine, calling primitives written in the *C Vector Library* (CVL) [BC93]. In this section, we explain the NESL compilation process using quicksort as a running example.

2.1 Flattened quicksort

Figure 2 shows an idealized version of this flattened program in syntax similar to that of NESL. This version is not actually emitted by the NESL compiler, but is helpful for understanding the transformation.

Flattening transforms the `quicksort` function from operating over a single vector at a time into a new version that operates over a nested vector of vectors. In a language such as C, this structure might be represented with an array of pointers to arrays. In the implementation of NESL, however, nested vectors are represented in two parts. One part is a flat vector containing the data from all of the different vectors. The other part is one or more segment descriptors (one per level of nesting). Segment descriptors are vectors that contain the index and length information required to reconstruct the nested vectors from the flat data vector. To cope with the change in representation, the body of `quicksort` must also be changed. Each operation that previously operated on a scalar value is lifted to operate on a vector and vector operations are lifted to operate on nested vectors.

The scalar `p`, which was a scalar value holding the single pivot element from the input vector `a` in the original program becomes the vector `ps`, holding all of the pivots from each of the vectors inside of the nested vector `as`. The vectors that previously held the `lt`, `eq`, `gt`, and `r` vectors from `a` are now turned into nested vectors holding all of the vectors of corresponding elements from the vectors represented by `as`. The termination condition for this function, which was previously that the vector `a` is of length less than 2, becomes a check to ensure that *all* of the vectors in `as` have length less than 2.

The `flatten` operator, which is used to combine the vectors for the recursive call to `quicksort'`, is of type $[[\alpha]] \rightarrow [\alpha]$. It removes one level of nesting by appending the elements of each of the top-level vectors.

2.2 NESL runtime

The output of the NESL compiler is the flattened program translated into a virtual-machine code called *VCODE*. The NESL runtime system can be viewed as consisting of a VCODE interpreter that runs on a CPU and a machine-specific vector library that runs on a device. In some cases, the vector device might be the CPU, but in our case it is the GPU. Because most of the computational load is in the vector operations, performance is dominated by the efficiency of the vector library.

2.2.1 VCODE

VCODE is a stack-based language that is intended to be run on a host computer with vector operations performed on a vector machine. Figure 3 contains a small fragment of the actual VCODE

```
FUNC QUICKSORT_13          function entry
  COPY 1 0                 stack manipulation
  CALL PRIM-DIST_37        function call
  COPY 2 1
  COPY 1 2
  CALL VEC-LEN_13
  CONST INT 2              push the vector [2]
  COPY 1 2
  CALL PRIM-DIST_6
  COPY 1 2
  POP 1 0                  stack manipulation
  < INT                    element-wise integer comparison
...
```

Figure 3. A small section of the over 1500 lines of VCODE corresponding to the original quicksort example. This section is the beginning of the flattened version, which determines whether the lengths of all of the segments passed in are less than 2.

generated by the NESL compiler from the quicksort program in Figure 1. We have annotated some of the instructions with their meaning. There are four categories of VCODE instructions:

Vector instructions. These include both element-wise vector operations (such as the `< INT` in Figure 3) as well as various reduction, prefix-scan, and permutation operations. These operations are executed by the vector device.

Stack instructions. These include various instructions for permuting, copying, and discarding stack elements, and are executed on the interpreter. All references to live data are on the stack and the VCODE interpreter uses these manage space automatically (there are no memory-management instructions in VCODE). Stack values are represented by a device-specific handle to the data paired with type information. Underlying data values are *not* transferred between the CPU and device except when required for I/O.

Control instructions. These include function entries calls, and returns, as well as conditionals and the `CONST` instruction. These instructions are executed on the CPU and do not directly affect the device (except for the `CONST` instruction, which allocates a vector on the device).

Other instructions. VCODE includes instructions for I/O, system initialization and shutdown, and various other miscellaneous operations.

2.2.2 CVL

The C Vector Library (CVL) is a set of C library functions callable from a host machine that implement the VCODE vector operations and is the primary hook between the host machine and hardware on which the program is executing. This library is well-documented in its manual [BC93] and has been ported to many of the high-performance parallel computers of the 1990's. It also has a sequential C implementation for execution on scalar uniprocessors. In this work, we have implemented a version of the CVL library in CUDA.

3. GPU hardware and programming model

Graphics processing units (GPUs) are high-performance parallel processors that were originally designed for computer graphics applications. Because of their high performance, there has been growing interest in using GPUs for other computational tasks. To support this demand, C-like languages, such as CUDA [NVI11b] and OpenCL [Khr11] have been developed for general-purpose programming of GPUs. While these languages provide a C-like expression and statement syntax, there are many aspects of their programming models that are GPU-centric. In this paper, we focus on the CUDA language and NVIDIA's hardware, although our results should also apply to OpenCL and other vendors' GPUs.

A typical GPU consists of multiple streaming multiprocessors (SMP), each of which contains multiple computational cores. One of the major differences between GPUs and CPUs is that the memory hierarchy on a GPU is explicit, consisting of a global memory that is shared by all of the SMPs, a per-SMP local memory, and a per-core private memory. An SMP executes a group of threads, called a *warp*, in parallel, with one thread per computational core. Execution is *Single-Instruction-Multiple-Thread* (SIMT), which means that each thread in the warp executes that same instruction. To handle divergent conditionals, GPUs execute each branch of the conditional in series using a bit mask to disable those threads that took the other branch. An SMP can efficiently switch between different warps, which allows it to hide memory latency. GPUs have some hardware support for synchronization, such as per-thread-group barrier synchronization and atomic memory operations.

3.1 CUDA

CUDA is a language designed by NVIDIA to write parallel programs for their GPU hardware [NVI11b]. It can be viewed as an extension of C++ and includes support for code synthesis using templates. In CUDA, code that runs directly on the GPU is called a *kernel*. The host CPU invokes the kernel, specifying the number of parallel threads to execute the kernel. The host code also specifies the configuration of the execution, which is a 1, 2, or 3D grid structure onto which the parallel threads are mapped. This grid structure is divided into blocks, with each block of threads being mapped to the same SMP. The explicit memory hierarchy is also part of the CUDA programming model, with pointer types being annotated with their address space (*e.g.*, global vs. local).

3.2 Key programming challenges on GPUs

The various features of the GPU hardware and programming models discussed above pose a number of challenges to effective use of the GPU hardware. Our implementation largely addresses these challenges, which allows the programmer to focus on correctness and asymptotic issues instead of low-level hardware-specific implementation details.

Data transfer Communication between the host CPU and GPU is performed over the host computer's interconnect. These interconnects are typically high-bandwidth, but not nearly as high in bandwidth as is available on the card itself. For example, a PCI-E 2.0 bus provides 16 GB/s of bandwidth, but the bandwidth between the NVIDIA Tesla C2050's SMPs and the global memory is 144 GB/s. Therefore, fast GPU programs carefully balance the number and timing of their data transfers and other communications between the host and device.

Memory access Within a GPU kernel, access to the global memory is significantly slower than access to local or private memory. In fact, naive kernels that rely excessively on global memory are often slower than native CPU speeds. Furthermore, GPU global memory performance is very sensitive to the patterns of memory accesses across the warp [NVI11a]. Lastly, the memory characteristics differ between cards, so a kernel that is tuned for one card may not run well on another.

Divergence Threads within a warp must execute the same instruction each cycle. When execution encounters divergent control flow, the SMP is forced to execute both control-flow paths to the point where they join back together. Thus care must be taken to reduce the occurrence of divergent conditionals.

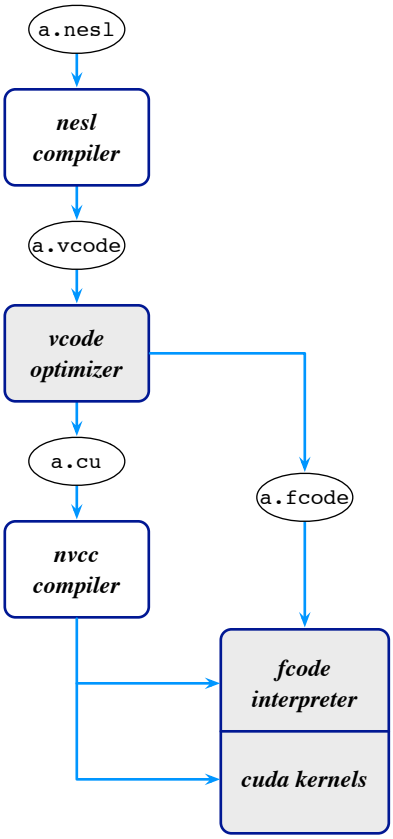

Figure 4. The NESL/GPU toolchain. The shaded portions represent our contributions.

No recursive calls Recursion is not, in general, permitted in GPU kernels.[1] This limitation means that any program with recursive calls must be transformed into a non-recursive one. This can be done by CPS converting and using a trampoline (as is done by the OptiX library [PBD+10]), by explicitly managing the return stack on the GPU [YHL+09], or by managing control flow on the host CPU. Our NESL implementation does the latter.

4. Implementation

Figure 4 shows the structure of our implementation. It takes a NESL program (`a.nesl`) and compiles it to VCODE (`a.vcode`) using the preexisting NESL compiler from CMU. We then optimize the VCODE to produce specialized CUDA kernels (`a.cu`) and fused VCODE (`a.fcode`) for the program. Finally, the code is executed by a modified version of the VCODE host interpreter, using our own CUDA-based implementation of the CVL.

The remainder of this section describes our implementation of the CVL, which consists of over 200 kernel functions, split up into segmented and unsegmented versions of element-wise operations, reductions, scans, permutations, and miscellaneous conversion and management operations. This implementation is the end product of many iterations of design and careful performance profiling. Our implementation passes all of the single-precision NESL regression tests.[2] We leave a discussion of the VCODE optimizer and supporting interpreter modifications to Section 5.

4.1 Implementation basics

CVL uses a fixed-sized heap and performs garbage collection when there is insufficient space remaining for a requested allocation. All data values in our implementation are 32-bits long — `float`, `integer`, and `bool` values. The VCODE interpreter keeps reference counts to track the lifetime of vectors. We augmented each of our CUDA calls with information about opportunities to reuse a source vector for the destination values for cases where they are both the same length and there is no concurrency issue with reusing the storage space. This optimization results in a significant reduction of the maximum memory footprint.

4.2 Segment Descriptors

As mentioned in Section 2, segment descriptors are used in the flattened data representation to describe the original structure of the data. For example, consider the nested vector $[[4],[5,6,7],[8,9]]$. This vector can be represented by the flat data vector $[4,5,6,7,8,9]$ paired with the segment descriptor $[1,3,2]$, which indicates that the first segment is one element long, the second segment is three elements long, and the third segment is two elements long. Although simple, this representation is terrible for execution of segmented operations on GPUs.

Many kernels need to know for a given thread which segment it is in. Using the representation described above would mean that each thread would be scanning the segment descriptor to determine its segment, which would result in excessive memory traffic. To avoid this problem, we expand segment descriptors to the length of the underlying data vector, with the ith element of the segment descriptor holding the segment number that the ith data element belongs to. For our example, the descriptor is $[1,2,2,2,3,3]$. With this representation, each thread has constant-time access to its segment and the access pattern is optimal for global memory bandwidth.

This representation can be further improved. Some kernels, such as the extraction and permutation kernels, need to know how far offset a given segment is within the data, which requires knowing the length of the segment. For example, to extract the second element from the third segment in the data vector, the kernel needs to compute the total number of elements from the first two segments and then add two. Because these operations are common in a large number of kernels, we concatenate the original segment lengths vector to the end of the segment descriptor vector. Returning to this section's example, the full descriptor is $[1,2,2,2,3,3,1,3,2]$.

These representation choices are the result of experimentation with three different alternatives. The worst — lengths only — resulted in individual kernel calls taking multiple *seconds*, even after optimization. Using a flags vector increased performance over the lengths-only representation and used less space (since multiple boolean flags can be compacted into single words), but were slower than our final representation because of the need to compute the per-element segment index in many kernels. After settling on this format and carefully tuning each of the different types of kernel calls, those same calls take no more than a couple of hundred microseconds, even on very-large input data. The remaining subsections discuss implementation details and trade-offs in these kernel calls.

4.3 Element-wise kernels

The element-wise kernels perform a unary or binary operation uniformly across the elements of its arguments, independent of

[1] Recursion is supported on some newer cards, but only for kernels that may be called from other kernel functions — not the host.

[2] Our implementation does not yet support double-precision codes.

their nesting structure. These kernels are the only ones that are unaffected by the choice of segment descriptor representation, but they do have the property that they are memory bound, since their inputs and output are stored in global memory. This property means that they are very sensitive to the number of warps per thread block. With too many warps per block, warps will be idle waiting on computing resources, whereas with too few warps per block, the SMP will be idle waiting for memory operations to complete, This problem is the *memory access* challenge discussed in Section 3.2.

We tested block sizes between 32 and 4096 threads and found that blocks of 256 threads (8 warps) provided the best overall performance, by up to a factor of 8. At lower numbers of threads, the blocks were too small to keep the hardware busy, since most of our kernels are memory-bound. Increasing beyond 256 threads per block caused low occupancy rates. At 8 warps per block, this breakdown allows roughly 12 cycles to compute the address and make a request for memory from each thread before all of the memory requests have been issued and the first warp has been filled and is ready to execute. These results agree with those recommended by NVIDIA for memory-bound kernels [NVI11a].

In addition to the basic element-wise vector math operations, there is also an element-wise random number generation operation. While there are many random number generators available for GPUs, we have elected to execute this operation on the host CPU instead. This strategy makes it easier to compare our GPU implementation's results against CPU versions and the extra communication cost is not detrimental to performance, since the benchmarks only use random number generation during initialization.

4.4 Scans and reductions

Scan and reduction operators perform an associative binary operation over an input vector. The reduction operation results in a single value, whereas scan operations produce a vector containing all the intermediate results.

Our implementation uses the Thrust implementation of scan and reduction [HB11], which also supports the segmented scan operation directly. These implementations are based on the work by Sengupta *et al.* that produced efficient versions of the scan primitives tuned for GPUs [SHZO07]. Thrust also allows custom binary operators, so we also provide custom classes implementing the full set of binary operations available in the NESL language. Our segment descriptor format is directly usable as a flag vector for the segmented scan primitives by design.

While Thrust includes a native reduction operator, it does not include a segmented version of reduction. Our segmented reduction is implemented by first performing a segmented inclusive scan and then performing a second kernel call to extract all of the reduction sums. This second kernel call requires the locations of the final, reduced values from each of the segments.

In our initial implementation of this kernel, we computed — per warp — the corresponding offsets for each of the segments. But, this was far too slow on vectors with more than a few thousand segments, as each warp needed to read every element from the lengths portion of the segment descriptor. Now, we perform a prefix sum of the lengths portion of the segment descriptor so that each thread can directly get the resulting value. For example, if we had a thousand segments, each of length 10, in the first version there would be `ceil(1000/32)=32` threads accessing and adding up, on average, half of those numbers. Worse, nearly all of those threads would access the first few elements, resulting in nearly the worst possible GPU memory access pattern. In the final version, we efficiently compute a vector corresponding to each of the offsets for each thread index and both avoid the multiple-access penalty and remove the portion of code from the start of each kernel where every other thread but the first was blocked, waiting for the first to compute all of the offsets for each of the threads in the warp.

In many cases, the segmented scan and reduction operators are called on single-segment vectors (*i.e.*, flat or non-nested vectors). Because of the overhead of creating and initializing extra data structures to handle multiple segments and the extra bounds checking in the segmented operators, we wrote custom implementations of the segmented operations for the single-segment case. These specialized implementations achieve a nearly 20% speedup over the general versions.

4.5 Permutation and vector-scalar operations

Permutation operators shuffle elements from their locations in a source vector to a target vector, sometimes incorporating default values for unmapped or extra elements. Vector-scalar operations extract or replace values in a vector or create a vector from a base value and an optional stride value. When a stride is provided, the resulting vector elements are equal to the sum of the base value and the product of the element index and the stride value.

These operators rely on the segment descriptor information that allows individual threads to avoid each reading the vector of lengths repeatedly. Similar to the previous section, this requirement means that we often perform a `+-scan` of the segment descriptor lengths to provide the per-segment offsets, sometimes for both source and target segment descriptors if they differ and are required. The operations requiring these data are infrequent relative to the cost in both memory and time of always computing these scanned segment lengths, so we do not just maintain both formats.

However, computing indexing information through composition of efficient scan-vector operations is critical for high-performance kernel code. In every case that we tried to compute indexed offsets in a kernel by hand on a per-warp or per-thread basis, there arose input data edge cases that pushed execution times for each of those inefficient kernel calls into the range of seconds.

4.6 Miscellaneous operations

The general facilities of the implementation include memory allocation, windowing, I/O, host/vector hardware data transfer, and timing operators. During the initialization process, we currently only select a single GPU — the one with the estimated highest performance.

There are also library functions that directly implement sorting. These *rank* operations take a vector of numbers and return a vector of integer indices indicating the target index that the corresponding number would take on in a sorted vector. We use the Thrust library's radix sort to implement the rank operations. The segmented rank operation is implemented by multiple invocations of the unsegmented radix sort operation, once per segment. We do not rely on the *rank* operation in any of our benchmark code, but provide it for API compatibility.

The *index* operations are heavily used and fill in vectors with default values. The values are not only constants, but can also support a stride, allowing the value at an index to be a function of an initialization value and a stride factor. Again, this is an operation that benefited from both the new segment descriptor format that provided the index number associated with an element as well as an extra data pass where we generate the offset into the segment of each element. This offset allows the computation of the value at an index to just be a simple multiply and add. Similar to the scan and reduction operators, this operation is frequently called in its segmented form but with a single vector. There is a significant performance gain by optimizing the case of a single segment since all elements share the same stride and index and can be loaded from their fixed location in GPU memory once per warp and shared across all of the threads.

The final operation with an interesting implementation is the creation of a segment descriptor, which is performed even more often than the indexing operation — once per vector, unless the segment descriptor can be reused from a previous vector. Vectors are immutable in the NESL implementation, so segment descriptors are frequently shared and reused in the generated VCODE. Creation of the segment descriptor takes a vector with the lengths of each segment. For example, the vector `[1,3,3]` describes an underlying vector of seven elements divided into three segments. We allocate space for the underlying vector's length plus the length of the segment descriptor. In this case, that would be 10 elements. First, we fill the portion of the vector that is the length of the underlying data vector with zeros. In our example, there would now be a vector of seven zeros. Then, at the location of the start of each segment in the vector, we place the index of that segment. In this case, the beginning of the vector is now: `[1,2,0,0,3,0,0]`. An inclusive `max-scan` carries the segment indices over each of the zero-initialized elements quickly, resulting in: `[1,2,2,2,3,3,3]`. Finally, we copy the original lengths input to the end of the vector for use in the numerous kernels that require it. The final segment descriptor is: `[1,2,2,2,3,3,3,1,3,3]`. This operation is executed as a single kernel invocation.

There are many other more straightforward ways to fill in this data, such as having one thread per element that computes its correct index, but, as in the other examples that required the `+-scan` of the segment descriptor lengths, any kernel implementation that requires a large number of kernel threads to touch the same piece of memory will perform so poorly that an individual kernel call will take longer than the entire remainder of the benchmark.

5. Optimization

A straightforward porting of the NESL implementation to GPUs suffers from some obvious performance issues. The VCODE produced by the NESL compiler is sub-optimal. It includes many trivial utility functions and a significant amount of stack churn. Furthermore, a straightforward port of the CVL library to GPUs requires an individual CUDA kernel invocation for each computational VCODE operation. This property adds significant scheduling overhead and reduces memory locality, since the arguments and results of the operations must be loaded/stored in global memory. For example, consider the following simple NESL function:

```
function muladd (xs, ys, zs) =
    {x * y + z : x in xs; y in ys; z in zs};
```

Figure 5 shows the unoptimized VCODE as produced by the NESL compiler. From this figure, we can see examples of the first two issues. The code includes trivial utility functions (*e.g.*, `ZIP-OVER_8`) and is dominated by stack manipulations; many of which end up computing the identity. In addition, the multiplication is run as a separate operation from the addition, which means that the intermediate result (*i.e.*, the `x*y` value) must be stored in global memory and then reloaded to perform the addition.

To improve performance of our system, we have implemented an optimizer that takes the VCODE produced by the NESL compiler and produces an optimized program in an extension of VCODE that we call *FCODE* (for *Fused vCODE*). As shown in Figure 4, this optimization fits inbetween the NESL compiler and the VCODE interpreter. We describe this optimizer in the rest of this section and discuss its impact on performance in Section 6.4.

5.1 VCODE optimizations

The VCODE optimizer consists of five phases. The first two of these address inefficiencies in the VCODE generated by the NESL compiler. The first phase is the inliner, which visits functions in reverse topological order inlining calls to other VCODE operations.

```
FUNC MULADD1_7
CPOP 2 4
CPOP 2 4
CPOP 2 4
CALL ZIP-OVER_8
CALL ZIP-OVER_10
COPY 1 3
CPOP 1 4
CPOP 1 4
CPOP 1 4
COPY 1 3
POP 1 0
* INT
CPOP 1 3
CPOP 1 3
POP 1 0
+ INT
RET
```

```
FUNC ZIP-OVER_8
CPOP 2 2
CPOP 2 2
POP 1 1
CPOP 1 2
CPOP 1 2
CPOP 1 2
RET

FUNC ZIP-OVER_10
CPOP 2 3
CPOP 3 2
POP 1 2
CPOP 1 3
CPOP 1 3
CPOP 2 2
RET
```

Figure 5. Unoptimized VCODE for the `muladd` function

The inliner does not inline recursive functions and uses a size metric to limit code growth. Our size metric is the number of computational instructions in the function.

Once we have performed inlining, the next phase converts the stack machine code into an expression language with let-bound variables. In this representation, each computation is bound to a unique variable. Continuing with our example, the `muladd` function is represented as follows:[3]

```
function MULADD1_7 (p0, p1, p2, p3)
    let t033 = p1 * p2
    let t034 = t033 + p3
    in
      RET (p0, t034)
```

This conversion has the effect of compiling away stack manipulation instructions (*i.e.*, `POP`, `COPY`, and `CPOP`). When we convert back to the stack machine representation, we are careful to avoid redundant and unnecessary stack operations, so the final result is much more compact. For example, the resulting code for the `muladd` function is:

```
FUNC MULADD1_7
  CPOP 2 1
  * INT
  CPOP 1 1
  + INT
  RET
```

5.2 Fusion

While the VCODE optimizations produce much more compact programs, they do not address the most significant performance issue, which is the use of individual kernel invocations for each computational instruction. For example, in the `muladd` code, a kernel invocation is used to perform the element-wise multiplication on the `xs` and `ys` to produce an intermediate result array. Then a second kernel invocation is used to add the result of the first with the `zs` array. For element-wise operations, this pattern is extremely inefficient, since we incur kernel invocation overhead for relatively small kernels that are dominated by global memory traffic. As has been observed by others, the flattening approach to implementing NDP requires *fusion* to be efficient [Kel99, Cha93]. VCODE does not have a way to express fused operators, so we must leave the confines of the VCODE instruction set and extend the interpreter.

[3] Parameter `p0` is the segment descriptor for the result vector.

Our VCODE optimizer identifies element-wise computations that involve multiple operations and replaces them with synthesized *superoperators*. This approach is similar to Proebsting's superoperators [Pro95] and Ertl's super instructions [Ert01], with the main difference being that we introduce superoperators for any eligible subcomputation, independent of its frequency.

In our implementation, fusion is a two-step process. The first step is the reducer, which replaces variables in argument positions with their bindings. The reducer limits its efforts to element-wise operations and "flat" constants. The reducer also eliminates unused variables and function parameters. After reduction, the `muladd` function consists of a single, multi-operation expression:

```
function MULADD1_7 (p0, p1, p2, p3, p5)
    let t034 = (p1 * p3) + p5
    in
      RET (p0, t034)
```

The second step is to identify the unique fused expressions and to lift them out of the program as superoperators, as is shown in the following code:

```
fused OP0 ($0 : INT, $1 : INT, $2 : INT) =
    ($0 * $1) + $2

...

function MULADD1_7 (x025, x026, x027, x028)
    let x029 = OP0 (x026, x027, x028)
    in
      RET (x025, x029)
```

The two benefits of the fused kernels over the optimized VCODE are reductions in the number of kernel calls and the number of global memory loads and stores. For this example, we reduce the number of kernel calls by four for each dynamic instance of the `MULADD1_7` function and we reduce the number of global loads and stores by one each per element of the argument vectors.

Limitations Our implementation currently fuses only element-wise operations. Fusing map-reduce combinations would further reduce the number of kernel calls and global memory accesses, but we have not yet implemented this feature.

5.3 Code generation

The final phase of the optimizer is code generation, which is responsible for both converting the expression representation back to stack-machine format and generating CUDA C code for the fused superoperators. First, the optimized VCODE is transformed to remove all identified opportunities for fusion with a call to a superoperator. In this example, we produce the following code:

```
FUNC MULADD1_7
  FUSED 0
  RET
```

where the "`0`" is the ID of the fused operation that is being invoked. Then, we generate custom CUDA kernels for each fused superoperator. The corresponding kernel for the fused operation in this example is shown in Figure 6.

5.4 Calling the custom kernels

Alongside each of these CUDA kernels, we generate both a host C function and a set of data structures. The data structures contain summary information about the kernel, such as its arity, parameter and return types, vector size information, and whether or not last-use input parameters can be reused as the output storage. We have modified the VCODE interpreter to handle fused operators, such as `FUSED 0` in the example. When the interpreter hits a fused kernel call it runs code to perform the proper allocations, argument checks, and invoke the kernel through the host C function.

```
__global__ void fused0Kernel (MAXALIGN *data,
  int dst, int s0, int s1, int s2,
  int len, int scratch)
{
  int addr = blockDim.y * blockIdx.y
      + blockDim.x * blockIdx.x + threadIdx.x;
  if (addr < len) {
    int *pDst = (int*)(&data[dst]);
    int *pSrc0 = (int*)(&data[s0]);
    int *pSrc1 = (int*)(&data[s1]);
    int *pSrc2 = (int*)(&data[s2]);

    pDst[address] = pSrc0[addr] * pSrc1[addr]
      + pSrc2[addr];
  }
}
```

Figure 6. Kernel for fused operation `OP0`

6. Evaluation

To evaluate the effectiveness of our approach to implementing NESL on GPUs, we compare the performance of our system to CPU-based systems that use flattening to implement NDP and to GPU-based systems that support flat data parallelism. Specifically, we compare with NESL running on a single CPU, Data Parallel Haskell (DPH) running on 8 cores, Copperhead running on a GPU, and hand-written CUDA code running on a GPU. We choose the best available implementation of each benchmark for each measured system. We take this approach because each platform varies in compiler technology, target hardware, and implementation tuning, resulting in unfair penalties to platforms with different hardware tradeoffs or where benchmark authors were no longer maintaining tuned algorithms for the evolving GPU hardware.

While our implementation does not achieve the level of performance of hand-tuned CUDA programs, our results are better than other high-level NDP and flat data-parallel programming languages. Furthermore, NESL programs are significantly smaller than the corresponding hand-tuned CUDA (typically a factor of 10 smaller) and require no direct knowledge of GPU idiosyncrasies. Thus, we make GPUs applicable to a wider range of parallel applications and a wider range of programmers.

6.1 Experimental framework

Our benchmark platform has an Intel i7-950 quad-core processor running at 3.06GHz, with hyper-threading enabled for a total of eight cores available for parallel execution. Our GPU is an NVIDIA Tesla C2050, which has 14 SMPs, each with 32 cores for a total of 448 cores, and 3GB of global memory. For most experiments, we ran Linux x86-64 kernel version 3.0.0-15 with the CUDA 4.1 drivers, but for one test we used Microsoft Windows 7 (also with the CUDA 4.1 drivers).

We report the wall-clock execution time of the core computation, excluding initialization and result verification times. We exclude the latter, because those times differ widely between different platforms. Each benchmark was run at least 20 times at each size configuration and we report the mean.

6.2 Comparison systems

We compare our NESL/GPU implementation with a number of different systems, which are described in this section. To illustrate the programming models of these systems, we give code for the data-parallel computation of the dot product. In NESL, this code is:

```
function dotp (xs, ys) =
  sum ({ x*y : x in xs; y in ys })
```

```
import Data.Array.Parallel
import Data.Array.Parallel.Prelude.Double
    as D

dotp' :: [:Double:] -> [:Double:] -> Double
dotp' xs ys = D.sumP (zipWithP (*) xs ys)
```

Figure 7. Data Parallel Haskell implementation of dot product

6.2.1 NESL/CPU

We measured the CPU-based version of NESL running on a single core of our test machine. For these experiments, we ran the output of our optimizer with fusion turned off (*i.e.*, the VCODE after inlining and elimination of redundant stack operations). The CPU version of NESL uses an implementation of CVL that is quite efficient. The code uses macros for loop unrolling and inlining, which exposes multiple adjacent operations that the C compiler optimizes into SSE instructions.

6.2.2 Data Parallel Haskell

Data Parallel Haskell (DPH) also uses flattening in its implementation of NDP [CLPK08]. This extension of the Glasgow Haskell Compiler (GHC) [GHC] implements a subset of the Haskell language with strict evaluation semantics. On a per-module basis, code restricted to this subset and using the appropriate types will be flattened. DPH does not support GPU execution, but it does support parallel execution on multicore systems. For our benchmarks, we report the DPH performance on all eight cores of our test machine.[4] We measured the DPH implementation as of February 2012 and GHC Version 7.4.1.

Dot product in DPH The DPH version of our dot-product example is shown in Figure 7. At its core, the last line is very similar to the NESL code for this example.

6.2.3 Copperhead

Copperhead [CGK11] is an extension of the Python language that provides direct language support for data parallel operations on GPUs. It is limited to element-wise operations and reductions (*i.e.*, it does not support NDP computations). Sources intended to run on the GPU (and optionally also the CPU) are annotated with an `@cu` tag, indicating that only the subset of Python that can be compiled to the GPU may occur in the following definitions. Using a special `places` keyword, those annotated definitions can be executed either on the GPU or CPU.

We are using the Copperhead sources as of February, 2012, available from: `http://code.google.com/p/copperhead/`. These sources are compiled against Python 2.7.1, Boost 1.41, PyCuda 2011.1.2, and CodePy 2011.1. The Copperhead project is still under active development and the compiler is not yet mature enough to implement all of the benchmarks used in this paper.

Dot product in Copperhead The dot-product example is very straightforward to write in Copperhead and appears in Figure 8. The `@cu` annotation on the `dot_product` function tells Copperhead to compile the code to run on the GPU. When the `dot_product` function is invoked, its arguments are converted to CUDA representation and transferred to the GPU. Likewise, the result of the function is transferred back to the CPU and converted to Python representation automatically.

[4] We also measured the performance on four cores without hyper-threading, but found that the eight-core performance was better.

```
from copperhead import *

@cu
def dot_product(xs, ys):
    return sum(map(lambda x,y: x*y, xs, ys))
```

Figure 8. Copperhead implementation of dot product

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512

__global__ void dot (int *xs, int *ys, int *res)
{
  __shared__ int temp[THREADS_PER_BLOCK];
  int index = threadIdx.x + blockIdx.x
      * blockDim.x;
  temp[threadIdx.x] = xs[index] * ys[index];

  __syncthreads();

   if (0 == threadIdx.x) {
     int sum = 0;
     for (int i = 0; i < THREADS_PER_BLOCK; i++)
       sum += temp[i];
     atomicAdd (res , sum);
   }
}

int main(void)
{
  dot<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>
    (dev_xs, dev_ys, dev_res);
}
```

Figure 9. Basic CUDA implementation of dot product

6.2.4 CUDA

We also measured hand-coded implementations of our benchmarks to provide a measure of the best possible performance that one might expect for these benchmarks. These implementations represent significant programmer effort, but make very efficient use of the GPU.

Dot product in CUDA Figure 9 gives CUDA kernel code for the dot product. This program performs basic blocking and uses fast shared memory on the GPU, but is slower than the optimized version available in CUBLAS.[5] Even a simple implementation in CUDA of this small example is significantly more complicated and verbose than the code on any of the other platforms.

6.3 Benchmarks

Table 1 summarizes our benchmark results. As would be expected, hand-coded CUDA code outperforms the other implementations on all benchmarks. But the NESL/GPU implementation is the second fastest system on almost all of the benchmarks and also has the shortest programs.

All of the NESL results reported in this section are for programs optimized as described in Section 5. The NESL/CPU programs do not include fused kernel operations because those were implemented only for the NESL/GPU backend.

[5] For our performance experiments, we measured the faster CUBLAS version of dot product.

	Problem Size	Lines of code				Execution time (ms)				
	(# elements)	NESL	CUDA	DPH	Copper	NESL/CPU	NESL/GPU	CUDA	DPH	Copper
Dot Product	10,000,000	**8**	80	39	31	27	3.0	< **1.0**	12	710
Sort	1,000,000	**12**	136	52	46	230	259	**3.1**	2,360	230
Black-Scholes	10,000,000	**37**	337	*N/A*	*N/A*	8,662	163	**1.9**	*N/A*	*N/A*
Convex Hull	5,000,000	**25**	*unknown*	72	*N/A*	1,000	283	**269**	807	*N/A*
Barnes-Hut	75,000	**225**	1930	414	*N/A*	22,200	4,502	**40**	10,200	*N/A*

Table 1. Lines of non-whitespace or comment code for the benchmark programs, omitting extra testing or GUI code. Benchmark times are also reported, as the mean execution times in milliseconds (ms). Smaller numbers are better for both code and time.

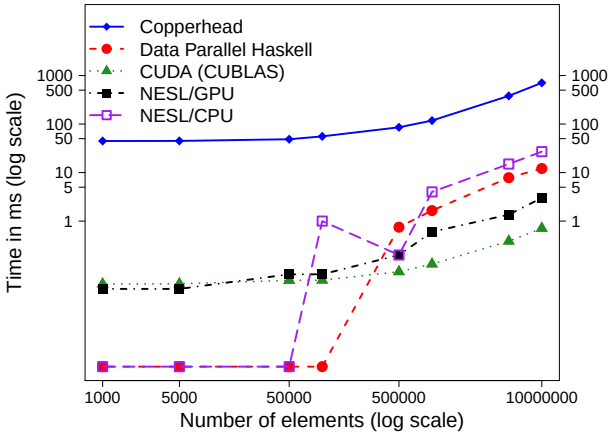

Figure 10. Dot product execution times (ms) for a range of problem sizes. Smaller times are better. Axes are log scale.

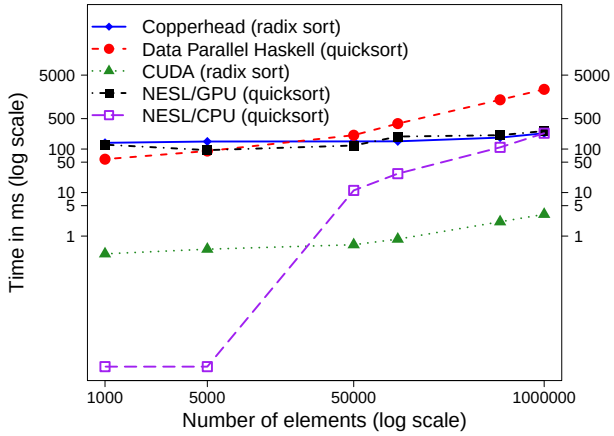

Figure 11. Sorting execution times (ms) for a range of problem sizes. Smaller times are better. Axes are log scale.

6.3.1 Dot product

The dot product of two vectors is the result of adding the pairwise multiplications of elements from each of the vectors. This small benchmark is interesting because it contains one trivial vector-parallel operation (multiplying each of the elements) and one that requires significant inter-thread communication (reducing those multiplied elements to a single value through addition).

In the summary data in Table 1, each vector contains 10,000,000 32-bit floating point. Figure 10 provides a more detailed breakdown for each system across many vector lengths. Because the performance of these systems vary widely, we use a logarithmic scale for the time axis in this plot.

The superb CUDA performance is provided by a highly-tuned implementation of dot product from the CUBLAS library. The NESL/GPU version of dot product also performs well. For vector sizes less than 5,000,000, the NESL/GPU version finishes faster than the finest resolution of the timing APIs. This good performance is owed to the use of a highly-tuned parallel reduction operator for the addition operation and relatively low additional overhead around the element-wise multiplication. The poor Copperhead behavior appears to be related to failing to take full advantage of the parallel hardware.

6.3.2 Sorting

For the sorting benchmarks, we selected the best algorithm for the particular platform, which was quicksort for the CPU (DPH and NESL/CPU) and radix sort for the GPU (CUDA and Copperhead). The one exception is that we measured quicksort for the NESL/GPU platform, since it runs nearly 10% faster than radix sort owing to lower CPU/GPU communication overhead. The sorting benchmarks in Table 1 measure the sorting of a vector of 1,000,000 random integers (the actual vectors differ because of differences in the random-number generation algorithms).

Figure 11 provides a more detailed breakdown for each system across many vector lengths. Copperhead and CUDA both make very effective use of the GPU and scale well across increasing problem sizes. The NESL/CPU version of quicksort performs better than the NESL/GPU version because of the reduced cost of vector operations.

6.3.3 Black-Scholes

The Black-Scholes option pricing model is a closed-form method for computing the value of a European-style call or put option, based on the price of the stock, the original strike of the option, the risk-free interest rate, and the volatility of the underlying instrument [BS73]. This operation is numerically dense and trivially parallelizable. Reported numbers are for the pricing of 10,000,000 contracts.

The NESL/GPU version is able to perform much faster than the NESL/CPU version because of the very aggressive fusion of the dense numeric operations. The CUDA version is a hand-optimized version included in the NVIDIA SDK.

Figure 12 provides a more detailed breakdown for each system across many vector lengths.

6.3.4 Convex Hull

The convex-hull benchmark results shown in Table 1 are the result of determining the convex hull of 5,000,000 points in the plane. While this algorithm is trivially parallel, the parallel subtasks are not guaranteed to be of equal work sizes, providing an example of irregular parallelism. The NESL and DPH codes are based on the algorithm by Barber *et al.* [BDH96]. The convex-hull algorithm

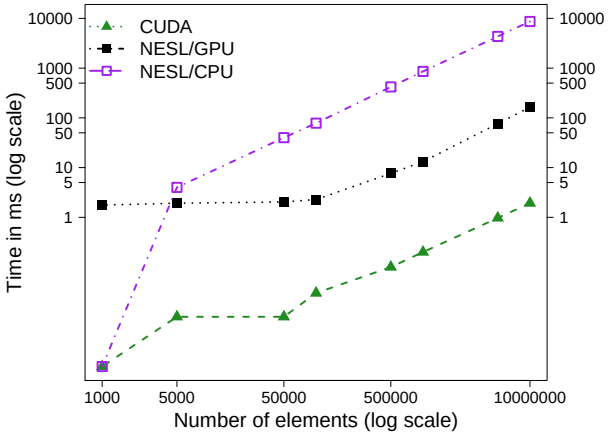

Figure 12. Black-Scholes execution times (ms) for a range of problem sizes. Smaller times are better. Axes are log scale.

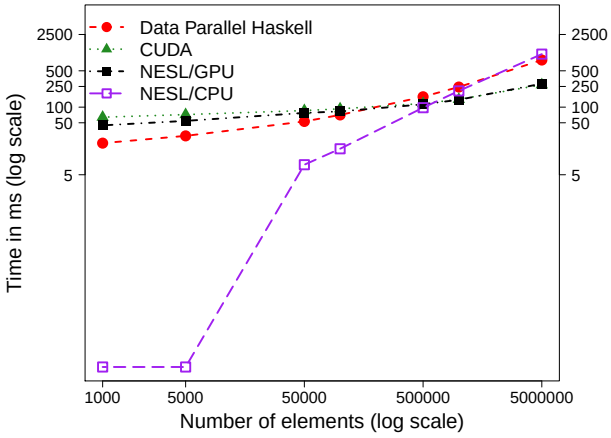

Figure 13. Convex Hull execution times (ms) for a range of problem sizes. Smaller times are better. Axes are log scale.

written in CUDA is from ongoing work at the National University of Singapore [GCN+12]. Their gHull algorithm was originally based on the quickhull algorithm, but has been optimized for better performance on a GPU. Their algorithm also works with points in 3D, but in our testing we constrained it to the 2D case for comparison with these other systems. It performs slightly better than the quickhull algorithm implemented in NESL and run on the GPU. This code was only made available to us in binary form, for execution on Windows, so we measured its performance on our benchmark machine under Windows 7 using the CUDA 4.1 drivers.

Figure 13 provides a more detailed breakdown for each system across many vector lengths.

6.3.5 Barnes-Hut (N-Body)

The Barnes-Hut benchmark [BH86] is a hierarchical N-body problem solver; we measure the 2D version that uses a quadtree to accelerate the computation. Each iteration has two phases. In the first phase, the quadtree is constructed from a sequence of mass points. The second phase then uses this tree to accelerate the computation of the gravitational force on the bodies in the system. All versions of this benchmark shown in Table 1 run one iteration over 75,000 random particles. Figure 14 provides a more detailed breakdown

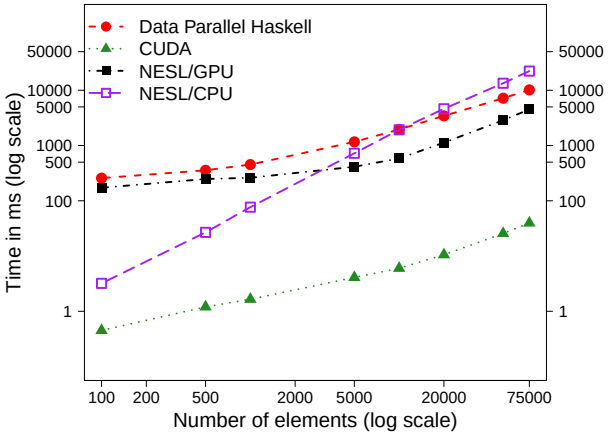

Figure 14. Barnes-Hut execution times (ms) for a range of problem sizes. Smaller times are better. Axes are log scale.

for each system across many vector lengths. The number of iterations is held constant at one.

The CUDA version of Barnes-Hut was implemented by Burtscher and Pingali [BP11]. We use version 2.2 of their source, which was the newest available as of February 2012.

All implementations exhibit roughly the same scalability. While the NESL/GPU implementation is the second fastest implementation, it is still significantly slower than the CUDA version. The NESL/GPU implementation's runtime is split roughly $1/3$ on memory operations and $2/3$ on kernel calls. These memory operations are related to allocations and conditional branches. When there is a conditional statement over a vector, first we perform the conditional check. Then, we sum the number of true elements from that conditional check and then transfer the integer (or integers, for a segmented vector) back from the GPU to the CPU to allocate a memory block of the correct size for each of the true and false elements. After that allocation, we copy the true elements into the true vector and the false ones into the false vector and then execute the clauses of the conditional on the appropriate vectors. These memory transfers back to the CPU and extra kernel calls could be avoided by moving some of the allocation code onto the GPU in a future implementation.

6.4 Effects of optimizations

In Section 5, we described a set of optimizations we perform on the code produced by the NESL compiler. Table 2 compares the execution time for the benchmarks across the optimizations on the NESL/GPU implementation, normalized to the mean execution time of the baseline. In nearly all cases, these optimizations result in improvements.

The one place where there is a decrease in performance is between the optimized and fused versions of quicksort. In the GPU version of quicksort, the vast majority of the remaining time spent in execution after optimization is in segmented sums and reductions that determine the number of elements in each of the less-than and greater-than partitions in order to shuffle them for the recursive calls. Because of this balance of work, the fusion of the element-wise operations that compare elements does not have a large enough effect to increase performance. While the mean execution time is slightly slower under fusion than the simple optimized version for this benchmark, the median of the fused

	NESL/GPU		
	Base	Opt	Fused
Dot Product	1.0	0.95	0.94
Sort	1.0	0.92	0.93
Black-Scholes	1.0	1.0	0.67
Convex Hull	1.0	0.95	0.91
Barnes-Hut	1.0	0.94	0.90

Table 2. Performance benefits from VCODE optimization. Execution times are normalized to the base (unoptimized) strategy, with smaller values being better.

values is lower and a more rigorous statistical analysis shows that the fused version is the same speed as the optimized one.[6]

One benchmark that particularly benefits from fusion and the creation of kernels is Black-Scholes option pricing. This benchmark is numerically dense and our optimizer aggressively reduces the number of kernel invocations, generating several kernels that each turn what would have been more than 10 separate GPU calls into a single call that both performs all of the operations and avoids intermediate writes to global memory.

7. Related Work

To the best of our knowledge, our work is the only example of an NDP language that is compiled for GPUs, so we divide the related work into languages that support (non-nested) data parallelism on GPUs and languages that use flattening to implement NDP on CPUs.

7.1 GPU Languages

The work on languages targeting GPUs is focused on regular data-parallel programs. These languages typically also provide library functions with efficient parallel implementations of `map` and `reduce`. While many of the available languages address some of the issues listed in Section 3.2, none of them address either the recursive call issue or any of the memory, data, and thread issues with respect to irregular data-parallel programs.

Barracuda [Lar11] and Single-Assignment C (SAC) for GPUs [GS06, GTS11], both provide techniques for compiling applicative array languages down to GPUs. Array languages have proven ideally suited for translation to the regular, flat parallelism available on the GPU. While these languages do not support parallelizing over irregular operations, SAC includes a fusion pass that finds operations whose results are based on the same index set and reduces those operations to a single pass generating multiple results. This fusion technique is not supported by our system, but it and other optimizations used in SAC may be useful for optimizing VCODE.

Nikola and Accelerate provide support for compiling operations on flat vectors in Haskell programs to run on GPUs [MM10, CKL+11]. Similarly Copperhead, discussed in more detail in Section 6.2, is a language targeting GPUs based on Python [CGK11]. These languages add `map`, `reduce`, and other high-level operations to significantly ease programming for GPUs and rely on the CPU to handle more complicated control-flow.

OptiX is an embedded domain-specific language and library that supports ray-oriented applications on GPUs [PBD+10]. These applications have a significant recursive component, which OptiX handles by CPS conversion and a trampoline.

[6] The Wilcoxon signed-rank test — appropriate for nonparametric, independent data points — provides 92% confidence that the difference between the two distributions is zero.

7.2 CPU Languages

Data Parallel Haskell (DPH) is the culmination of many years of research into expanding the flattening transformation to handle both more datatypes and higher-order functions [CKLP01, LCK06]. The NESL language does not support datatypes and is only first-order. DPH also supports partial vectorization, which allows portions of a program to remain unflattened [CLPK08]. In NESL, the entire program and all data are flattened. Finally, DPH implements a much wider range of fusion operations to remove the redundant allocations than the implementation presented in Section 5.2.

The Manticore project takes a different approach to nested data parallelism, implementing it without flattening and relying on efficient runtime mechanisms to handle load balancing issues [BFR+10].

8. Conclusion

We have shown that with careful implementation of the library primitives, the flattening transformation can be used on NDP programs to achieve good performance on GPUs. By focusing our performance tuning efforts on the VCODE implementation, we make it possible for a wide range of irregular parallel applications to get performance benefits from GPUs, without having to be hand ported and tuned. While performance does not match that of hand-tuned CUDA code, NESL programs are a factor of 10 shorter and do not require expertise in GPU programming. We hope that this work will be used as a better baseline for new implementations of irregular parallel applications than the usual sequential C programs. Better, of course, would be the integration of these classic compilation techniques for vector hardware into modern programming languages.

8.1 Future work

The most obvious limitation of our approach is that communication with the host CPU is required for allocation of memory. This requirement, as described Section 6.3.5, results in many additional communications between the CPU and GPU merely to provide an address. Moving the memory allocation responsibility into the GPU kernels would remove much of this communication cost. This issue is addressed by the compilation model used by Chatterjee in his work porting NESL to the MIMD Encore Multimax [Cha93]. His implementation included size analysis of programs and full custom C code generation. Some of his techniques may be applicable to improving GPU performance of NDP.

Acknowledgments

Ben Lippmeier provided help understanding the benchmark results and current status of Data Parallel Haskell. Bryan Catanzaro untangled the Copperhead dependency stack and provided insight into its current implementation and performance limitations.

We thank the NVIDIA Corporation for their generous donation of both hardware and financial support. This material is based upon work supported by the National Science Foundation under Grants CCF-0811389 and CCF-1010568, and upon work performed in part while John Reppy was serving at the National Science Foundation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

References

[BC90] Blelloch, G. and S. Chatterjee. VCODE: A data-parallel intermediate language. In *FOMPC3*, 1990, pp. 471–480.

[BC93] Blelloch, G. and S. Chatterjee. CVL: A C vector language, 1993.

[BCH+94] Blelloch, G. E., S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *JPDC*, **21**(1), 1994, pp. 4–14.

[BDH96] Barber, C. B., D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM TOMS*, **22**(4), 1996, pp. 469–483.

[BFR+10] Bergstrom, L., M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Lazy tree splitting. In *ICFP '10*. ACM, September 2010, pp. 93–104.

[BH86] Barnes, J. and P. Hut. A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, **324**, December 1986, pp. 446–449.

[Ble96] Blelloch, G. E. Programming parallel algorithms. *CACM*, **39**(3), March 1996, pp. 85–97.

[BP11] Burtscher, M. and K. Pingali. An efficient CUDA implementation of the tree-based Barnes Hut n-body algorithm. In *GPU Computing Gems Emerald Edition*, chapter 6, pp. 75–92. Elsevier Science Publishers, New York, NY, 2011.

[BS73] Black, F. and M. Scholes. The pricing of options and corporate liabilities. *JPE*, **81**(3), 1973, pp. 637–654.

[BS90] Blelloch, G. E. and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *JPDC*, **8**(2), 1990, pp. 119–134.

[CBS11] Cunningham, D., R. Bordawekar, and V. Saraswat. GPU programming in a high level language compiling X10 to CUDA. In *X10 '11*, San Jose, CA, May 2011. Available from `http://x10-lang.org/`.

[CGK11] Catanzaro, B., M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *PPoPP '11*, San Antonio, TX, February 2011. ACM, pp. 47–56.

[Cha93] Chatterjee, S. Compiling nested data-parallel programs for shared-memory multiprocessors. *ACM TOPLAS*, **15**(3), July 1993, pp. 400–462.

[CKL+11] Chakravarty, M. M., G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP '11*, Austin, January 2011. ACM, pp. 3–14.

[CKLP01] Chakravarty, M. M. T., G. Keller, R. Leshchinskiy, and W. Pfannenstiel. Nepal – nested data parallelism in Haskell. In *Euro-Par '01*, vol. 2150 of *LNCS*. Springer-Verlag, August 2001, pp. 524–534.

[CLPK08] Chakravarty, M. M. T., R. Leshchinskiy, S. Peyton Jones, and G. Keller. Partial vectorisation of Haskell programs. In *DAMP '08*. ACM, January 2008, pp. 2–16. Available from `http://clip.dia.fi.upm.es/Conferences/DAMP08/`.

[DR11] Dhanasekaran, B. and N. Rubin. A new method for GPU based irregular reductions and its application to k-means clustering. In *GPGPU-4*, Newport Beach, California, March 2011. ACM.

[Ert01] Ertl, M. A. Threaded code variations and optimizations. In *EuroForth 2001*, Schloss Dagstuhl, Germany, November 2001. pp. 49–55. Available from `http://www.complang.tuwien.ac.at/papers/`.

[GCN+12] Gao, M., T.-T. Cao, A. Nanjappa, T.-S. Tan, and Z. Huang. A GPU Algorithm for Convex Hull. *Technical Report TRA1/12*, National University of Singapore, School of Computing, January 2012.

[GHC] GHC. The Glasgow Haskell Compiler. Available from `http://www.haskell.org/ghc`.

[GS06] Grelck, C. and S.-B. Scholz. SAC — A Functional Array Language for Efficient Multi-threaded Execution. *IJPP*, **34**(4), August 2006, pp. 383–427.

[GTS11] Guo, J., J. Thiyagalingam, and S.-B. Scholz. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *DAMP '11*, Austin, January 2011. ACM, pp. 15–24.

[HB11] Hoberock, J. and N. Bell. Thrust: A productivity-oriented library for CUDA. In W. W. Hwu (ed.), *GPU Computing Gems, Jade Edition*, chapter 26, pp. 359–372. Morgan Kaufmann Publishers, October 2011.

[Kel99] Keller, G. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines*. Ph.D. dissertation, Technische Universität Berlin, Berlin, Germany, 1999.

[Khr11] Khronos OpenCL Working Group. OpenCL 1.2 Specification, November 2011. Available from `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`.

[Lar11] Larsen, B. Simple optimizations for an applicative array language for graphics processors. In *DAMP '11*, Austin, January 2011. ACM, pp. 25–34.

[LCK06] Leshchinskiy, R., M. M. T. Chakravarty, and G. Keller. Higher order flattening. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra (eds.), *ICCS '06*, number 3992 in LNCS. Springer-Verlag, May 2006, pp. 920–928.

[Les05] Leshchinskiy, R. *Higher-Order Nested Data Parallelism: Semantics and Implementation*. Ph.D. dissertation, Technische Universität Berlin, Berlin, Germany, 2005.

[MGG12] Merrill, D., M. Garland, and A. Grimshaw. Scalable GPU graph traversal. In *PPoPP '12*, New Orleans, LA, February 2012. ACM, pp. 117–128.

[MLBP12] Mendez-Lojo, M., M. Burtscher, and K. Pingali. A GPU implementation of inclusion-based points-to analysis. In *PPoPP '12*, New Orleans, LA, February 2012. ACM, pp. 107–116.

[MM10] Mainland, G. and G. Morrisett. Nikola: Embedding compiled GPU functions in Haskell. In *HASKELL '10*, Baltimore, MD, September 2010. ACM, pp. 67–78.

[NVI11a] NVIDIA. NVIDIA CUDA C Best Practices Guide, 2011.

[NVI11b] NVIDIA. NVIDIA CUDA C Programming Guide, 2011. Available from `http://developer.nvidia.com/category/zone/cuda-zone`.

[PBD+10] Parker, S. G., J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. OptiX: a general purpose ray tracing engine. *ACM TOG*, **29**, July 2010.

[PPW95] Palmer, D. W., J. F. Prins, and S. Westfold. Work-efficient nested data-parallelism. In *FoMPP5*. IEEE Computer Society Press, 1995, pp. 186–193.

[Pro95] Proebsting, T. A. Optimizing an ANSI C interpreter with superoperators. In *POPL '95*, San Francisco, January 1995. ACM, pp. 322–332.

[SHZO07] Sengupta, S., M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *GH '07*, San Diego, CA, August 2007. Eurographics Association, pp. 97–106.

[YHL+09] Yang, K., B. He, Q. Luo, P. V. Sander, and J. Shi. Stack-based parallel recursion on graphics processors. In *PPoPP '09*, Raleigh, NC, February 2009. ACM, pp. 299–300.