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. Porting algorithms that do not easily fit this model requires laborious manual program conversion to run programs with irregular control-flow or data layout. Language support for programming GPUs is rather limited. The low-level GPU languages, such as CUDA and OpenCL, appear general-purpose, but must be used in restricted styles to get adequate performance. Some effort has been made to support computation on GPUs from high-level languages, such as C++, X10, and Haskell, but these mechanisms are limited to computations over flat vectors.

NESL is a first-order functional language that was designed by Guy Blelloch to allow programmers to write irregular-parallel programs — such as parallel divide-and-conquer algorithms — for wide-vector parallel computers. NESL supports *nested data parallelism* (NDP), where the elements of a data-parallel computation can themselves be data-parallel computations, and it uses a novel compilation technique, called *flattening*, to map nested computations onto a SIMD execution model. This work demonstrated that NDP can be effectively used for a wide range of irregular-parallel computations.

Wide-vector machines pose many of the same programming challenges as GPUs, so it is worth exploring the question of whether an NDP language, like NESL, can make an effective tool for programming GPUs. This paper presents the results of such an exploration. We describe our port of the NESL implementation to work on GPUs and present empirical evidence that NDP on GPUs significantly outperforms CPU-based implementations and matches or beats newer GPU languages that support only flat parallelism. We also compare our performance with hand-tuned CUDA programs. While our performance does not match that of hand-tuned codes, 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—Concurrent, distributed, and parallel languages; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Languages, Performance

Keywords GPU, GPGPU, NESL, nested data parallelism

[Copyright notice will appear here once 'preprint' option is removed.]

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. Codes compiled for these cores run best when the same program runs the same instructions across cores and when applied to large amounts of data. This model works well for a wide variety of highly 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 require care to use effectively. 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 code such applications by hand. The literature is rife with papers on the implementation of irregular-parallel algorithms on 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 also excelled at high performance on 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 and the parallel summation reduction:

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

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.

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 results in a vectorized stack-machine code, called VCODE. We use a series 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 and demonstrates 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 the quicksort example. Then, we provide an overview of the graphics processing unit (GPU) hardware and 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.

Source code for our complete implementation and all the benchmarks described in this paper is available at: `http://smlnj-gforge.cs.uchicago.edu/projects/ndp2gpu/`.

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, which it transforms into flat arrays for execution on vector hardware. A standard example of NDP computation in NESL is the quicksort algorithm, which is given in Figure 1. The pair of recursive calls to the **quicksort** function on an unknown division of the data leads to irregular parallel execution. That is, up front the compiler cannot know how the data will be partitioned, so it cannot statically allocate and balance the workload.

```
function quicksort(a) =
    if (#a < 2) then a
    else let
        pivot   = a[#a/2];
        lesser  = {e in a | e < pivot};
        equal   = {e in a | e == pivot};
        greater = {e in a | e > pivot};
        result  = {quicksort(v)
                   : v in [lesser,greater]};
      in result[0] ++ equal ++ result[1];
```

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
        pivots = {a[#a/2] : a in as};
        lessers  = {{e in es | e < pivot}
            : pivot in pivots; es in as};
        equals   = {{e in es | e == pivot}
            : pivot in pivots; es in as};
        greaters = {{e in es | e > pivot}
            : pivot in pivots; es in as};
        results  = quicksort' (
            flatten ([lessers, greaters]));
      in join(results, equals);
```

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

The NESL compiler supports such irregular NDP computations by transforming the computation into flattened version. It produces an intermediate representation, called VCODE, for an vectorized stack-based virtual machine [BC90]. The NESL implementation uses 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 by 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.

The flattening transformation changes the **quicksort** function from operating over a single vector at a time into a new version that operates over a nested vector; *i.e.* a 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 date from all of the different vectors. The other part is one or more segment descriptors, which are vectors that contain the index and length information required to reconstruct the nested vectors from the flat data vector.

After the parameters to the **quicksort** function are changed, all of the code within the body of the function must be changed. This change requires every operation that previously operated on a scalar value to now operate on a vector. Further, any operation that took a vector must be lifted to work on a nested vector.

The scalar **pivot**, which was a scalar value holding the single pivot element from the input vector **a** in the original program

```
FUNC QUICKSORT_13
  COPY 1 0
  CALL PRIM-DIST_37
  COPY 2 1
  COPY 1 2
  CALL VEC-LEN_13
  CONST INT 2
  COPY 1 2
  CALL PRIM-DIST_6
  COPY 1 2
  POP 1 0
  < INT
...
```

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.

becomes the vector **`pivots`**, holding all of the pivots from each of the vectors inside of the nested vector **`as`**. The vectors that previously held the **`lesser`**, **`equal`**, **`greater`**, and **`result`** 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 represented by **`as`** have length less than 2.

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

The code that reassembles the vector of vectors of equal values with the results from the recursive calls has been is represented by the **`join`** operator. In its implementation, this function uses the segment descriptors associated with each of the original partitioning operators to assemble a vector describing the required permutations of the elements from the **`results`** and **`equals`** nested vectors to place the vectors in their correct order.

2.2 NESL runtime

The output of the NESL compiler is the flattened program transformed into the VCODE language. The VCODE program is executed by an interpreter that calls into a C-based library of vector operations. That language is then run by an interpreter, making calls into a C-based library that implements the vector operations. Because most of the computational load in a VCODE program is in the vector operations, a tuned implementation of the vector operations is required for good performance.

2.2.1 VCODE

Figure 3 contains a very small portion of the actual VCODE generated by the NESL compiler from the original quicksort algorithm in Figure 1. VCODE is a stack-based language that is intended to be run on a host computer with vector operations performed on a vector machine.

2.2.2 CVL

The C Vector Library (CVL) is a set of C library functions callable from a host machine that implement high-level vector operations [BC93]. This library has implementations for many of the high-performance parallel computers of the 1990's, as well as a basic C implementation for sequential execution on traditional computers. This library is called by VCODE for all of the high-level vector operations and is the primary hook between the host machine and hardware on which the program is executing. 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 section, we describe the aspects of the GPU hardware and programming models that must be addressed in high-performance implementations of parallel programs on GPUs. For purposes of 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 group 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 take the other branch. An SMP can efficiently switch between different groups of threads, 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. For the results of this paper, we use an NVIDIA Tesla C2050, which has 14 SMPs, each with 32 cores for a total of 448 cores.

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. All the threads in a block are mapped to the same SMP. These blocks are then further subdivided into warps, which are the basic unit of parallel execution. 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).

3.1 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. In this section, we discuss the key challenges in the rest of this section. 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 data transfers 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 take to reduce the occurrence of divergent conditionals.

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 a 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 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 followed by careful performance profiling. Our implementation passes all of the NESL regression tests. We leave a discussion of the VCODE optimizer and supporting interpreter modifications to Section 5.

4.1 Implementation basics

In the CVL model, all allocation is done at startup, based on a cost estimate of the maximum live memory requirements of the VCODE program. This number of elements is requested during startup from our library implementation. 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 can be represented with a flat data vector $[4, 5, 6, 7, 8, 9]$

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

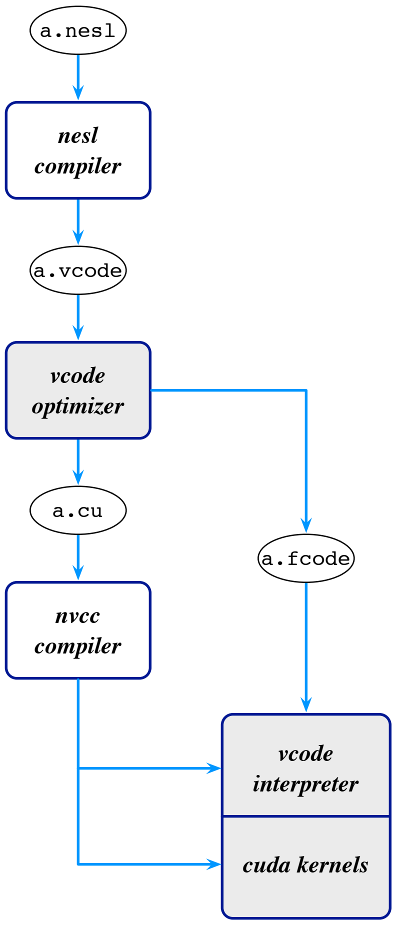

Figure 4. The NESL/GPU toolchain. The shaded portions represent our contributions.

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 make 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.

It turns out that 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,in order 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 direct experimentation with several different alternatives. The worst — lengths only — resulted in individual kernel calls taking multiple *seconds*, even after optimization. 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 input data that is a small constant factor less than the total memory available on the GPU. The remaining subsections discuss implementation details and trade-offs in these kernel calls.

4.3 Element-wise operations

The element-wise operations perform operations that operate regularly over their inputs, independent of the nesting structure. Thus, these operations are the only ones that are unaffected by the choice of segment descriptor representation. These operations 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 compute resources, whereas with too few warps per block, the SMP will not have any code to execute as all of the warps will be waiting on memory operations. This problem is the *memory access* challenge discussed in Section 3.1.

Based on our testing, blocks of 256 threads (8 warps) improves performance relative to either lower or higher numbers of threads by a factor of up to 8. Furthermore, we found that increasing beyond 256 threads per block causes low occupancy rates.[2] Our experiments 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 our performance results because 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 the scan operation retains all 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. For this version of the implementation, we perform a segmented inclusive scan and then perform 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 the initial version of this kernel function, 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 `+-scan` 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 an initialization value and incremental stride.

These operators rely on the segment descriptor information that allows individual threads to avoid each running over 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 to provide the per-segment offsets, sometimes for both source and target segment descriptors if they differ and are required. The operations requiring this 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.

But, 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 Facilities

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

[2] 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.

support a stride, allowing the value at an index to be a function of an initialization value and a stride factor. Again, this was 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, since 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. In the single segment case, all elements share the same stride and index, so they 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 in the library is the creation of the 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.[3] Creation of the segment descriptor is provided a vector with the lengths of each segment. So, the vector `[1,3,3]` describes an underlying vector of length 7 with 3 segments.

We are provided space for that 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. If there are 7 elements in the array, there would now be a vector of 7 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]`.

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 potentially large number of kernel threads to touch the same piece of memory will immediately generate performance so poor 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

[3] Vectors are immutable in the NESL implementation, so segment descriptors are frequently shared and reused in the generated VCODE.

```
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

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 six 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. The inliner does not inline recursive functions and uses a size metric to limit code growth.[4] 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:[5]

```
function MULADD1_7 (p0, p1, p2, p3, p5)
    let t033 = (* INT @ p1 p3)
    let t034 = (+ INT @ t033 p5)
    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
```

[4] Our size metric is the number of computational instructions in the function.

[5] The reader might wonder what happened to parameter `p4`: it was an unused parameter that was eliminated by the optimizer.

```
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. 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 = (+ INT @ (* INT @ 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.

```
fused OP0 ($0 : INT, $1 : INT, $2 : INT) =
    (+ INT @ (* 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. In this example, run on three vectors of 1,000,000 integers, we reduce the number of kernel calls by 4. Further, we reduce the number of 32-bit loads from 19,000,417 to 12,300,417 for a savings of 6,700,000. The 32-bit stores were reduced from 11,000,000 to 10,000,000, saving 1,000,000.

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
```

Then, we generate custom CUDA kernels for each fused superoperator. The corresponding kernel for the fused operation in this example is shown below.

```
__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];
  }
}
```

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 this example. When the interpreter hits a fused kernel call it runs code that to perform the proper allocations, argument checks, and invoke the kernel through the host C function. While our extensions do not support arbitrary C functions — the interpeter must be able to determine the size of the result based on the size of its inputs — they do support a wide set of pure element-wise operations and the common vector initialization arguments.

6. Evaluation

To evaluate the effectiveness of our approach to implementing NESL on GPUs, we compare the performance of our system to a number of other implementations, both CPU based and GPU based. 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 is system with an Intel i7-950 quad-core processor running at 3.06GHz. Hyper-threading is enabled, which makes eight cores available to parallel languages. Our GPU is an NVIDIA Tesla C2050 with 448 cores and 3Gb of global memory. Our main software platform is the Linux x86-64 kernel version 3.0.0-15 with the CUDA 4.1 drivers. For one test we used Microsoft Windows 7 on this hardware (also with the CUDA 4.1 drivers).

We report all benchmark measurement numbers as the wall-clock execution time of the core computation, excluding initialization and result verification times, since those times differ widely between the different systems being compared. Each benchmark was run 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 })
```

6.2.1 NESL/CPU

In our evaluation, we measured the optimized, but not fused, VCODE executed using the the CPU implementation of the CVL library and vector primitives that is included with NESL. While not fully hand-optimized with vector operations, these primitives were developed and optimized over many years. Because the flattening transformation provides such straightforward vectors to perform operations over, the C compiler is able to produce very efficient code that provides an extremely competitive baseline implementation.

This implementation is sequential and uses only a single processor, but it does make use of the SSE vector hardware in that processor, as the significant loop unrolling, extensive macros to inline all operations, and aggressive compiler flags used enable the C compiler's autovectorization optimizations.

6.2.2 CUDA

To provide a reality check, we measured hand-coded implementations of the various benchmarks. These implementations represent significant programmer effort and provide a limit on the performance that one might reasonably hope to active.

Dot product in CUDA Figure 6 gives CUDA kernel code for the dot product. This program performs basic blocking and use of fast shared memory on the GPU, but is slower than the optimized version available in CUBLAS.[6] This example code was presented at the 2010 NVIDIA GPU Technical Conference [NVI10], and our presentation omits initialization, data transfer, and error checking.

Some of the issues that a CUDA programmer needs to worry about include:

- The number of threads per chunk of work and number of chunks of total work will determine the *occupancy* — the percentage of the GPU that is utilized.
- Threads within a block may be synchronized using the **__syncthreads** function, which ensures that all of the writes to the **temp** variable will be seen by all threads in the same thread block. Threads not running together may not communicate across shared memory.
- The **atomicAdd** function allows values to be incrementally and safely added into global memory from independent sets of threads. Again, the number of these writes must be balanced against the total number of blocks, and in a more advanced implementation there might be multiple kernel invocations, each one reducing the array down to a smaller and smaller set of values to remove the need for global atomic operations.

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. Later,

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

__global__ void dot( int *a, int *b, int *c )
{
  __shared__ int temp[THREADS_PER_BLOCK];
  int index = threadIdx.x + blockIdx.x
      * blockDim.x;
  temp[threadIdx.x] = a[index] * b[index];

  __syncthreads();

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

int main(void)
{
  dot<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>
    (dev_a, dev_b, dev_c);
}
```

Figure 6. Basic CUDA implementation of dot product

```
from copperhead import *

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

Figure 7. Copperhead implementation of dot product

using a special **places** keyword, those annotated definitions can be executed either on the GPU or CPU.

We are using the Coperhead 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.

Dot product in Copperhead The dot product example is very straightforward to write in Copperhead and appears in Figure 7. The definition to run on the card, **dot_product**, is annotated with the `@cu` keyword. When the Copperhead function is invoked, data is transferred as needed and coerced between Python and CUDA types.

6.2.4 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. Code in source files 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.[7] This system is included because it also implements a version of the flattening transformation.

The DPH sources as of February, 2012 were used. They were compiled with the released GHC version 7.4.1 compiler.

[6] We measured the faster, optimized CUBLAS version in our benchmark comparisons.

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

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

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

Figure 8. Data Parallel Haskell implementation of dot product

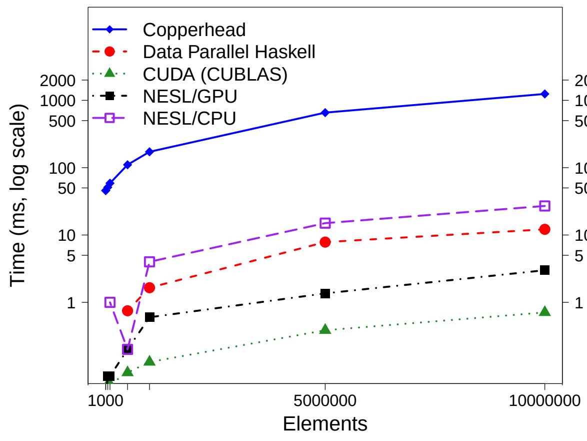

Figure 9. Dot product execution times (ms) for a range of problem sizes. Smaller times are better. Times are in log scale.

Dot product in DPH Currently, DPH programs require explicit type declarations, as shown in Figure 8. But, at its core, the last line is very similar to those seen in both the Copperhead and NESL versions of this small program.

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 — despite having the smallest programs — is the next fastest implementation for all but the sorting benchmark, which has unbalanced computations that fail to take full advantage of the GPU hardware.

All of the NESL numbers reported in this section have been 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.

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, both vectors contain 10,000,000 32-bit floating point numbers on all platforms. Figure 9 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 element sizes less than 5,000,000 on the NESL/GPU version, the program 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.

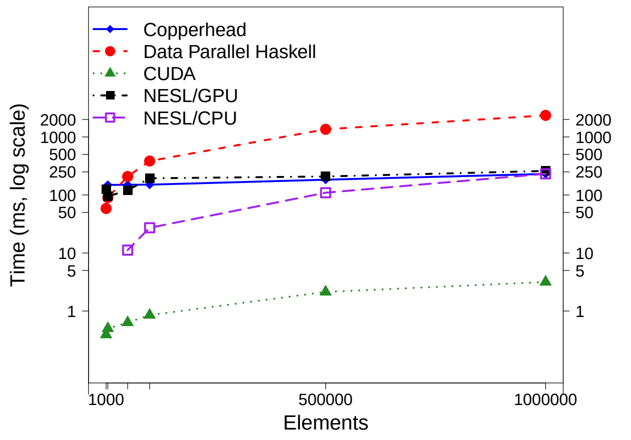

Figure 10. Sorting execution times (ms) for a range of problem sizes. Smaller times are better. Times are in log scale.

6.3.2 Sorting

The sorting benchmarks in Table 1 are all performed on a vector of 1,000,000 random integers. Different sorting algorithms have better performance on various platforms. For DPH, NESL/GPU, and NESL/CPU, quicksort is the fastest and most scalable sorting implementation available, whereas for CUDA and Copperhead, radix sort is the fastest.

The NESL implementations run faster with quicksort than radix sort because of the reduced number of vector operations (which turn into kernel calls on NESL/GPU). The NESL/CPU speed difference between radix sort and quicksort is very small since these operations are just a CPU function call on that platform.

Figure 10 provides a more detailed breakdown for each system across many vector lengths. Copperhead, like the CUDA version of sorting, makes much more effective use of the GPU and scales significantly better than in the dot product benchmark. The NESL/CPU version of quicksort performs better than the NESL/GPU version because of the reduced vector operation overhead and the fact that the unbalanced computations result in a workload that does not take full advantage of the available parallelism on the GPU.

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.

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

	Problem Size	Lines of code				Execution time (ms)				
	(# elements)	NESL	CUDA	Copper	DPH	NESL/GPU	CUDA	Copper	DPH	NESL/CPU
Dot Product	10,000,000	**8**	80	31	39	3.0	<**1**	1,240	12	27
Sort	1,000,000	**12**	136	46	52	259	**3.1**	230	2,360	230
Black-Scholes	10,000,000	**37**	337	*N/A*	*N/A*	164	**1.9**	*N/A*	*N/A*	8,666
Convex Hull	5,000,000	**25**	*unknown*	*N/A*	72	283	**269**	*N/A*	807	1,000
Barnes-Hut	75,000	**225**	1930	*N/A*	414	671	**40**	*N/A*	10,200	2,200

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.

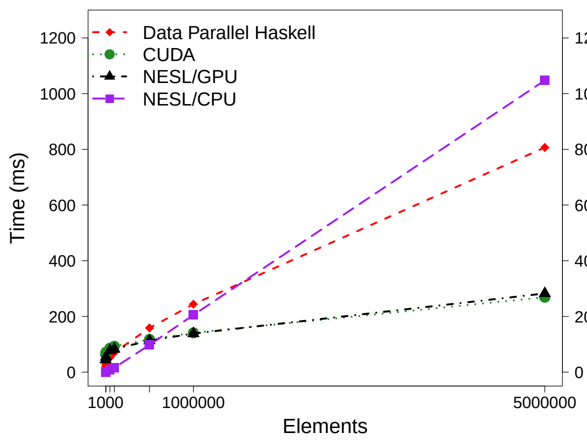

Figure 11. Convex Hull execution times (ms) for a range of problem sizes. Smaller times are better.

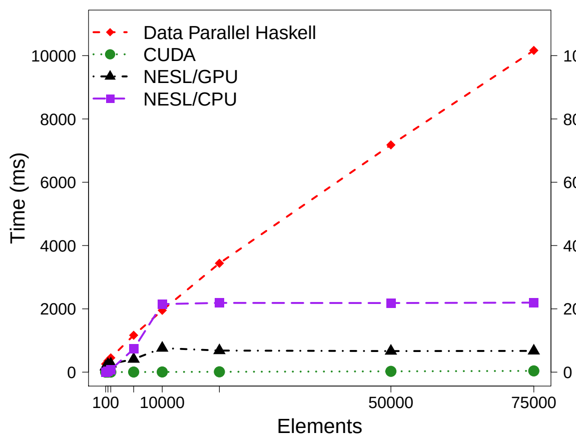

Figure 12. Barnes-Hut execution times (ms) for a range of problem sizes. Smaller times are better.

not guaranteed to be of equal work sizes, providing an example of irregular nested data-parallelism. The NESL and DPH codes are based on the algorithm by Barber *et al.* [BDH96].

The convex hull algorithm 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. Furthermore, their algorithm also works with points in 3D. 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.

There is no implementation of the Quickhull algorithm in Copperhead.

Figure 11 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 classic N-body problem solver. The version we measure is the 2D quadtree version. Each iteration has two phases. In the first phase, a 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. There is no current implementation of the Barnes-Hut algorithm for Copperhead.

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.

Figure 12 provides a more detailed breakdown for each system across many vector lengths. The number of iterations is held constant at one.

The NESL implementations both scale much better than the DPH version, but still have significant overheads relative to the CUDA version. In the NESL/GPU implementation, the 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 in order 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 could be avoided by moving some of the allocation code onto the GPU.

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

	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	0.68
Convex Hull	1.0	0.95	0.91
Barnes-Hut	1.0	0.85	0.83

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

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 values is lower and a more rigorous statistical analysis shows that the fused version is the same speed as the optimized one.[8]

One benchmark that particularly benefits from fusion and the creation of kernels is Black-Scholes option pricing. This benchmark is very 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

7.1 GPU Languages

The work on languages targeting GPUs is similarly focused on regular parallelism, paired with library functions intended to save programmers from copying the implementations of **`map`** and **`reduce`** from the CUDA toolkit in each program they write. While many of the available languages address some of the issues listed in Section 3.1, 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 for GPUs [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. But, these languages do not support parallelizing over irregular operations, such as divide-and-conquer algorithms.

Nikola and Accelerate provide support for compiling operations on flat vectors in Haskell programs to run on GPUs [MM10, CKL$^+$11]. Similarly, Copperhead, mentined in more detail in Section 6.2, is a language targeting GPUs based on Python [CGK11]. Both of these languages add **`map`**, **`reduce`**, and other high-level operations to significantly ease programming for GPUs. But, neither of them has any method for dealing with recursion, or irregular, nested vectors.

OptiX is a domain-specific language and library that implements a runtime for executing ray-oriented applications, which traditionally have a significant recursive component [PBD$^+$10]. These recursive calls in the program are transformed by the OptiX compiler into a save of any live variables at the time of the call into a custom, per-element stack and then a direct jump to the labeled entry point corresponding to the original function.

[8] 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 reserach into expanding the flattening transformation to handle both more datatypes and higher-order functions [CKLP01, CLPK08, LCK06]. This language uses *partial vectorization* to convert vectors containing functions or vectors whose contents are the partial application of a function into a flat data-parallel program. 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 communcations between the CPU and GPU merely to provide an address. Moving the memory allocation responsibility into the GPU kernels would remove much of this communcation 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 *POPL '11*, San Antonio, 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, 2011. ACM.

[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.

[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`.

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

[HB11] Hoberock, J. and N. Bell. Thrust: A Productivity-Oriented Library for CUDA. In W. W. Hwu (ed.), *GPU Computing Gems, Jade Edition*. 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, Texas, USA, 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, Louisiana, USA, 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, Louisiana, USA, 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.

[NVI10] NVIDIA. NVIDIA GPU Technical Conference, 2010.

[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, California, 2007. Eurographics Association.

[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, 2009. ACM.