NOVA: A Functional Language for Data Parallelism

Alexander Collins
University of Edinburgh
a.collins@ed.ac.uk

Dominik Grewe
University of Edinburgh
dominik.grewe@ed.ac.uk

Vinod Grover
NVIDIA Corporation
vgrover@nvidia.com

Sean Lee
NVIDIA Corporation
selee@nvidia.com

Adriana Susnea
NVIDIA Corporation
adriana@alumni.princeton.edu

Abstract

Functional languages provide a solid foundation on which complex optimization passes can be designed to exploit parallelism available in the underlying system. Their mathematical foundations enable high-level optimizations that would be impossible in traditional imperative languages. This makes them uniquely suited for generation of efficient target code for parallel systems, such as multiple Central Processing Units (CPUs) or highly data-parallel Graphics Processing Units (GPUs). Such systems are becoming the mainstream for scientific and commodity desktop computing.

Writing performance portable code for such systems using low-level languages requires significant effort from a human expert. This paper presents NOVA, a functional language and compiler for multi-core CPUs and GPUs. The NOVA language is a polymorphic, statically-typed functional language with a suite of higher-order functions which are used to express parallelism. These include **map**, **reduce** and **scan**. The NOVA compiler is a light-weight, yet powerful, optimizing compiler. It generates code for a variety of target platforms that achieve performance comparable to competing languages and tools, including hand-optimized code. The NOVA compiler is stand-alone and can be easily used as a target for higher-level or domain specific languages or embedded in other applications.

We evaluate NOVA against two competing approaches: the Thrust library and hand-written CUDA C. NOVA achieves comparable performance to these approaches across a range of benchmarks. NOVA-generated code also scales linearly with the number of processor cores across all compute-bound benchmarks.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming

General Terms Languages, Performance

Keywords Functional programming, Compilation, Code generation, Array-oriented programming, CUDA, Multi-core CPU

ARRAY'14 June 11 2014, Edinburgh, United Kingdom
 ACM 978-1-4503-2937-8/14/06
http://dx.doi.org/10.1145/2627373.2627375. . . $15.00

1. Introduction

Although a number of programming systems have emerged to make parallel programming more accessible on multi-core CPUs and programmable GPUs for the last several years, many of them — including *Threading Building Block* (TBB) [3], *CUDA* [4], and *Open Compute Language* (OpenCL) [2] — are targeted towards C/C++ programmers who are familiar with the intricacies of the underlying parallel hardware. They provide low-level control of the hardware, with which C/C++ programmers can fine-tune their applications to enhance the performance significantly, by sacrificing high-level abstraction. Whilst the level of abstraction they provide gives much flexibility to the C/C++ programmers, it also has been an obstacle for others to adopt these low-level programming systems.

To broaden the application of parallel programming, there have been various attempts to provide programming systems with high-level abstraction and performance gains comparable to what the aforementioned low-level systems offer [1, 10–12, 15, 20]. This paper presents NOVA, a new functional language for parallel programming that shares the same goal as these systems. NOVA allows the user to express parallelism using high-level parallel primitives including **map**, **reduce** and **scan**. The NOVA compiler generates multi-threaded C code for CPUs or CUDA C code for NVIDIA GPUs. The generated code achieves performance comparable to similar approaches and hand-written code.

While NOVA can be used on its own, it can also be used as an *intermediate language* (IL) for other languages such as *domain specific languages* (DSLs). The NOVA compiler can be extended with additional front-ends for DSLs. This allows them to exploit the optimizations and multiple back-ends present in the compiler. Moreover, the compiler can be extended with additional back-ends. There are currently three back-ends (sequential C, parallel C and CUDA C). By allowing extension of both the front and back-ends, NOVA can be integrated within existing tools.

The main contributions of this paper are:

- *The NOVA language:* a high-level functional programming language for parallel computation. It includes support for nested parallelism, recursion and type polymorphism.
- *The NOVA compiler:* which produces efficient, scalable and performance portable target code from the NOVA language. It achieves performance comparable to existing state-of-the-art low-level tools and hand written parallel code.

2. Motivation

Consider implementing an algorithm that computes the tight bounding box of a set of two-dimensional points. Implementing this by hand for both multi-core CPU and GPU would require two distinct versions of the program. For example, one would be implemented in C++ and the other in CUDA C [4]. An alternative would be to use Thrust [6], which provides parallel abstractions for both CPU and GPU using CUDA C. The bounding box example from the Thrust example distribution[1] achieves this with 86 lines of code.

However, we can implement this algorithm far more succinctly using a functional programming language. Using NOVA, we can implement this in 32 lines of code, compared to the 86 lines required by Thrust. NOVA completely removes the need for low-level boiler plate code, such as device management and host to device memory transfers which are required when using CUDA C. NOVA also removes the need for platform specific optimizations. These are often required to achieve best performance in hand-written C++ and CUDA C, and Thrust.

On a NVIDIA GeForce GTX480[2], the NOVA version of bounding box achieves a speedup in execution time of 1.07× over the Thrust version.

This performance improvement is due to the high-level optimizations that are enabled by expressing the algorithm in a functional language form. For example, the **map** and **reduce** operations can be merged into a composite **mapReduce** operation. This overlaps the execution of the **map** and **reduce** operations, increasing the utilization of the GPU and therefore improving performance.

The code is also far more maintainable. It does not require the programmer to have an in-depth knowledge of the intricacies of the underlying hardware. They simply choose the parallel primitives that best suit their algorithm, and the NOVA compiler decides how best to implement them on the given hardware.

3. The NOVA Language

NOVA is a statically-typed functional language, intended to be used as an intermediate language, or target language, for domain specific front-ends. Figure 1 summarises the structure of the compiler; with multiple front-ends, and back-ends (for code generation). The design of the language is centered around vectors and data parallel operations, rather than registers and instructions. It is also designed to facilitate high-level language transformations such as deforestation of vector operations and closure conversion [14]. A representative set of the parallel operations provided by NOVA are listed in Table 1.

The rest of this section is structured as follows. Section 3.1 describes the salient features of NOVA, Section 3.2 details the polymorphic type system and Section 3.3 shows two simple example programs, highlighting the salient features of the language.

3.1 Language Features

NOVA includes the usual features of a functional language such as, lambda expressions and let-expressions.

NOVA allows existing code, written in the target language (such as C) to be used within a NOVA program. These are defined in a **foreign** section at the start of the program. For example the following makes the foreign function f available within a NOVA program:

```
(foreign
  (f : (int → int))
)
```

[1] `http://thrust.googlecode.com/files/examples-1.6.zip`

[2] using CUDA 4.1

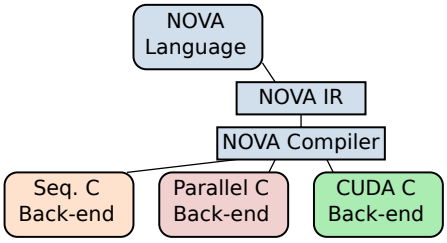

Figure 1. Summary of the structure of the NOVA compiler infrastructure, with a common NOVA-Intermediate Representation, and multiple front-ends and back-ends.

Operation	Description
map $f\ X_1 \ldots X_n$	Applies function f to every tuple of elements at the same index in vectors $X_1 \ldots X_n$
reduce $f\ i\ X$	Performs a reduction on vector X using function f and initial value i
scan $f\ i\ X$	Performs a prefix-scan on vector X using function f and initial value i
permute $I\ X$	Generates an output vector Y such that $Y[I[i]] = X[i]$
gather $I\ X$	Generates an output vector Y such that $Y[i] = X[I[i]]$
slice $b\ s\ e\ X$	Generates an output vector Y such that $Y[i] = X[b + i.s]$ with a length of $\lceil (e - b)/s \rceil$
filter $f\ X$	Given a filter function f and a vector X, returns a vectors containing only those elements from X for which f evaluates to *true*

Table 1. A representative sample of NOVAs built-in parallel operations

The generated code can then be linked against a library containing the implementation of the foreign function.

NOVA supports *sum types*, a form of user-defined algebraic (or compositional) data type. This allows types to be combined to create more complex types. For example, a *Maybe* data type, that either holds a value of type **int**, or *nil* can be defined as follows:

```
(types
  (Maybe :
    (+ (Some : int)
       (None : unit))))
```

Sum types also support recursion. This allows complex structures such as lists or trees to be defined. For example, a list of integers can be defined as follows:

```
(types
  (IntList :
    (+ (Nil : unit)
       (Cons : (int, IntList)))))
```

However, use of NOVA's built-in vector data types and parallel operators is recommended over the use of, for example, a user-defined list type, as this will achieve best performance.

3.1.1 Recursion

NOVA supports recursion through the use of μ-expressions, which are similar to the fixed-point combinator. A μ-expression has the form **mu** $(x : t)\ e$. This binds identifier x to expression **mu** $(x : t)\ e$ in expression e. In other words, e can refer to itself using the identifier x. On top of this very general definition, we add the constraint that e must be a λ-expression that is statically determinable. This allows us to perform closure conversion on μ-expressions.

For example, consider the following recursive μ-expression:

```
(mu (fib : int → int)
  (lambda (n : int)
    (if (< n 2) then 1
    else (fib (- n 1) (- n 2)))))
```

The enclosing μ-expression defines the identifier *fib*. This identifier is bound to the entire μ-expression. When *fib* is used within the body of the μ-expression, it evaluates to this entire μ-expression.

Many functional languages use a recursive let-expression (sometimes called `letrec`) to implement recursion. Our μ-expressions are equivalent:

$$\texttt{mu } (x{:}\tau)\ e \equiv \texttt{letrec } (x\ e) \texttt{ in } x$$

NOVA allows type generalization and specialization, in a similar manner to System F [13, 18].

We impose a few restrictions on the use of generalized, **forall** types and type applications. Firstly, general types can only be constructed within the **types** section at the start of a NOVA program.

Secondly, after type checking a program, all types must be specialized, or turned into concrete types. This is because the programming environments that NOVA targets (including CUDA C and parallel C) require all types to be concrete. If an expression is discovered whose type is not specialized to a concrete type (such as partial application of a parallel operator), the compiler complains that it could not statically determine the concrete type of the expression.

3.2 Type Inference

The NOVA language performs Hindley-Milner type inference [17], with some extensions to support polymorphism in the arity of some of the built-in parallel operators.

For example, the **map** parallel operator can take a variable number of input vectors, which are used to compute a single output vector. Performing type inference over this variable number of input parameters is not possible with Hindley-Milner type inference, but it is restricted to the built-in operations. Therefore the compiler includes hand-coded rules to perform type inference for expressions involving these operators.

3.3 Example Programs

This section presents some simple example programs in the NOVA language, demonstrating the salient features of the language.

Computing the Length of a List This example uses a tail-recursive algorithm to count the number of items in a polymorphic list.

```
((input : (List int)))
(types
  (List 'a : (+ (Nil : unit)
                (Cons : ('a, (List 'a))))))
(let
  (len (mu (len : int → (List 'a) → int)
    (lambda (l : int) (xs : (List 'a)) : int
      (case xs
        (Nil  x  l)
        (Cons xs (len (+ l 1) (xs.1)))))))
  in (len 0 input))
```

Reduction on a Polymorphic Binary Tree This example demonstrates tree reduction on a polymorphic binary tree, using a polymorphic reduction function with type $'a \to 'a \to 'b$.

```
((input : (Tree int)))
(types
  (Tree 'a : (+ (Leaf : 'a)
                (Node : ('a, (Tree 'a), (Tree 'a))))))
(let
  (red
    (mu (red : ('a→'a→'a) → (Tree 'a) → 'a)
      (lambda (f:'a→'a→'a) (t:(Tree 'a)) : 'a
        (case t
          (Leaf l (l))
          (Node n
            (let
              (left  (red f (n.1)))
              (right (red f (n.2)))
            in
              (f (t.0) (f left right))))))))
in (red + input))
```

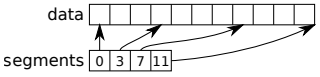

Figure 2. Nested vector example. The data vector stores all 11 elements of the nested vector. The segment vector points to the beginning of each segment and behind the last segment.

4. Optimizations

This section details some of the optimization passes performed by the NOVA compiler. These passes are implemented as graph transformations applied to the abstract syntax tree of a NOVA program. Section 4.1 then details how NOVA handles *nested parallelism* a vital optimization pass for performance of NOVA programs.

Fusion Each application of a higher-order function such as map produces a new vector. To avoid unneccessary allocation of temporary results, the compiler tries to fuse these functions whenever possible. Consider this example:

```
(let (Y (map f X))
 in (map g Y))
```

This code produces a new vector *Y* by applying function *f* to vector *X*. Elements of *Y* are then passed to function *g*. Instead of creating the temporary variable *Y*, the two map operations can be fused:

```
(let
  (h (lambda (x : 'a) : 'b
    (g (f x))))
  in (map h X))
```

This way, we avoid the allocation of a temporary vector.

Currently, the compiler supports fusion of maps inside map, fold and reduce. In addition, filters can be fused, too, when inside a reduce or another filter. Fusing filters is especially important because they are expensive operations.

4.1 Nested parallelism

NOVA supports nested vectors, i.e., vectors whose elements are vectors. Any level of nesting is possible. Nested vectors are stored as a flat data vector holding *all* data values of the vectors and one segment vector for each level describing the shape of the vector. See Figure 2 for an example.

To operate on nested vectors, we need nested parallelism. Consider a vector *X* whose type is **vector vector int**. To add 1 to each element of the vector, we write:

```
(map (map (+ 1)) X)
```

In other words, for all inner vectors of *X*, we apply map with (+ 1). To execute this expression, we have to break up the nested vector X into individual sub-vectors $X_1, \ldots, X_n$ and then apply each of these to **map** (+ 1).

There are several ways to execute the above expression in parallel. We could divide the sub-vectors equally among the available processors and perform the inner map sequentially. However, this can lead to load imbalance because some sub-vectors may be much bigger than others. In a different approach we could execute the outer map sequentially and the inner maps in parallel. This may cause a lot of overhead though especially when the sub-vectors are small.

To avoid these problems, the NOVA compiler can automatically *flatten* the vectors [15]. Since all data values of the sub-vectors are stored contiguously we can simply apply the map on the flattened data without any overhead. However, the result of the map is now a flat vector and the compiler must *unflatten* it using the shape of the input vector. The above expression gets thus transformed into:

```
(unflatten (map (+ 1) (flatten X)))
```

While nested maps are simple to deal with a reduce inside of a map is more complex:

```
(map (reduce + 0) X)
```

Simply performing the reduction on the flattened array is wrong because we have to take the segment boundaries into account. The expression is thus transformed into a special node called *segmented reduction* [8]. A segmented reduction works on the flattened data, thus avoiding load imbalance, and adheres to segments, i.e., one value is computed for each segment.

5. Code Generation

The NOVA compiler currently contains three back-ends for code generation: sequential C, parallel (multi-threaded) C and CUDA C. All of them are C-based which means they can be easily integrated in other programs and frameworks.

There are some built-in functions whose use is currently restricted. **gather**, **slice**, and **range** can only be used in conjunction with vector functions such as map, but not on its own. This is because the return value of these functions is never computed as such. It is only used to change the index computation when accessing vector elements. For example,

```
(map f (gather I X))
```

results in (pseudo code):

```
for i in 0 .. N
  x = X[I[i]]
  ...
```

Slice and range are handled similarly.

The next sections describe how higher-level built-in functions are handled in the different code generators.

5.1 Sequential C code generation

When generating sequential C code, all higher-level built-in functions are mapped to loops. A reduction (**reduce** *f i X*), for example, gets translated to

```
accu = i
for it in 0 .. N
  accu = f(accu, X[it])
```

Segmented reduction is implemented as a nested loop with the outer loop iterating over segments and the inner loop iterating over the elements of that segment.

Tuples For every tuple type in a NOVA program, a new struct type is declared. The components of the struct reflect the components of the tuple. A tuple value is thus represented as an object of the corresponding struct.

Closures Closures are represented by objects containing a pointer to the closure function and memory to hold the values of free variables. On encountering a closure, a new closure object is created and the values of the free variables are captured. At function applications, the function associated with the closure is called and the closure itself and the argument are passed. Inside the function, the captured values are unpacked from the closure object and the function body is evaluated.

Foreign functions When using foreign functions in NOVA, a certain signature is expected based on the function's type. The return value of the C implementation of a foreign function is always void. The first parameter to the function is a pointer to the result variable which is followed by the function's parameters. The following rules explain how NOVA types are mapped to C types: primitive values are mapped to their C counterparts (**bool** is mapped to int); tuples are represented by a corresponding struct as explained above; vectors are represented by a pointer to a data array and a length (of type int). If the vector is nested, there will be a pointer for each nest pointing to the segment descriptors at that level. In that case, the length of the vector is the length of the first segment descriptor. Foreign functions must not have functions as parameters. The signatures of foreign functions can be found in the header file.

Example:

```
(ff : vector float → float → vector float)
```

has the signature

```
void ff (int*, float**, float, int, float*)
```

If the return value of a foreign function is a vector, the function is expected to allocate the memory for it.

5.2 Parallel C code generation

The parallel C code generator generates code to run on a multi-core CPU using multiple threads. On encountering a parallel operation such as map or reduce, the generated code calls a multi-core runtime passing it information on how to process this operation. This data contains a pointer to the function to be executed as well as the data needed to execute the function. The runtime passes control back to the host program when the operation has finished.

The function that corresponds to the operation is essentially a sequential version of the operation. However, instead of processing the entire input, it only processes a section of the input. Information on which part of the input to process is passed together with the input data. It is the runtime's responsibility to split the work between the CPU cores.

Some operations require individual results to be merged. When performing a reduction, for example, each thread may compute the result for a share of the input. In this case the runtime passes the results back to the host program which then performs the final reduction step sequentially.

Multi-core runtime The current implementation of the multi-core runtime is straightforward: When a parallel operation needs to be performed, it creates a certain number of threads. Each thread gets assigned an equal share of the input to process. The number of threads can be specified by the user setting the `NOVA_NUM_THREADS` environment variable. If the variable is not set, the runtime creates as many threads as there are CPU cores.

5.3 CUDA C code generation

When targeting the GPU, NOVA code gets translated to CUDA C. Parallel operations result in CUDA kernel launches to perform the operation. The map operation, for example, gets translated to a kernel where each thread computes one or more elements of the

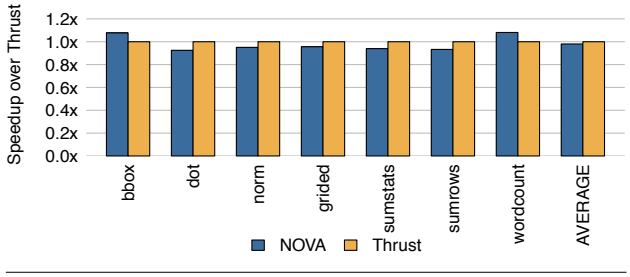

Figure 3. Speedup over Thrust.

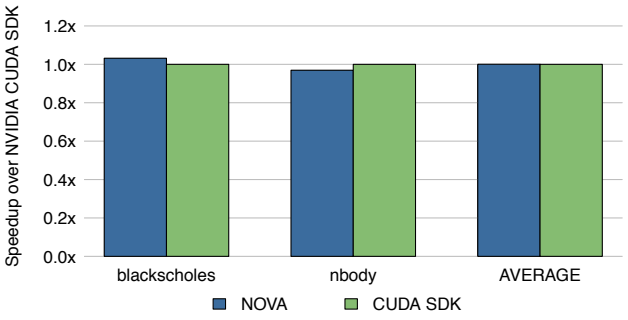

Figure 4. Speedup over NVIDIA CUDA SDK.

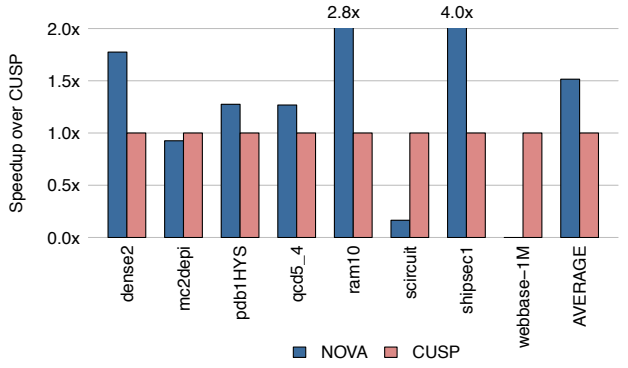

Figure 5. Speedup over CUSP sparse matrix-vector multiplication benchmark for a range of input matrices.

result vector. Other operations are slightly more complex because they require communication between threads.

There is a default maximum number of thread blocks that are created at kernel launch as well as a default block size. These values can be changed by the user in the generated header file. If there are more elements to process than there are threads being launched, each thread processes multiple elements sequentially inside the kernel.

In NOVA, the data resides in main memory initially. When a kernel launch is encountered, the data needed to perform the computation is copied to the GPU's device memory. After a kernel has finished, the result is copied back to the host memory. Every variable gets copied exactly once even if it is used multiple times. Since the value of variables in NOVA cannot be changed the copies of variables are never out-of-date.

A tree-based reduction is used for both normal and segmented reduction [19]. The scan operation is implemented in three steps: a partial reduction, followed by a scan on the intermediate result and a "down-sweep" phase to compute the final result [19]. The filter operation is also implemented as a sequence of operations. First, we perform a map operation on the input vector using the filter function. The resulting vector is a mask of ones and zeros indicating for each element if it should be part of the output. A +-scan is performed on the mask resulting in an index vector indicating the position of each element that is part of the output. Finally, the elements are moved from the input vector to the output vector based on the mask and the index vector.

Foreign functions in CUDA C When generating CUDA C code parallel operations, such as map and reduce, are performed on the device. If a foreign function is used within such an operation, it thus needs to be a *device* function (`__device__` in CUDA C).

If a foreign function is used outside of parallel operations, the compiler assumes that it is a *host* function, i.e., written in standard C/C++ code. The function itself may launch kernels on the device but the compiler has no knowledge of that. Any vector arguments the function works on are passed as *host* pointers. If inside the function they are passed to kernels, the user has to allocate the device memory and perform the data copy.

6. Performance Evaluation

This section evaluates the performance of NOVA generated code for both CPU and GPU systems. The experiments were run on an 8-core CPU and NVIDIA GeForce GTX480 using CUDA 4.1. We use a suite of 10 benchmark programs, taken from the Trust source distribution and CUDA SDK.

6.1 Comparison with Existing Languages

Figures 3 and 4 show performance results of the CUDA C code generated by NOVA and Thrust [6], and the hand-written CUDA C code for the benchmarks from the NVIDIA CUDA SDK. We measure the kernel-execution time only. This does not include memory transfers and other CUDA device initialization. This allows direct comparison of the quality of the CUDA C code generated by each approach, which would otherwise be skewed by memory transfer times.

In Figure 3, we compare the performance of NOVA to that of Thrust on 7 benchmarks. Both NOVA and Thrust achieve similar performance. The performance results for NOVA and Thrust are within 10% of each other.

Figure 4 compares the performance of NOVA to hand-written CUDA C code from the NVIDIA CUDA SDK. On both benchmarks, the performance of the NOVA code is within 4% of the hand-written code. Again, the two approaches achieve similar performance.

Figure 5 shows the performance of the SpMV benchmark across a range of different input matrices. The matrix format used is ELL [16]. The performance of the NOVA generated code is compared against CUSP [5], a library of carefully-tuned sparse matrix algorithms. The NOVA-generated CUDA C codes significantly outperform the CUSP generated code on all but three of the benchmarks (mc2depi, scircuit and webbase-1M). On average, over all of the benchmarks, NOVA achieves a speedup of 1.5× over the CUSP implementations.

6.2 Performance on multi-core CPUs

Figure 6 shows the speedup of the parallel C++ code over the sequential version. The experiments are run on an 8-core Intel i7 CPU and the number of threads is varied from 1 to 8.

Most applications demonstrate good scaling behavior. The performance roughly scales linearly with the number of threads. However, for nbody and box3x3 benchmarks, the performance only improves marginally when using more than 4 threads. These appli-

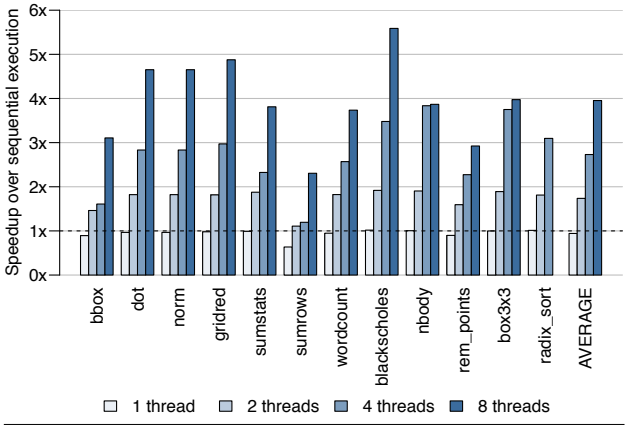

Figure 6. Speedup over sequential execution on 8-core CPU.

cations are memory-bound, thus adding more compute resources does not improve performance significantly. Their performance is bounded by the memory bandwidth of the system which does not scale linearly with the number of cores used.

7. Related Work

The closest work to our own is on generating parallel target code from functional languages. Data Parallel Haskell [15] (DPH) allows programmers to express nested parallelism [9] in Haskell. Nested parallelism is particularly useful for irregular computation and recursive parallelism. An important optimization is 'vectorization' which flattens nested computations to better load balance work across multiple processors. Our approach, described in Section 4.1, extends this by allowing nested parallelism on GPUs as well as multi-core CPUs. However, we do not support recursive nesting. In contrast to DPH, which adds parallelism for CPUs to Haskell, Accelerate [11] adds GPU support to Haskell. Accelerate uses array-based operations such as `zipWith` and `fold` to map data parallelism to GPUs. It does not support nested arrays. Copperhead [10] is a language for GPU computing embedded in Python. Similarly to Accelerate and NOVA, it uses array-based operations such as `map` and `scan`. Copperhead supports nested parallelism but unlike DPH the computation is not flattened. Instead the different levels are mapped to the hierarchy of the GPU, including thread blocks and threads. There also have been efforts to compile Nesl to CUDA for GPUs [7, 21].

8. Conclusions and Future Work

This paper has presented NOVA, a functional language and compiler for parallel computing. NOVA allows users to write parallel programs using a high-level abstraction. This makes the code concise and maintainable, but also performance portable across a variety of processors.

NOVA provides support for integrating existing code into NOVA programs through the use of *foreign functions*. However, foreign functions are assumed to be side effect free. We are extending NOVA with support for monads to allow side effects to be handled in a safe manner. We are also experimenting with the use of NOVA as an intermediate language.

References

[1] The OpenACC application programming interface, 2011. URL `http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf`. Version 1.0.

[2] The OpenCL specification version 1.2, 2011. URL `http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf`.

[3] Intel Threading Building Blocks reference manual, 2011. URL `http://software.intel.com/sites/products/documentation/hpc/tbb/referencev2.pdf`.

[4] CUDA C programming guide version 4.1, 2012. URL `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`.

[5] N. Bell and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2012. Version 0.3.0.

[6] N. Bell and J. Hoberock. Thrust: A productivity-orientied library for CUDA. In *GPU Computing Gems: Jade Edition*. 2011.

[7] L. Bergstrom and J. Reppy. Nested data-parallelism on the gpu. In *ICFP*, 2012.

[8] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990.

[9] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel Distrib. Comput.*, 21(1):4–14, 1994.

[10] B. C. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *PPOPP*, 2011.

[11] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore GPUs. In *DAMP*, 2011.

[12] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing Using GPUs*, 2007.

[13] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[14] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. 1985.

[15] S. L. P. Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in haskell. In *FSTTCS*, 2008.

[16] D. R. Kincaid, J. R. Respess, and D. M. Young. ITPACK 2.0 user's guide. Technical Report CNA-150, Center for Numerical Analysis, University of Texas, Austin, Texas, 1979.

[17] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 1978.

[18] J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, 1974.

[19] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Graphics Hardware*, 2007.

[20] M. Wolfe. Implementing the PGI accelerator model. In *GPGPU*, 2010.

[21] Y. Zhang and F. Mueller. Cunesl: Compiling nested data-parallel languages for simt architectures. In *ICPP*, 2012.