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
asusnea@nvidia.com

Abstract

Functional languages provide a solid foundation on which complex optimization passes can be designed to exploit available parallelism 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 '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.

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, 9–11, 17, 24]. 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.

1.1 Contributions

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.

1.2 Structure

The rest of this paper is structured as follows. Section 2 presents an example that motivates our work. Section 3 details the NOVA language, and Section 4 describes the NOVA compiler. Section 5 presents more details of the optimization passes performed by the compiler. Section 6 describes the code generation phase of the compiler. We evaluate the performance and scalability of NOVA generated code in Section 7. Related work is discussed in Section 8 and we conclude in Section 9.

* Direct all correspondence to vgrover@nvidia.com.

NVIDIA Technical Report NVR-2013-002, Jul. 2013.
© 2013 NVIDIA Corporation. All rights reserved.

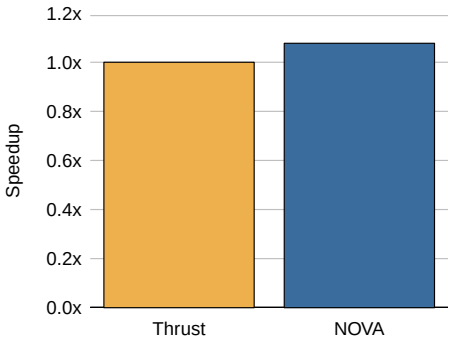

Figure 1. Performance of the NOVA and Thrust implementations of the bounding box algorithm. Higher is better.

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.

Figure 1 shows the performance achieved by the NOVA and Thrust versions of the bounding box algorithm. The experiments were run on a NVIDIA GeForce GTX480 using CUDA 4.1. NOVA achieves a speedup in execution time of 1.07× over Thrust.

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 2 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 [16]. A representative set of the parallel operations provided by NOVA are listed in Table 1.

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

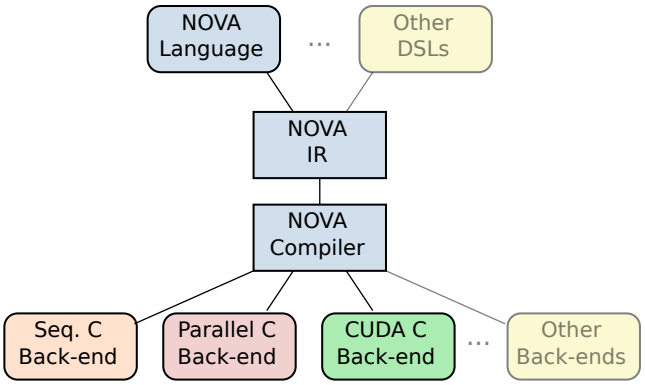

Figure 2. Summary of the structure of the NOVA compiler infrastructure, with a common NOVA-Intermediate Representation, and multiple front-ends and back-ends.

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 rest of this section is structured as follows. Section 3.1 describes the salient features of NOVA, Section 3.2 describes the syntax of NOVA, Section 3.3 details the polymorphic type system and Section 3.4 presents informal operational semantics for the language. Section 3.5 shows two simple example programs, highlighting the salient features of the language.

3.1 Language Features

NOVA includes the usual functionality you would expect from a functional language including lambda expressions and let-expressions.

3.1.1 Foreign Functions

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))
)
```

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

3.1.2 Algebraic Data Types

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.

3.1.3 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:

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

3.1.4 Type Generalization and Specialization

NOVA allows type generalization and specialization, in a similar manner to System F [13, 21]. Polymorphic types can be defined as follows:

```
(types
  (List 'a :
    (+ (Nil : unit)
       (Cons : ('a, (List 'a))))))
```

This example demonstrates both type *generalization* and type *specialization*. *List* is the type constructor:

```
forall 'a . (+ (Nil : unit)
               (Cons : ('a, (List 'a)))))
```

This type can be specialised to store a list of integers by using a type application:

```
(List int)
```

This produces the concrete type:

```
(+ (Nil : unit)
   (Cons : (int, (List int))))
```

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.

3.1.5 Type Inference

The NOVA language performs Hindley-Milner type inference [20], 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.

3.2 Syntax

The syntax of NOVA is based on *S-expressions* [19]. A large subset of the syntax is given in Figure 3. Operations on primitive types, such as arithmetic operations, are not explicitly represented in the syntax. Instead, they are encoded as function applications to 'built-in' functions. For example, adding numbers a and b is written as $(+\ a\ b)$. This makes code generation simpler for the compiler front ends that target NOVA, as the operator precedence and associativity are explicitly specified.

A NOVA program consists of multiple sections. First, the input variables have to be specified. The compiled program will expect these variables to be passed by the user at runtime. The type of every input must be specified, as programs are statically typed. Optional *foreign functions* are declared in the second section, according to the required convention. Another optional section is used to specify user-defined types, such as sum types. The final section of a NOVA program is an expression specifying the program code.

Expressions can either be simple expressions (such as identifiers and constants) or composite expressions (such as conditionals and **let** expressions). Composite expressions generally have to be enclosed in parenthesis. The only exceptions are function applications and **let** expressions. These are expanded into nested, single argument expressions before being type checked. For example, applying function f to arguments x and y will be transformed into a pair of function applications. $(f\ x\ y)$ is treated as $((f\ x)\ y)$. Our subsequent description of the operational semantics assumes transformation has been performed.

Functions can be defined using **lambda** expressions. Multiple formal parameters can be specified for a **lambda** expression. Similarly to function application, these are transformed into nested **lambda** expressions. Again, our later description of the operational semantics assumes that this transformation has been performed. For example, the following expression:

```
(lambda (x : int) (y : int) : int (+ x y))
```

is transformed into:

```
(lambda (x : int) : (int → int)
  (lambda (y : int) : int (+ x y))
```

Tuple elements can only be accessed using a constant index. This restriction is necessary to ensure that the accessed tuple element type is known at compile time. NOVA contains no vector

Const	= [0 − 9]$^+$.$^?$[0 − 9]* | `true` | `false` | `nil`	Constants
ArithOp	= `+` | `-` | `/` | `*` | `%`	Arithmetic operators
CompareOp	= `=` | `~=` | `<` | `<=` | `>` | `>=`	Comparison operators
LogicOp	= `and` | `or` | `not`	Logical operators
MathOp	= `abs` | `sqrt` | `exp` | `log`	Math operators
BitewiseOp	= `&` | `|` | `^` | `~` | `<<` | `>>`	Bitwise operators
VectorOp	= `map` | `reduce` | `permute` | `gather` | `slice`	Vector operators
	| `range` | `length` | `scan` | `filter`	
BuiltInOp	= ArithOp | CompareOp | LogicOp	Built-in operations
	| MathOp | BitwiseOp | VectorOp	
Id	= BuiltInOp | a string that is not a reserved word	Identifiers
SimpleType	= `int` | `int32` | `int16` | `int8` | `float` | `double` | `bool` | `unit`	Basic types
	| `(`SimpleType{`,` SimpleType}$^+$`)`	Tuple type
	| `+` TypeCtorDecl$^+$	Sum type
	| `vector` SimpleType	1D vector
	| `vector#`n SimpleType	nD vector, where $n \geq 1$
Type	= Id	Type identifier
	| Type `->` Type	Function type
	| `(`SimpleType`)`	Parentheses
	| SimpleType	Simple types
Expr	= Id | Const	
	| `(`Expr Expr$^+$`)`	Function application
	| `(`Expr {`,` Expr}$^+$`)`	Tuple constructor
	| `(`Expr`.`Const`)` | `(`Expr`!`Expr`)`	Tuple and vector access
	| `(if` Expr `then` Expr `else` Expr`)`	Conditional expression
	| `(case` Id CasePattern$^+$`)`	Case expression
	| `(let` VarDef$^+$ `in` Expr`)`	Let expression
	| `(lambda` ParamDecl$^+$ `:` Type Expr`)`	Lambda expression
	| `(mu` ParamDecl Expr`)`	Mu (recursive) expression
	| `(`Expr`)`	
CasePattern	= `(`Id Id Expr`)`	Pattern in a case expression
VarDef	= `(`Id Expr`)`	Let variable definitions
ParamDecl	= `(`Id `:` Type`)`	Lambda parameter declarations
InputDecl	= `(`Id `:` SimpleType`)`	Input variable declarations
TypeDecl	= `(`Id `:` SimpleType`)`	Type identifier declaration
TypeCtorDecl	= `(`Id `:` SimpleType`)`	Type constructor declaration
Program	= `(`InputDecl$^+$`)` {`(types` TypeDecl$^+$`)`}$^?$ {`(foreign` ParamDecl$^+$`)`}$^?$ `(`Expr`)`	NOVA programs

Figure 3. Syntax of NOVA programs. Non-terminals in the grammar are denoted NonTerminal and terminals as `terminal`. The pipe character | separates rules. Regular expressions are used to simplify the grammar: e^+ denotes e repeated 1 or more times, and e^* that e is repeated 0 or more times. $e^?$ denotes that e is optional. $[0-9]$ denotes all integers from 0 to 9. { and } are used to group terms and do not themselves appear in the language.

$$\text{(CONST)}\ \frac{c \text{ is a constant of type } \tau}{\Gamma \vdash c : \tau} \qquad \text{(ID)}\ \Gamma\{x:\tau\} \vdash x : \tau$$

$$\text{(APP)}\ \frac{\Gamma \vdash e_1 : \tau_2\ \texttt{->}\ \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ \ e_2 : \tau_1}$$

$$\text{(TUPLE-CTOR)}\ \frac{\Gamma \vdash e_i : \tau_i \qquad \tau_i \in \underline{\text{SimpleType}}}{\Gamma \vdash \texttt{(}e_1\texttt{, ..., }e_n\texttt{)} : \texttt{(}\tau_1\texttt{, ..., }\tau_n\texttt{)}}$$

$$\text{(TUPLE-ACCESS)}\ \frac{\Gamma \vdash e : \texttt{(}\tau_0\texttt{, ..., }\tau_n\texttt{)} \qquad \Gamma \vdash c : \texttt{int} \qquad 0 \le c \le n}{\Gamma \vdash e_1\texttt{.}c : \tau_c}$$

$$\text{(VECTOR1-ACCESS)}\ \frac{\Gamma \vdash e_1 : \texttt{vector\#1}\ \tau \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\texttt{!}e_2 : \tau}$$

$$\text{(VECTORN-ACCESS)}\ \frac{\Gamma \vdash e_1 : \texttt{vector\#}n\ \tau \qquad \Gamma \vdash e_2 : \texttt{(int, ..., int)} \qquad n > 1}{\Gamma \vdash e_1\texttt{!}e_2 : \tau}$$

$$\text{(IF)}\ \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

$$\text{(CASE)}\ \frac{\Gamma \vdash x : (T_1 : \tau_1) + \ldots + (T_n : \tau_n) \qquad \Gamma\{x_i : \tau_i\} \vdash e_i : \tau}{\Gamma \vdash \texttt{case}\ x\ \ (T_1 x_1 e_1) \ldots (T_n x_n e_n) : \tau}$$

$$\text{(LET)}\ \frac{\Gamma \vdash e' : \tau' \qquad \Gamma\{x:\tau'\} \vdash e : \tau}{\Gamma \vdash \texttt{let}\ \ (x\ \ e')\ \ \texttt{in}\ \ e : \tau}$$

$$\text{(LAMBDA)}\ \frac{\Gamma\{x:\tau'\} \vdash e : \tau}{\Gamma \vdash \texttt{lambda}\,(x\texttt{:}\tau')\texttt{:}\tau\,(e)\ \texttt{:}\ \tau'\ \texttt{->}\ \tau}$$

$$\text{(MU)}\ \frac{\Gamma\{x_1 : \tau_1\} \vdash \texttt{lambda}\,(x_2\texttt{:}\tau_2)\texttt{:}\tau_3\,(e)\ \texttt{:}\ \tau_1}{\Gamma \vdash \texttt{mu}\,(x_1\texttt{:}\tau_1)\ (\texttt{lambda}\,(x_2\texttt{:}\tau_2)\texttt{:}\tau_3\,(e))\ \texttt{:}\ \tau_1}$$

$$\text{(PARENTHESES)}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{(}e\texttt{)}\ \texttt{:}\ \tau}$$

$$\text{(PROGRAM)}\ \frac{\Gamma_\emptyset \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \vdash e : \tau \qquad \tau_1, \ldots, \tau_n \in \underline{\text{SimpleType}}}{\Gamma_\emptyset \vdash\ \texttt{(}\texttt{(}x_1\texttt{:}\tau_1\texttt{)}\ \ldots\ \texttt{(}x_n\texttt{:}\tau_n\texttt{)}\texttt{)}\ \texttt{(}e\texttt{)}\ \texttt{:}\ \tau}$$

Figure 4. Type rules for the NOVA language. The notation is explained in Section 3.3

$$\text{(VALUE)}\ \langle v, \sigma\rangle \Downarrow v \qquad \text{(ID)}\ \langle x, \sigma\rangle \Downarrow \sigma(x) \qquad \text{(PARENTHESES)}\ \frac{\langle e, \sigma\rangle \Downarrow v}{\langle\, \texttt{(}e\texttt{)}\,, \sigma\rangle \Downarrow v}$$

$$\text{(APP)}\ \frac{\langle e_1, \sigma\rangle \Downarrow \texttt{lambda}\,(x\texttt{:}\tau')\texttt{:}\tau\,(e) \qquad \langle e_2, \sigma\rangle \Downarrow v' \qquad \langle e, \sigma[x \mapsto v']\rangle \Downarrow v}{\langle e_1 e_2, \sigma\rangle \Downarrow v}$$

$$\text{(MU-APP)}\ \frac{\langle e_1[x_1/\texttt{mu}\,(x_1\texttt{:}\tau_1)\,e_1]e_2, \sigma\rangle \Downarrow v}{\langle(\texttt{mu}\,(x_1\texttt{:}\tau_1)\,e_1)\,e_2, \sigma\rangle \Downarrow v}$$

$$\text{(TUPLE-CTOR)}\ \frac{\langle e_i, \sigma\rangle \Downarrow v_i}{\langle\, \texttt{(}e_1\texttt{, ..., }e_n\texttt{)}\,, \sigma\rangle \Downarrow \texttt{(}v_1\texttt{, ..., }v_n\texttt{)}}$$

$$\text{(TUPLE-ACCESS)}\ \frac{\langle e, \sigma\rangle \Downarrow \texttt{(}v_1\texttt{, ..., }v_n\texttt{)} \qquad c \in \{1, ..., n\}}{\langle e\texttt{.}c, \sigma\rangle \Downarrow v_c}$$

$$\text{(VECTOR-ACCESS)}\ \frac{\langle e_1, \sigma\rangle \Downarrow v_1 \qquad \langle e_2, \sigma\rangle \Downarrow v_2 \qquad v \text{ is the element at position } v_2 \text{ in vector } v_1}{\langle e_1\texttt{!}e_2, \sigma\rangle \Downarrow v}$$

$$\text{(IF-TRUE)}\ \frac{\langle e_1, \sigma\rangle \Downarrow \texttt{true} \qquad \langle e_2, \sigma\rangle \Downarrow v}{\langle \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \sigma\rangle \Downarrow v}$$

$$\text{(IF-FALSE)}\ \frac{\langle e_1, \sigma\rangle \Downarrow \texttt{false} \qquad \langle e_3, \sigma\rangle \Downarrow v}{\langle \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \sigma\rangle \Downarrow v}$$

$$\text{(CASE)}\ \frac{\sigma(x) = T_i v \qquad \langle e_i, \sigma[x_i \mapsto v]\rangle \Downarrow v'}{\langle \texttt{case}\ x\ \ (T_1 x_1 e_1) \ldots (T_n x_n e_n), \sigma\rangle \Downarrow v'}$$

$$\text{(LET)}\ \frac{\langle e, \sigma\rangle \Downarrow v \qquad \langle e', \sigma[x \mapsto v]\rangle \Downarrow v'}{\langle \texttt{let}\,(x\ \ e)\,\texttt{in}\ \ e', \sigma\rangle \Downarrow v'}$$

$$\text{(PROGRAM-TYPEDECL)}\ \frac{\langle\, \texttt{(}\texttt{(}x_1\texttt{:}\tau_1\texttt{)}\ \ldots\ \texttt{(}x_n\texttt{:}\tau_n\texttt{)}\texttt{)}\ \texttt{(}e\texttt{)}\,, \sigma_0\rangle \Downarrow v}{\langle\, \texttt{(}\texttt{(}x_1\texttt{:}\tau_1\texttt{)}\ \ldots\ \texttt{(}x_n\texttt{:}\tau_n\texttt{)}\texttt{)}\ \texttt{(types}\ \ (\phi_1\texttt{:}\tau_1'\texttt{)}\ \ldots\ \texttt{(}\phi_1\texttt{:}\tau_1'\texttt{)}\texttt{)}\ \texttt{(}e\texttt{)}\,, \sigma_0\rangle \Downarrow v}$$

$$\text{(PROGRAM)}\ \frac{\langle e, \sigma_\emptyset[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]\rangle \Downarrow v \qquad \text{for input values } v_1, \ldots, v_n}{\langle\, \texttt{(}\texttt{(}x_1\texttt{:}\tau_1\texttt{)}\ \ldots\ \texttt{(}x_n\texttt{:}\tau_n\texttt{)}\texttt{)}\ \texttt{(}e\texttt{)}\,, \sigma_0\rangle \Downarrow v}$$

Figure 5. Operational semantics of the NOVA language. The notation is explained in Section 3.4

constructor. This means NOVA cannot create vectors with arbitrary values. It can only modify vectors passed as input or created by built-in operations such as **range**.

Besides the primitive, integer, floating point, boolean and unit types, NOVA supports tuples, vectors, function types, sum types and recursive types. A tuple contains a fixed number of elements of different types that are known at compile time. All elements of a vector, on the other hand, must be of the same type, and the number of elements only needs to be known at runtime. Vectors can also be multi-dimensional with the dimension being known at compile time. Both tuples and vectors can contain primitive values, vectors, or tuples. They, however, cannot contain functions.

Functions in NOVA are always unary, i.e. they expect a single parameter. The return type of the function can be another function. The built-in function for addition on **float**, for example, is of type **float** $\rightarrow$ (**float** $\rightarrow$ **float**). This approach is common in functional languages and allows for partial function applications.

3.3 Type System

NOVA allows parametric polymorphism using standard Hindley-Milner type inference[20], with some extensions to allow for polymorphism in the arity of the built-in parallel operators. A program is only valid if all polymorphic types in the program can be specialized to concrete types by the compiler. The type rules, which define NOVAs type system, are given in Figure 4. These rules do not include type inference for the built-in functions, which is explained in Section 3.3.1. We use the following notation:

- Each type rule has the following form:

$$(\text{RULE-NAME})\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \dots \qquad \Gamma_n \vdash e_n : \tau_n}{\Gamma \vdash e : \tau}$$

  This denotes that expression e has type τ (under type environment Γ) if all of the expressions e_i have types τ_i (under corresponding type environment Γ_i).
- Types are denoted τ. Valid types are defined by Type in the syntax specified in Figure 4.
- Expressions are denoted e, constant values are c and variable identifiers are x.
- Type environments are denoted Γ. These map variable identifiers x to types τ. The notation $\Gamma\{x : \tau\}$ denotes mapping Γ updated with identifier x mapped to type τ. Γ_0 denotes an empty type environment, that does not map any identifiers to types.

NOVA allows the user defined polymorphic lambda functions through the use of free type variables, denoted `'a`. For example, the following defines a polymorphic lambda function that increments a number, regardless of it being an integer or floating point number:

```
(lambda (x : 'a) : 'a
  (from_double (+ (to_double x) 1.0L)))
```

from_double and **to_double** are built-in polymorphic casts provided by NOVA, with types $\forall\alpha.\texttt{double} \rightarrow \alpha$ and $\forall\alpha.\alpha \rightarrow \texttt{double}$ respectively.

3.3.1 Types for Built-in Operators

Hindley-Milner does not allow type inference of functions with variable arity. This feature is required by some of the built-in parallel operators. For example, **map** takes a variable number of input vectors. Its signature is **map** $f\ X_1 \dots X_n$, and it has type:

$$\frac{\Gamma \vdash f : (\tau_1, \dots, \tau_n) \rightarrow \tau \qquad \tau_i \text{ are appropriate vector types} \qquad \Gamma \vdash X_1 : \tau_1 \qquad \dots \qquad \Gamma \vdash X_n : \tau_n}{\Gamma \vdash \texttt{map}\ f\ X_1 \dots X_n : \tau}$$

We therefore extend Hindley-Milner type inference with a hand-coded pass to infer the types of the built-in functions. Note that this does not allow variable arity of user defined functions. The type inferrence works as follows:

1. First, regular Hindley-Milner type inference is performed to infer types for all the expressions in the program. If these types are all concrete, type inference is complete. If not, at least one must contain free type variables and further inference is needed.
2. A hand-coded pass is performed that attempts to constrain any free type variables. For example, the arity of a built-in operator may be determinable from the context in which it is used. Also, many of the operators have strict requirements on the types of vectors they operator on.
3. Type inference is repeated (by going back to 1) until either a concrete type is found for all expressions in the program, or the types do not change. In the latter case, type inference fails to statically determine a concrete type for the expressions in the program.

3.4 Operational Semantics

The operational semantics of NOVA is presented in Figure 5. We use the following notation:

- The semantics are given as 'big-step' or 'natural' operational semantics. The notation $\langle e, \sigma\rangle \Downarrow v$ denotes that the evaluation of expression e with machine state σ produces value v.
- Each rule has the following form:

$$(\text{RULE-NAME})\ \frac{\langle e_1, \sigma_1\rangle \Downarrow v_1 \qquad \dots \qquad \langle e_n, \sigma_n\rangle \Downarrow v_n}{\langle e, \sigma\rangle \Downarrow v}$$

  This denotes that with machine state σ, expression e evaluates to value v, if every expression e_i evaluates to a value v_i.
- e is an expression and σ is a mapping from variable identifiers x to values v. Expressions are defined in the syntax by Expr, and variable identifiers by Id (see Figure 3).
- Values v are defined as follows:

$v =$ Const	Constant values
$\mid (v_1, \dots, v_n)$	Tuple values
$\mid Tv$	Sum type values
$\mid \texttt{lambda}\,(x : e')\,:\tau\,(e)$	Lambda expressions

- NOVA is a *call-by-value* language.
- $\sigma(x) = v$ if σ maps variable identifier x to value v.
- $\sigma_\emptyset$ denotes the empty mapping from variable identifiers x to values v.
- The notation $\sigma[x \mapsto v]$ denotes the mapping σ updated with identifier x mapped to value v.

3.5 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))
```

4. The NOVA Compiler

The NOVA compiler consists of several passes. They can be divided into 'basic' passes, which are essential for compilation, optimization passes, which are optional, and code generation. Figure 6 shows the default order of passes in the compiler. The 'Reader' parses the input source code and produces an *abstract syntax tree* (AST). All subsequent passes work directly on this AST, by applying transformations to it or annotating it with additional information. The optimization passes, 'Inlining' through to 'Tail Call Elimination', are called repeatedly until either there is no change to the AST or a specified maximum number of iterations is reached. Finally, one of the code generators is called to produce the output code. The user chooses which target they wish to generate code for, through a command line argument to the compiler.

4.1 Basic Passes

The basic passes of the compiler perform the minimum set of tasks needed to enable code generation. Further optimization passes and code generation can only be performed if these passes succeed.

Reader The reader parses a NOVA program to produce an AST. The reader does not check for syntax, type or semantic correctness.

Monomorphization This pass inlines the bodies of polymorphic lambda expressions at their call sites. This simple transformation allows concrete types to be inferred for an expression whilst allowing let-bound polymorphism.

Type checker The type checker ensures type correctness and performs type inference. It also tries to infer the type of built-in functions and numerical constants because their type is dependent on the context in which they are used. After type checking, every node in the AST is annotated with type information.

Assign unique IDs This pass assigns unique IDs to all identifiers, so that optimization passes do not encounter naming conflicts.

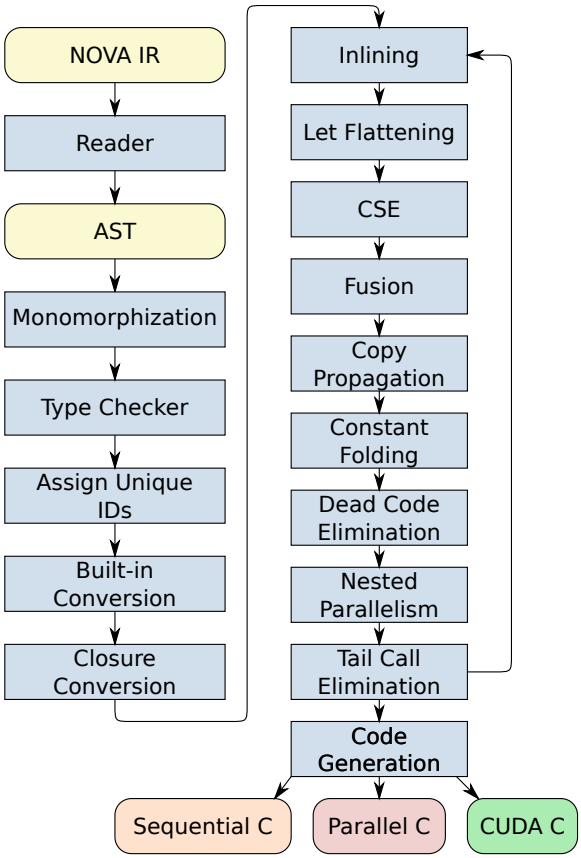

Figure 6. Overview of the passes in the NOVA compiler. Arrows indicate the flow of information between the passes. Rounded-boxes indicate input/output/intermediate code, and sharp-cornered boxes indicate compiler passes.

Built-in conversion Built-in functions in NOVA are curried, i.e. they expect one argument at a time and do not necessarily need all arguments at a call site (partial application). However, at later stages, e.g. fusion or code generation, all arguments to built-in functions must be known. The compiler thus converts all built-in functions to nested lambdas with *uncurried* 'built-in function applications'. Given an addition on **int**, for example, the compiler inserts the following expression:

```
(lambda (arg0 : int) : int → int
    (lambda (arg1 : int) : int
        (@+ [] (arg0, arg1))))
```

The notation `@+ [] (...)` stands for an application to an uncurried built-in function.

For higher-order functions such as **map** or **reduce**, the compiler inserts special 'parameter expressions'. They represent the arguments to the functions that are being passed to the built-in function. Consider the following function application:

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

This is equivalent to:

```
@map [0] (((+ 1) _p0_), X)
```

The number in brackets is the ID of the built-in function parameter which is represented as *_p0_*. In the case of **map**, this parameter stands for an element of the vector argument. Introducing this parameter to the AST allows later passes to perform optimizations

such as inlining on the AST. Built-in conversion transforms this expression into:

```
@map [0] (@+ [] (1, _p0_), X)
```

Without making this parameter explicit in the AST, this would have not been possible at this stage.

Closure conversion The closure conversion [16] pass transforms every lambda expression into a closure. It searches the body of the lambdas for free variables that must be captured by the closure. It also creates 'named' closures for recursive expressions, so that a recursive closure can refer to itself.

4.2 Optimization passes

All optimization passes in the NOVA compiler are optional and can be enabled/disabled from the compilers command line interface. One exception is that the CUDA C code generator expects function applications to be inlined because closures are not supported by the code generated for CUDA C. The passes are detailed in Section 5.

4.3 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 and do not have any external dependencies. This means they can be easily integrated into other programs and frameworks. The back-ends are detailed in Section 6.

4.4 Deployment

NOVA code only describes computational kernels. The rest of the application, for example Input/Output, is written in a separate language such as C. The NOVA compiler generates two files, a header and a source file. The header file contains the function declaration corresponding to the NOVA code. It constitutes the entry point to the NOVA program. The user needs to include the header file to be able to run the NOVA code.

The source file contains the actual implementation of the NOVA code. In the case of CUDA C code it contains both the host code, for data management and kernel launches, and the kernel code. The user thus only needs to pass the host arrays to the function and does not have to deal with communication between the host and the device.

5. 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 5.1 then details how NOVA handles *nested parallelism* a vital optimization pass for performance of NOVA programs.

Inlining The NOVA compiler performs aggressive inlining. Given a function application, where the function is known at compile time, it is replaced by the function body with the inlined argument. Instead of simply replacing every occurence of the function parameter with the argument, a new identifier is introduced which gets assigned the value of the argument. The function parameter is then replaced by the new identifier. This corresponds to the *call-by-value* evaluation strategy and eliminates multiple evaluations of the argument.

For example, the inliner transforms:

```
(let (inc (lambda (x:int) : int (+ x 1)))
 in (inc (+ a b)))
```

into:

```
(let (inc (lambda (x:int) : int (+ x 1)))
 in (let (x1 (+ a b) in (+ x1 1))))
```

Sometimes the function being called cannot be determined statically. For example, when the function expression is a conditional:

```
((if C then f else g) x)
```

In this case the compiler is unable to apply inlining.

Let flattening The inlining pass introduces a number of nested lets - one for each argument. The let flattening pass flattens nested lets into a single flat let. For example,

```
(let (a 2)
 in (let (b 3)
      in (let (c 4)
           in (+ (- a b) c))))
```

will be transformed to:

```
(let (a 2)
     (b 3)
     (c 4)
 in (+ (- a b) c))
```

This transformation is only valid because identifiers are unique. Otherwise, it may introduce naming conflicts.

Common subexpression elimination The goal of *common subexpression elimination* (CSE) is to avoid redundant computation. Consider the expression

```
((+ (- a b) (- a b)))
```

The subexpression $(-\ a\ b)$ appears twice and is thus redundant. CSE will transform this expression to

```
(let (cse (- a b))
 in (+ cse cse))
```

The compiler also performs CSE on the vector arguments of higher-order functions. If the same vector gets passed multiple times to a higher-order function, the AST gets modified such that it is only passed once. This may allow for more optimizations. Consider the following example:

```
(let (f (lambda (x:int) (y:int) (z:int)
                        : int
                (+ (- x z) (- y z))))
 in (map f X X Z))
```

After built-in conversion, inlining, dead code elimination, and other optimizations, this expression looks as follows:

```
@map [0,1,2]
     (@+ [] (@- [] (_p0_, _p2_),
             @- [] (_p1_, _p2_)),
      X, X, Z)
```

Considering the body of the map on its own, there is no possibility for CSE. Knowing, however, that *_p0_* and *_p1_* are the same, namely elements from vector *X*, we can simplify this expression. The compiler detects that the first and second vector arguments to the map are equal and performs the following transformation:

```
@map [0,2]
     (@+ [] (@- [] (_p0_, _p2_),
             @- [] (_p0_, _p2_)),
      X, Z)
```

Now we can use the previously described CSE method to simplify the expression to

```
@map [0,2]
     ((let (cse @- [] (_p0_, _p2_))
       in @+ [] (cse, cse)),
      X, Z)
```

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))
```

or the equivalent

```
(let (Y (@map [0] ((f _p0_), X)))
 in (@map [1] ((g _p1_), 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:

```
(@map [2] ((g (f _p2_)), 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.

Copy propagation If a variable A is bound to another variable B (without performing any operation), we can propagate this information and replace all occurences of A with B. For example:

```
(let (t x)
 in (+ (- x y) (- t y)))
```

Copy propagation will result in

```
(let (t x)
 in (+ (- x y) (- x y)))
```

The occurrence of t has been replace by x. Now CSE can eliminate the redundant computation of $(- x\ y)$.

Constant folding If all operands in an operation are constant, the compiler computes the result of the operation at compile time. For example:

```
(let (t (+ 2 3))
 in (+ x t))
```

This will be transformed to

```
(let (t 5)
 in (+ x t))
```

We can now also replace occurrences of t with the constant 5:

```
(let (t 5)
 in (+ x 5))
```

Dead Code Elimination Many optimization passes, e.g., copy propagation or constant folding, leave sub-expressions behind that are never used. This pass eliminates variable bindings where the variable does not have any uses. The example above would be simplified to:

```
(+ x 5)
```

because t is never used.

Tail Call Elimination This optimization pass identifies *named* closures that are tail recursive. The AST nodes for tail-recursive closures are annotated as such. The code generator can use this additional information to improve the performance of the generated code.

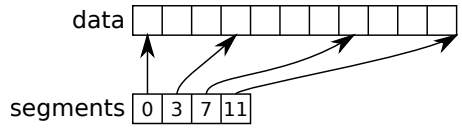

Figure 7. Nested vector example. The data vector stores all 11 elements of the nested vector. The segment vector points to the beginning of each segment and behind the last segment.

5.1 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 7 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 [17]. 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* [7]. 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.

6. 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.

6.1 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.

6.2 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.

6.3 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 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 [22]. 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 [22]. 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.

Benchmark	Description	Source
bbox	Bounding box	Thrust
dot	3D dot product	Thrust
norm	Vector norm	Thrust
gridred	Grid reduction	Thrust
sumstats	Summary of statistics	Thrust
sumrows	Sum of rows (irregular)	Thrust
wordcount	Count words in text	Thrust
blackscholes	Financial modelling	CUDA SDK
nbody	Physics simulation	CUDA SDK
SpMV	Sparse matrix-vector product	CUSP

Table 2. Benchmarks used for the experiments.

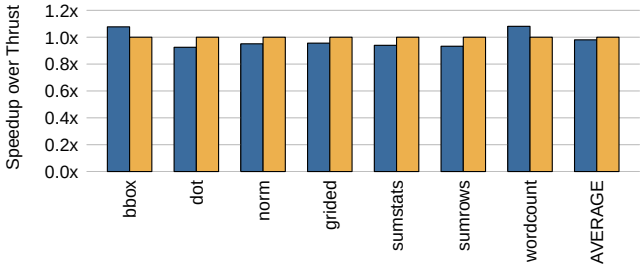

Figure 8. Speedup over Thrust.

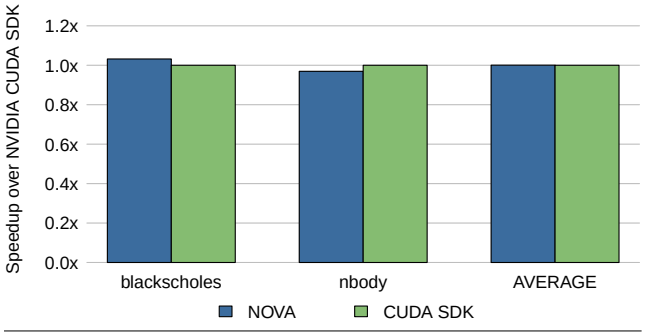

Figure 9. Speedup over NVIDIA CUDA SDK.

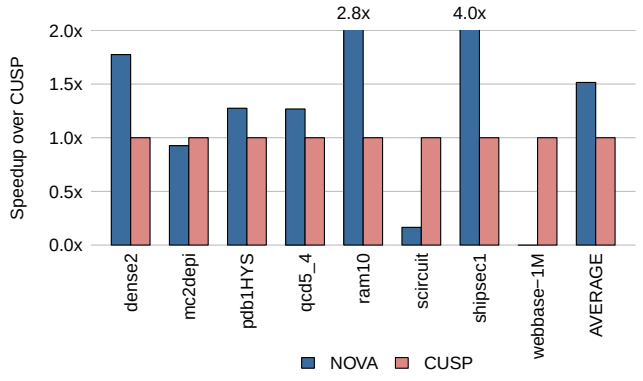

Figure 10. Speedup over CUSP sparse matrix-vector multiplication benchmark for a range of input matrices.

7. 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. Table 2 summarises the benchmarks used.

7.1 Comparison with Existing Languages

Figures 8 and 9 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 8, 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 9 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 10 shows the performance of the SpMV benchmark across a range of different input matrices. The matrix format used is ELL [18]. 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.

7.2 Performance on multi-core CPUs

Figure 11 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 applications 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.

8. Related Work

The closest work to our own is on generating parallel target code from functional languages. Data Parallel Haskell [17] (DPH) allows programmers to express nested parallelism [8] 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 5.1, 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 [10] 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 [9] 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

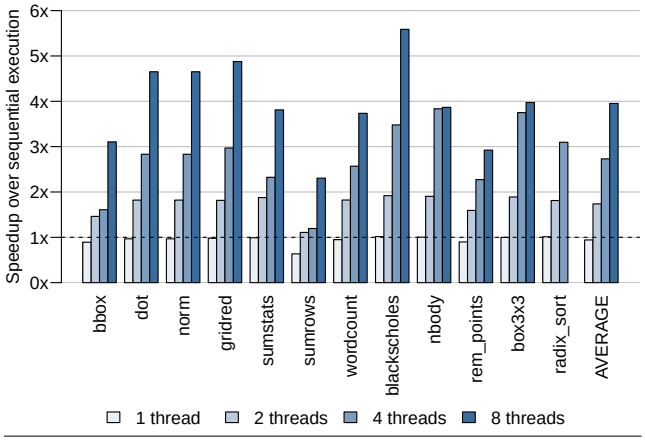

Figure 11. Speedup over sequential execution on 8-core CPU.

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.

Several other approaches target GPUs from high-level languages. Sponge [15] generates portable CUDA C code from programs written in StreamIt [23]. Dubach et al. [12] developed a compiler for Lime, a Java-compatible language targeting heterogeneous systems consisting of CPUs and GPUs. Thrust [6] is a C++ based framework using templates to express parallel code that gets executed on the GPU. It's design closely mirrors that of the C++ Standard Template Library.

Directive-based approaches to GPU programming have become popular. They allow incremental porting of existing applications by annotating code regions that can be run on the GPU. HMPP [11], PGI [24] and hiCUDA [14] are examples of such frameworks. Their work has lead to the publication of the OpenACC standard [1]. The use of directive-based frameworks is generally limited to 'embarrassingly parallel', regular loops.

9. 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.

The NOVA compiler produces code with performance comparable to similar frameworks as well as hand-written code, across a range of benchmarks. NOVA achieves this using a range of optimization passes including aggressive inlining and fusion. The compiler also supports nested parallelism, a powerful mechanism to describe irregular computations.

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 investigating the interoperability of existing data structures with NOVA generated code.

We are also experimenting with the use of NOVA as an intermediate language. Initial results using an R-style data-parallel language are promising, and we will extend the suite of tools that use NOVA to include further domain specific languages.

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

[8] 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.

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

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

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

[12] C. Dubach, P. Cheng, R. Rabbah, D. Bacon, and S. Fink. Compiling a high-level language for gpus (via language support for architectures and compilers). In *PLDI*, 2012.

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

[14] T. D. Han and T. S. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE Transactions on Parallel and Distributed Systems*, 2011.

[15] A. Hormati, M. Samadi, M. Woh, T. N. Mudge, and S. A. Mahlke. Sponge: portable stream programming on graphics engines. In *ASPLOS*, 2011.

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

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

[18] 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.

[19] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communication of the ACM*, 1960.

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

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

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

[23] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, 2002.

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