Vector: A High-Level Programming Language for the GPU

Harry Lee (hhl2114), Howard Mao (zm2169), Zachary Newman (zjn2101), Sidharth Shanker (sps2133), Jonathan Yu (jy2432)

1. Introduction

As the single-core performance of CPUs plateaus and becomes constrained by power consumption, more and more high-performance computing (HPC) must be done in parallel in order to see any performance increases. The current state of the art in HPC systems is using GPUs for compute-heavy tasks, due to the intrinsic parallelism of GPU architectures. However, programming GPUs remains difficult due to a lack of language tools. Currently, the only mature general-purpose GPU computing languages are low level languages, such as OpenCL and CUDA, which expose a lot of the incidental complexity of GPU hardware to the GPU programmer.

To address these issues, we have implemented *Vector*, a high-level programming language for the GPU. In Vector, GPU computation becomes almost as simple as CPU computation. The compiler abstracts away details such as allocating memory on the GPU, copying memory between CPU and GPU, and choosing the proper work sizes (i.e. number of threads per block and number of blocks per grid). In addition, certain parallel programming idioms (such as `map` and `reduce`) are supported natively in Vector as higher-order primitives.

Our strategy for implementing this language is to generate CUDA code, which can then be compiled and run on Nvidia GPUs. We chose to generate CUDA rather than the more platform-independent OpenCL due to our greater familiarity with CUDA. Also, most HPC systems use CUDA-compatible Nvidia Tesla GPUs. During development, we tested using [GPU Ocelot][], an emulator for Nvidia GPUs that runs PTX (the intermediate representation for CUDA) on the CPU.

1.1 Language Features

- basic type inference for assignments
- parallel for (`pfor`) loops: generate a CUDA kernel which is compiled into PTX and run on the GPU
- map/reduce: syntactic sugar built on top of `pfor` (because these generate kernels, we need to implement them as primitives)
- first-class functions (sort of): we can only implement these at compile-time
- handles memory allocation and communication between CPU and GPU
- abstracts grid and block sizing for kernels
- handles higher-dimensional kernel indices (CUDA handles at most 3)

1.2 Sample Program

```
__device__ int add(int x, int y) { return x + y; }

/*
 * Compute the dot product of two vectors
 */
int dot_product(int x[], int y[]) {
    int z[len(x)];

    pfor (i in 0:len(x)) {
        // each iteration of the for loop runs on a different thread on the GPU
        z[i] = x[i] * y[i];
    }

    return @reduce(add, z);
}
```

The equivalent CUDA program would look like this.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <libvector.hpp>

__device__ int32_t add (int32_t, int32_t);
__global__ void
__sym5_ (int32_t * output, int32_t * input, size_t n)
{
  extern __shared__ int32_t temp[];

  int ti = threadIdx.x;
  int bi = blockIdx.x;
  int starti = blockIdx.x * blockDim.x;
  int gi = starti + ti;
  int bn = min (n - starti, blockDim.x);
  int s;

  if (ti < bn)
    temp[ti] = input[gi];
  __syncthreads ();

  for (s = 1; s < blockDim.x; s *= 2)
    {
      if (ti % (2 * s) == 0 && ti + s < bn)
    temp[ti] = add (temp[ti], temp[ti + s]);
      __syncthreads ();
    }

  if (ti == 0)
    output[bi] = temp[0];
}

int32_t
```

```
__sym4_ (VectorArray < int32_t > arr)
{
  int n = arr.size ();
  int num_blocks = ceil_div (n, BLOCK_SIZE);
  int atob = 1;
  int shared_size = BLOCK_SIZE * sizeof (int32_t);
  VectorArray < int32_t > tempa (1, num_blocks);
  VectorArray < int32_t > tempb (1, num_blocks);

  __sym5_ <<< num_blocks, BLOCK_SIZE, shared_size >>> (tempa.devPtr (),
                               arr.devPtr (), n);
  cudaDeviceSynchronize ();
  checkError (cudaGetLastError ());
  tempa.markDeviceDirty ();
  n = num_blocks;

  while (n > 1)
    {
      num_blocks = ceil_div (n, BLOCK_SIZE);
      if (atob)
    {
      __sym5_ <<< num_blocks, BLOCK_SIZE,
        shared_size >>> (tempb.devPtr (), tempa.devPtr (), n);
      tempb.markDeviceDirty ();
    }
      else
    {
      __sym5_ <<< num_blocks, BLOCK_SIZE,
        shared_size >>> (tempa.devPtr (), tempb.devPtr (), n);
      tempa.markDeviceDirty ();
    }
      cudaDeviceSynchronize ();
      checkError (cudaGetLastError ());
      atob = !atob;
      n = num_blocks;
    }

  if (atob)
    {
      tempa.copyFromDevice (1);
      return tempa.elem (false, 0);
    }
  tempb.copyFromDevice (1);
  return tempb.elem (false, 0);
}

__global__ void
__sym3_ (struct range_iter *__sym6_, size_t __sym7_, size_t __sym8_,
     device_info < int32_t > *x, device_info < int32_t > *y,
     device_info < int32_t > *z)
{
  size_t __sym9_ = threadIdx.x + blockIdx.x * blockDim.x;
  if (__sym9_ < __sym8_)
    {
```

```
      size_t i = get_index_gpu (&__sym6_[0], __sym9_);
      {
    z->values[get_mid_index < int32_t > (z, i, 0)] =
      (x->values[get_mid_index < int32_t > (x, i, 0)]) *
      (y->values[get_mid_index < int32_t > (y, i, 0)]);
      }
    }
}

__device__ int32_t
add (int32_t x, int32_t y)
{
  return (x) + (y);
}

int32_t
dot_product (VectorArray < int32_t > x, VectorArray < int32_t > y)
{
  VectorArray < int32_t > z (1, (x).size ());
  {
    struct range_iter __sym0_[1];
    __sym0_[0].start = 0;
    __sym0_[0].stop = (x).size ();
    __sym0_[0].inc = 1;
    fillin_iters (__sym0_, 1);
    struct range_iter *__sym1_ = device_iter (__sym0_, 1);
    size_t __sym2_ = total_iterations (__sym0_, 1);
    __sym3_ <<< ceil_div (__sym2_, BLOCK_SIZE), BLOCK_SIZE >>> (__sym1_, 1,
                                __sym2_,
                                x.devInfo (),
                                y.devInfo (),
                                z.devInfo ());
    cudaDeviceSynchronize ();
    checkError (cudaGetLastError ());
    z.markDeviceDirty ();
    cudaFree (__sym1_);
  }

  return __sym4_ (z);
}

int
main (void)
{
  return vec_main ();
}
```

2. Language tutorial

Much of the Vector language is very similar to C. Therefore, to keep this tutorial short, we focus mainly on the differences between Vector and C. We assume readers have a basic understanding of C.

2.1 Environment Settings

To compile and run Vector Language programs, you must have a CUDA compiler and a working implementation of the CUDA libraries. We developed and tested Vector using the gpuocelot (code.google.com/p/gpuocelot). This tutorial from here on assumes that users have a working environment.

2.2 Building the Compiler

Compilation of the Vector compiler requires OCaml (version 4.0.0 or higher, including ocamllex and ocamlyacc) and the SCons build system, which requires Python2. Simply type 'scons' in the build directory and the compiler binary will be named 'generator' and located in the 'compiler' subdirectory.

2.3 Hello World

The Vector language will be very familiar to users who have prior experience with C-like languages. Without further ado, here is the Hello World program in Vector:

```
int vec_main()
{
    printf("Hello, World!\n");
    return 0;
}
```

vec_main, a special function that takes no arguments, is the entry point of your program and the first code that will be executed. The printf function is identical to the C implementation included in stdio.h. The vec_main function should return 0 for non-exceptional conditions.

To compile the above code, save the file as hello.vec and run the command

```
generator < hello.vec > hello.cu
```

The generator reads the Vector code and outputs CUDA code. (Input is on stdin and output is on stdout). The output of the compiler looks like the following:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <libvector.hpp>

 int32_t vec_main(void) {
printf("Hello, World!\n");
return 0;
}

int main(void) { return vec_main(); }
```

This should give you a basic idea of how Vector code maps to CUDA code.

To compile the CUDA code to binary, please consult the CUDA manual for instructions on compiling this CUDA for your specific architecture. Be sure that libvector.hpp is in your include path.

Test the program and make sure it prints "Hello, World!". Congratulations, you have written your first Vector program. The following sections will provide more information on the building blocks of more complex Vector programs.

2.4 Variables and Types

Variables can be considered named areas of memory that hold values that can be read and written. Only primitive types are supported; primitives in Vector (which include the various floating point and integral types, in various sizes) can be declared as follows:

```
type_name variable_name;
```

Alternatively, assignment and declaration can be performed in a single statement (the assigning declaration) using the ':=' operator. For example,

```
a := 42;
```

declares a variable named a, and assigns 42 to it. This is exactly equivalent to

```
int a;
a = 42;
```

Note that with assigning declarations it is not necessary to explicitly declare the type of the variable. This will be inferred from the type of the right hand side. In this case the integral literal is assumed to have type 'int'. To give it another type, simply cast the right hand side to the desired type. For example,

```
a := float(42);
```

will declare a variable named a as a single precision floating point with a value equivalent to 42. For more information on the available types, the type system and type inference, please consult the Language Reference Manual.

2.5 Arrays

As a language designed primarily for parallel computation, Vector and Vector programs rely heavily on arrays. Arrays in Vector can be multidimensional. For example, to declare a 3 x 4 x 5 array of ints, use

```
int a[3, 4, 5];
```

To access an element of the array, a similar syntax is used. For instance,

```
a[i, j, k]
```

is equivalent to `a[i][j][k]` in C-like languages.

2.6 Operators

Operators in Vector are the same as those in C and similar languages. We support the boolean operators (e.g. comparison), arithmetic operators, and assignment operators. For more information, please consult the Language Reference Manual.

A subtle difference in Vector compared with other C-like languages is that for binary operators, both operands must have the same type, or be cast to the same type – no automatic promotion of variables is supported.

2.7 Control Flow

The Vector language supports standard conditional statements and looping constructs similar to counterparts in C and Java. The if-statement and while-statement are identical:

```
//else is optional
if ( boolean_condition ) {
    // some code
}
else {
    //more code
}

//while statement
while ( boolean_condition ) {
    //some code
}
```

The for-statement is a looping construct more similar to that found in Python. It loops over iterators, which can either be ranges or array elements. A single for loop can loop over multiple iterators simultaneously.

Ranges are specified in the format a : b : c, where a is the initial value, b is the upper bound (non-inclusive), and c is the amount by which we increment on each iteration. c can be omitted, or it can be negative. a, b, and c need not be integer literals; they may also be expressions. Also, using multiple iterators separated by a comma will produce code equivalent to two nested loops. Example:

```
for (i in 0:5:2, j in 0:4) {
    // some code
}
```

is equivalent to

```
for (int i = 0; i < 5; i += 2) {
    for (int j = 0; j < 4; j++) {
        // some code
    }
}
```

in C++.

We also support iterating over arrays, with the basic syntax (where arr is the name of an array):

```
for (x in arr) {
    // some code
}
```

Iteration over multidimensional arrays uses the array as if it were single dimensional. That is, for a two dimensional array, we would iterate over every element in the first row, then every element in the second row, etc.

Range iterators and array iterators can be combined in the same for statement, with the same "nested" effect as described above. This covers most typical use cases of the for statement in other languages; if finer-grained control is required, a while loop can be used instead.

2.8 Functions

Function declarations and definitions in Vector are identical to those in C, and can only happen in the global scope. The format for a declaration-definition is, for example:

```
int f(int a, int b) {
    return a + b;
}
```

Which declares and defines a function f which adds two integers and returns the resulting integers. To call the function, use:

```
 result := f(2,4);
```

Notice that in the above declaration, the type of result is inferred from the return type of f. As in C, functions can have type void if they perform operations but return no result.

2.9 Higher Order Functions and pfor

The coolest parts about Vector are the parallel processing components which abstract away some of the details that may have confused users when programming in CUDA. Three of the most heavily used programming patterns for GPU processing, (map, reduce, parallel for) have been implemented in Vector.

2.9.1 map Higher-order functions `map` and `reduce` in Vector are called with a `@` character function at the beginning, but besides designation appear and are called like normal functions. `map` takes as an argument a function of a single argument, and an array. The function that gets passed to `map` must be designated with the `__device__` directive, as this function is run on the GPU. The map operation then applies the function passed in to every element of the array, and returns a new array with the results. Note that the argument type of the function passed must be equivalent to the type of the elements of the array, but the return type can be different.

Here's an example:

```
__device__ float square(float x) {
 return x * x;
}

int[] another_function(int inputs[]) {
  squares := @map(square, inputs);
  return squares;
}
```

In this example `another_function` returns an array of the squares of the elements of the input array.

2.9.2 reduce The higher-order function `reduce` takes as arguments a function that takes two arguments and an array. Reduce then applies the function to combine the elements of the array. It does this by first applying the function to pairs of elements of the array, obtaining a new array containing half the number of the values of the original, and then by performing the same operation on this new array, and continuing until a single value has been obtained.

The return value of the function passed to `reduce` must be of the same type as the elements of the input array, and the types of the arguments of the function passed to `reduce` must also be of the same type. Here's an example:

```
__device__ int add(int x, int y) {
  return x + y;
}

int another_function(int inputs[]) {
  sum := @reduce(add, inputs);
  return sum;
}
```

In this example, `another_function` returns the sum of the input array.

2.9.3 pfor Although not exactly a higher order function like map and reduce, pfor is another feature of Vector that abstracts away some of the complexities in GPU programming. It works exactly the same as the regular for statement except that the computation will be performed on the GPU - hence, the syntax remains identitcal to the regular for. Here is a simple example of the pfor:

```
void pfor_example()
{
    int arr[1000, 2];

    pfor (i in 0:len(arr, 0), j in 0:len(arr, 1))
        arr[i, j] = 2 * i + j;
}
```

3. Vector Language Manual

3.1 Lexical Conventions

3.1.1 Comments The characters `/*` introduce a comment, and the first `*/` ends the comment. Single-line comments are also supported and denoted with `//` at the beginning of a line.

3.1.2 Identifiers An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore `_` counts as alphabetic. Identifiers are case-sensitive.

3.1.3 Keywords The following identifiers are reserved for language keywords

```
bool char int8 byte uint8 int16 uint16 int int32 uint uint32 int64 uint64 double float float32
double float64 complex complex64 complex128 string if else while for pfor return
```

3.1.4 Constants Vector has the following constants:

3.1.4.1 Integer Constants An integer constant is a sequence of digits.

3.1.4.2 Character Constants A character constant is a single character enclosed in single quotes `' '`. Single quotes must be preceded by a backslash `\`. The `\` character, along with some non-graphic characters, can be escaped according to the following rules:

- Backspace `\b`
- Newline `\n`
- Carriage Return `\r`
- Tab `\t`
- `\ \\`

Character constants behave like integers. Characters are stored in two bytes, with the integer code for the character stored in the lower-order byte and 0 in the higher-order byte. For characters of length 2, for example if an escaped character is used, the integer code for the first character is stored in the lower-order byte and the integer code for the second character is stored in the higher-order byte.

3.1.4.3 String Constants String constants consist of a series of characters delimited by quotation marks `""`.

3.1.4.4 Floating Constants Floating constants consist of an integer part, a decimal point, a fraction part, an `e` and a signed exponent. If decimal point is not included, then the `e` and signed exponent must be included, otherwise, they are optional.

3.2 Syntax Notation

In this manual, a `typewriter` typeface indicates literal words and characters. An *italic* typeface indicates a category with special meaning. Lists are presented either inline or using bullets. If two items are presented on same line of a bulleted list separated by commas, they are equivalent. <epsilon> is used to indicate the empty string. Backus-Naur Form is used to express the grammar of Vector.

3.3 Types

type-specifier ::=

primitive-type | *array-type* | *function-type* | `void`

Each identifier is associated with a type that determines how it is interpreted. A void type has no value.

3.3.1 Primitive Types

primitive-type ::= *integer-type* | *floating-point-type* | *complex-type*

Vector supports three categories of primitive types: integers, floating point numbers, and complex numbers.

3.3.1.1 Integer Types Integer types are given by the following literals:

- `bool`, `char`, `int8`
- `byte`, `uint8`
- `int16`
- `uint16`
- `int`, `int32`
- `uint`, `uint32`
- `int64`
- `uint64`

The types starting with `u` are unsigned types. The number at the end of a type name indicates the size of that type and equivalent types in bits.

3.3.1.2 Floating Point Types Floating-point types are given by the following literals:

- `float`, `float32`
- `double`, `float64`

These two types correspond to IEEE single-precision and double-precision floating point numbers, respectively, as defined in IEEE 754.

3.3.1.3 Complex Number Types Complex number types are given by the following literals:

- `complex`, `complex64`
- `complex128`

These two complex types are constructed from two `float32` or two `float64` types, respectively. The real and imaginary parts of the numbers can be accessed or assigned by appending `.re` or `.im` to the identifier.

```
a := ##(3.1, 2.1)
b := a.re // b is 3.1
a.im = 1.2 // a is now ##(3.1, 1.2)
```

3.3.1.4 String Types

String types are given by the following literal:

- `string`

3.3.2 Array Types

array-type ::= *primitive-type* `[]`

Arrays are composed of multiple instances of primitive types laid out side-by-side in memory.

Arrays are a very important part of the Vector language, as they are the only type that can be modified on both the CPU and GPU. Allocation of arrays on the GPU and transfer of data between CPU and GPU are handled automatically.

Array elements are accessed using square-bracket notation. For instance `a[4]` returns the element at index 4 of array `a` (arrays are zero-indexed). The built-in `len` function returns an `int` representing the length of an array.

Array elements can be assigned to using the same syntax, so `a[4] = 3` will set the value at index 4 of the array to 3.

Arrays can also be multi-dimensional. Indexing into a multi-dimensional array is achieved by separating the dimensional index numbers by commas. So `a[1, 2]` will access row 1, column 2 of the two-dimensional array `a`.

Arrays can be initialized using comma-seperated list delimited by curly braces `{}`.

3.3.3 Function Types

Functions take in zero or more variables of primitive or array types and optionally return a variable of primitive or array type.

3.4 Objects and LValues

An object is a named region in memory whose value can be read and modified. An LValue is an expression referring to an object. It has a value, and a corresponding region in memory where the value is stored.

An expression is an RValue if it only has a value, but not a corresponding region in memory, i.e., it cannot have a new value assigned to it. In the following code:

```
a := 4
b := 5
a = b
```

a and b are LValues, because they have a value that can be assigned. 4 and 5 are RValues, because they cannot have new values assigned to them.

LValues are named because they can appear on the left side of an assignment (or also on the right), whereas RValues can appear only on the right side. All expressions in Vector are either RValues or LValues.

3.5 Conversions

Any scalar type can be converted to another scalar type. For any conversion from a narrower to a wider type, for example from a `int16` to `int32`, or from `float` to `double`, the conversion can be done with no loss of precision. If a `double` is converted to a `float`, then the `double` will be rounded and then truncated to be the length of the `float` type that it is converted to. For conversions between signed types and unsigned types,

if the signed type had a negative number, then `UINT_MAX + 1`, where `UINT_MAX` is the maximum unsigned integer of the target type, will be added to it so that the value is a valid unsigned type.

For conversions between any integer and floating-point type, everything up to 16 bits can be cast to `float` with no loss of precision, and everything up to 32 bits can be cast to `double` with no loss of precision. For any floating-point to integer type conversions, the fraction portion of the floating-point type is discarded.

When a `char` object is converted to an int, its sign is propagated through the upper 8 bits of the resulting int object.

explicit-cast ::= *primitive-type-specifier* `(` *identifier* `)`

3.6 Expressions

expression ::=

primary-expression

postfix-expression

unary-expression

cast-expression

multiplicative-expression

additive-expression

shift-expression

relational-expression

equality-expression

logical-bitwise-expression

logical-expression

assignment

compound-assignment

function-call

higher-order-function-call

3.6.1 Primary Expressions Primary expressions consist of constants, identifiers, or expressions in parentheses.

primary-expression ::=

identifier

constant

(*expression*)

3.6.2 Postfix Expressions The operators in postfix expressions group left to right.

postfix-expression ::=

expression++

expression--

3.6.3 Operators

3.6.3.1 Unary Operators

unary-expression ::=

unary-operator expression

unary operators include -, !, and ~. The - unary operator returns the negative of its operand. If necessary, the operand is promoted to a wider type. For unsigned quantities, the negative is computed by subtracting the promoted value from the largest value of the promoted type and adding 1. For the case that the value is 0, the value returned is also 0. The ! operator returns 1 if its operand is 0 and returns 0 otherwise. The ~ operator returns the one's complement of its operand.

3.6.3.2 Multiplicative Operators Multiplicative operators include *, /, and %, and group to the right. There are two operands and both must have arithmetic types.

multiplicative-expression ::=

expression * *expression*

expression / *expression*

expression % *expression*

The * operator denotes multiplication, the / operation gives the quotient, and the % operator gives the remainder after a division of the two operands.

3.6.3.3 Additive Operators The additive operators include `+` and `-` group, and they group left-to-right. If the operands have arithmetic types, then the appropriate arithmetic operation is performed.

additive-expression ::=

expression `+` *multiplicative-expression*

expression `-` *multiplicative-expression*

The `+` operator gives the sum of the two operands, and the `-` operator gives the difference.

3.6.3.4 Shift Operators The shift operators include << and >>. These operators group left to right, and each operator must be of an integral type.

shift-expression ::=

expression `<<` *expression*

expression `>>` *expression*

The value of shift expression *E1* << *E2* is interpreted as *E1* left-shifted by *E2* bits, and *E1* >> *E2* is interpreted as *E1* right-shifted *E2* bits.

3.6.3.5 Relational Operators Relational operators group left-to-right.

relational-expression ::=

expression `>` *expression*

expression `<` *expression*

expression `<=` *expression*

expression `>=` *expression*

The operator `>` denotes the greater-than operation, `<` denotes less-than, `>=` denotes greater-than-or-equal, and `<=` denotes less-then-or-equal.

Each of these operators returns 0 if false and 1 if true, and this result is always of type `int`.

3.6.3.6 Equality Operators

equality-expression :==

expression `==` *expression*

expression `!=` *expression*

The equality operator `=` denotes equal-to, and the operator `!=` denotes not-equal-to. These both return 1 if true and 0 if false, and this value is of type `int`. These operators have lower precedence than relational operators.

3.6.3.7 Bitwise Logical Expressions

logical-bitwise-expression ::=

expression `&` *expression*

expression `^` *expression*

expression `|` *expression*

The `&` operator denotes the bitwise-and operation. The result is the bitwise-and function applied to the operands.

The `^` operator denotes the bitwise-exclusive-or operation. The result is the bitwise-exclusive-or operation applied to the two operands.

The `|` operator denotes the bitwise-inclusive-or operation. The result is the bitwise-inclusive-or operation applied to the two operands.

These operations require both operands to have integral types.

3.6.3.8 Short-Circuit Logical Expressions

logical-expression ::=

expression `&&` *expression*

expression `||` *expression*

The `&&` operator returns 1 if both operands are not equal to 0, and 0 if at least one operand is equal to 0. The `||` operator returns 1 if at least one operand is not equal to 0, and 0 otherwise.

These operators group expressions left-to-right. Operands must be of arithmetic types, and the return value is `int`.

These operators are short circuiting. For `&&`, if the left operand evaluates to 0, the entire expression evaluates to 0 and the right operand is not evaluated. For `||`, if the left operand evaluates to something other than 0, the expression returns 1 and the right operand is not evaluated.

3.6.4 Operator Precedence and Associativity For binary operators, it is necessary to specify operator precedence and associativity in order to avoid ambiguity. The precedence of operators in vector from highest to lowest precedence is as follows.

- `*`, `/`, `%`
- `+`, `-`
- `<<`, `>>`
- `<`, `<=`, `>`, `>=`
- `==`, `!=`
- `&`
- `^`
- `|`
- `&&`
- `||`
- `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `|=`, `&=`, `^=`

All of these operators are associated left-to-right, with the exception of the last row (the assignment operators) which are associated right-to-left.

3.6.5 Function Calls

function-call ::= *identifier* `(` *argument-list* `)` | *identifier* `()`

argument-list ::= *argument-list* `,` *expression* | *expression*

The type of the identifier must be a function. When a function call is encountered, each of the expressions in its argument list (if it has one) is evaluated (with side effects); the order of evaluation is unspecified. Then, control of execution is given to the function specified by the identifier, with the a copy of the result of each of the expressions available in the scope of the function block as *parameters*.

The types of the each of the expressions in the argument list must match exactly the types of the parameters of the function.

All argument passing is done by-value; that is, a copy of each argument is made before the function has access to it as a parameter. A function may change the value of its parameters without affecting the value of the arguments in the calling function.

However, if an array type is passed to a function, the array is not copied, but merely the reference to the array. Therefore, any modifications the function makes to the array affect the value of the array in the calling context.

A function may call itself.

The result of evaluating the function call is the value returned by the function called.

Vector provides the following builtin functions

- `len(arr[, dim])` - Returns the length of an array. By default, it returns the total size of the array. If an optional second parameter is passed in, `len` returns the length along the given dimension of an array. Therefore, for a 2D array `arr` with 3 rows and 2 columns, `len(arr) == 6`, `len(arr, 0) == 3`, and `len(arr, 1) == 2`.
- `random()` - Generates a random integer.
- `printf(string_lit[, ...])` - prints to standard output, formatting is identical to C's printf function
- `inline(cuda_code)` - takes a string literal of cuda code and passes it verbatim to the CUDA output
- `assert(condition)` - checks if a condition is true and exits with error if it is not
- `abs(number)` - takes the absolute value of scalar numeric arguments and the magnitude of complex arguments
- `time()` - returns the current system time as a float64 representing the fractional number of seconds since the beginning of the epoch

3.6.6 Higher-Order Functions

higher-order-function-call ::=

`@` *identifier* `(` *identifier* `,` *argument-list* `)`

Vector supports a set of builtin higher order functions. Since these functions require compile-time support, they have a slightly modified syntax to distinguish them from normal functions. Higher-order functions start with an at sign and take a function identifier as the first argument. Currently-supported higher-order functions are `@map` and `@reduce`. Map takes a function of a single argument and an array and returns an array resulting from applying the function to each element of the array. This mapping operation is performed on the GPU. For instance,

```
int evenorodd(int num)
{
    return num % 2;
}

parities := @map(evenorodd, {1, 2, 3, 4, 5});
```

The `parities` variable will get the value `{1, 0, 1, 0, 1}`, where each element is 1 if the corresponding input element was odd of 0 if the input element was even.

The reduce HOF takes a function of two variables and an array and returns the resulting scalar value from applying the function to pairs of the inputs. For instance,

```
int add(int a, int b)
{
    return a + b;
}

sum := @reduce(add, {1, 2, 3, 4, 5});
```

This code takes the sum of the array. There is no guarantee on the order in which items in the array are reduced. Therefore, the reducing function must be associative and commutative or the result with be non-deterministic.

3.6.7 Assignment

assignment ::= *identifier* `=` *expression*

An assignment is simply an identifier and an expression separated by an equals sign. An assignment is itself an expression returning the value that was assigned, so they can be chained. For instance

```
a = b = 3;
```

Assigns the variables a and b to the value 3.

3.6.8 Compound Assignment

compound-assignment ::=

identifier `+=` *expression*

identifier `-=` *expression*

identifier `*=` *expression*

identifier `/=` *expression*

identifier `%=` *expression*

identifier `<<=` *expression*

identifier `>>=` *expression*

identifier `|=` *expression*

identifier `&=` *expression*

identifier `^=` *expression*

Compound assignments perform an operation with the identifier and expression given as operands and then assign the variable specified by the identifier to the result. For example `a += 5` is equivalent to `a = a + 5`.

3.7 Declarations

declaration ::=

primitive-declaration

array-declaration

function-declaration

A declaration specifies the type of an identifier; it may or may not allocate memory for the identifier.

3.7.1 Primitive Type Declarations

primitive-declaration ::=

primitive-type-specifier identifier `;`

identifier `:=` *expression* `;`

The first primitive declaration declares a primitive type variable unintialized. In this case, the value of the identifier before first assignment is unspecified.

The second declaration declares a primitive variable with the given identifier with its initial value set to the result of the expression. The type of the identifier will be inferenced from the expression. If you wish to specify the exact type of the identifier, use an explicit cast.

3.7.2 Array Declarations

array-declaration ::= *primitive-type-specifier identifier* `[];`

primitive-type-specifier identifier `[` *index-list* `];`

identifier `:=` *expression* `;`

index-list ::= *index-list* `,` *expression* | *expression*

The first syntax does not initialize the array or allocate any storage for it.

The second syntax declares an array and allocates storage but does not initialize its members. The expression is evaluated (with side effects) and the result is the number of members the array will have (and the size of the array is the size of the primitive type multiplied by the number of members). The type of the expression must be an unsigned integer.

The third syntax inferences the type of the array from the expression and is identical to the initializing declaration for primitives.

3.7.3 Function Declarations

function-declaration ::=

type-specifier identifier (*parameter-list*) *compound-statement*

param-list ::=

non-empty-param-list | <epsilon>

non-empty-param-list ::=

param , *non-empty-param-list* | *param*

A function declaration declares a function that accepts the parameters given by the parameter list and, when called, evaluates the given block (also known as a *function body*). A function may not be modified after declaration.

The parameter list is a series of primitive or array declarations separated by commas. Only the non-initializing primitive declarations and non-sizing array declarations are allowed. The identifiers specified by the parameter list are available in the function body.

3.7.4 Forward Declarations

forward-declaration ::= *type-specifier identifier* (*parameter-list*);

A forward declaration is the same as a function declaration except instead of having a compound statement implementing the function, it is terminated by a semicolon. Forward declarations are mainly used for declaring functions that are implemented in C.

3.8 Statements

statement ::= *expression-statement*

declaration

compound-statement

selection-statement

iteration-statement

jump-statement

Statements in Vector are executed in sequence except as described as part of compound statements, selection statements, iteration statements, and jump statements.

3.8.1 Expression Statements

expression-statement ::= *expression* `;`

An expression statement is an expression with its value discarded followed by a semicolon. The side effects of the expression still occur.

3.8.2 Declarations Declarations are also considered statements. The only caveat is that nested function declarations are not allowed.

3.8.3 Compound Statements

compound-statement ::= `{` *statement-list* `}`

statement-list ::= *statement-list statement*

<epsilon>

A compound statement is also called a *block*. When a block is executed, each of the statements in its statement list are executed in order. Blocks allow the grouping of multiple statements, especially where only one is expected. In addition, scoping is based on blocks; see the "Scope" section for more details.

3.8.4 Selection Statements

selection-statement ::=

`if (` *expression* `)` *statement* `else` *statement*

`if (` *expression* `)` *statement*

When a selection statement is executed, first the expression associated with the `if` is evaluated (with all side effects). If the resulting value is nonzero, the first substatement is executed. and control flow resumes after the selection statement. If the resulting value is zero and there is an `else` clause, the substatement of the `else` clause gets executed, and control resumes after the selection statement. If there is no `else` clause, control flow resumes after the selection statement without executing either of the substatements.

If statements can be nested (as in `else if`). If there is ambiguity in which `if` an `else` corresponds to, the ambiguity is always resolved to the closest non-matched `if`.

3.8.5 Iteration Statements

iteration-statement ::= `while` *expression statement*

`for (` *iterator-list* `)` *statement*

`pfor (` *iterator-list* `)` *statement*

iterator-list ::= *iterator-list* `,` *iterator* | *iterator*

iterator ::= *identifier* `in` (*array-expression* | *range*)

range ::=

expression : *expression* : *expression*

expression : *expression*

: *expression* : *expression*

: *expression*

When a while statement is reached, the expression is evaluated (with all side effects). If its value is nonzero, its block is executed, and after the execution of the block, the while statement is executed again. If the value of the expression is zero, the execution of the while statement is finished.

A for statement allows you to sweep one or more variables across an array or range, evaluating the inner statement with the identifiers assigned to a new set of values each time.

The expression following the `in` in an iterator expression can be an array or a range. If it is an array, the identifier is assigned to the value of each element of the array sequentially. If it is a range, the identifier is assigned to each integer in the range.

A range consists of three integers separated by colons. The integers represent the start of the range, the exclusive end of the range, and the step size of the range. So, for instance, the range `0:5:2` will generate the sequence `0, 2, 4`. The first integer can be ommitted, in which case it will default to zero. The third integer can also be ommitted, in which case it will default to one. If the third integer is ommitted, the second colon is ommitted as well.

The step size can also be negative, in which case the iteration will proceed in reverse order. Therefore, `5:0:-1` will generate the sequence `5, 4, 3, 2, 1`.

If multiple iterators are given in the for statement, the rightmost iterator will go to completion before the iterator to the left is advanced. For instance, the statement

```
for (i in 0:2, j in 0:2) { printf("%d, %d\n", i, j); }
```

Will result in the output

```
0, 0
0, 1
1, 0
1, 1
```

A pfor statement is identical to the for statement, except the iterations of the for statement all happen in parallel on the GPU.

3.8.6 Jump Statements

jump-statement ::= `return` *expression* `;` | `return ;`

A return statement returns control of execution to the caller of the current function. If the statement has an expression, the expression is evaluated (with side effects) and the result is returned to the caller.

3.9 Scope

Vector uses block-level scoping. A block is another name for a compound statement (see "Compound Statements" section). Most frequently, a block is a section of code contained by a function, conditional, or looping construct. Each nested block creates a new scope, and variables declared in the new scope supersede variables declared in higher scopes.

3.10 Undefined behavior

The following things are allowed by the vector compiler, but will produce code that will either not be recognized by the CUDA compiler or fail at runtime.

- Recursion in device functions
- Accessing global variables in device functions
- Manipulating strings in device functions or in the bodies of pfor statements
- Assigning to variables declared outside of a pfor statement inside the pfor statement

3.11 Grammar

top-level ::=

top-level-statement top-hevel

top-level-statement

top-level-statement ::=

datatype identifier `(` *param-list* `)` `{` *statment-seq* `}`

datatype identifier `(` *param-list* `);`

declaration

include-statement

statement-seq ::=

statement statement-seq

<epsilon>

datatype ::=

`int` | `char` | `float` | `bool` | `char` | `int8` | `byte` | `uint8`

`int16` | `uint16` | `int` | `int32` | `uint` | `uint32` | `int64`

`uint64` | `double` | `float` | `float32` | `double` | `float64`

`complex` | `complex64` | `complex128` | `string`

expression ::=

expression `+` *expression*

expression `-` *expression*

expression `*` *expression*

expression `/` *expression*

expression `%` *expression*

expression `<<` *expression*

expression `>>` *expression*

expression `<` *expression*

expression `<=` *expression*

expression `>` *expression*

expression `>=` *expression*

expression `==` *expression*

expression `!=` *expression*

expression `&` *expression*

expression `^` *expression*

expression `|` *expression*

expression `||` *expression*

expression `&&` *expression*

lvalue `+=` *expression*

lvalue `-=` *expression*

lvalue `*=` *expression*

lvalue `/=` *expression*

lvalue `%=` *expression*

lvalue `<<=` *expression*

lvalue `>>=` *expression*

lvalue `|=` *expression*

lvalue `&=` *expression*

lvalue `^=` *expression*

`-`*expression*

`!`*expression*

`~`*expression*

expression`++`

expression`--`

`(`*expression*`)`

lvalue `=` *expression*

lvalue

expression `[`*expression-list*`]`

expression `[`*expression-list*`]` `=` *expression*

constant

datatype `(`*expression*`)`

`{`*expression-list*`}`

identifier `()`

identifier `(`*expression-list*`)`

`@`*identifier* `(`*identifier*`,`*expression-list*`)`

lvalue ::=

identifier

expression `[`*expression-list*`]`

expression-list ::=

expression `,` *expression-list*

expression

declaration ::=

identifier `:=` *expression*`;`

datatype identifier `;`

datatype identifier `[];`

datatype identifier `[`*expression-list*`];`

statement ::=

`if (`*expression*`)` *statement* `else` *statement*

`if (`*expression*`)`*statement*

`while (` *expression* `)` *statement*

`for (` *iterator-list* `)` *statement*

`pfor (` *iterator-list* `)` *statement*

`{` *statement-seq* `}`

expression `;`

`;`

declaration

`return` *expression* `;`

`return;`

iterator-list ::=

iterator `,` *iterator-list*

iterator

iterator ::=

identifier `in` *range*

identifier `in` *expression*

range ::=

expression `:` *expression* `:` *expression*

expression `:` *expression*

`:` *expression* `:` *expression*

`:` *expression*

param ::=

datatype identifier

datatype identifier `[]`

non-empty-param-list ::=

param `,` *non-empty-param-list*

param

param-list ::=

non-empty-param-list

<epsilon>

constant ::=

int

int64

float

complex

string

char

4. Project Plan

4.1 Planning Process

The group planned the project iteratively and stuck to a feature-driven development rather than a component-driven process. That is, for each step of the process, the group discussed the desired (yet feasible at that point in time) feature requirements during the weekly meetings. Although mostly iterative, the planning process also included some initial planning from the beginning. Once the language reference manual was created, and the full scope of the project was sketched out, the team set major long-term milestones and an initial feature set with flexible deadlines. The plan was assessed and altered depending on progress reports in weekly group meetings as well as meetings with Professor Edwards.

4.2 Specification Process

As mentioned in the previous section, the initial feature set and specifications were planned as soon as the first draft of the language reference manual was written. While the group prioritized the initial feature set, additional features mentioned during meetings were kept in mind and committed to as time allowed.

4.3 Development Process

Although the organization of a compiler is such that there is some sequence in implementing components is required due to dependencies, the development process was largely shaped by the feature-driven planning process. That is, since the weekly commitments depended on feature goals that the group agreed upon, each member was not restricted to working on any one part. This resulted in members taking charge of the specific feature assigned, writing not only the compiler code but also tests, if necessary. Each feature was often developed in pairs or in smaller groups, depending on its difficulty and time commitment.

4.4 Testing Process

As mentioned in the development process description, each member contributed to testing, since the the test for a given feature was written by the member who implemented it. With each new feature development, additional tests were written and previous tests were run to check that the modifications kept all other parts intact.

4.5 Style Guide

Ocaml coding style

- Indentations should be four spaces
- Exception to rule above is pattern matchings, which should be two spaces, followed by the pipe character, another space, and then the pattern
- For `match` statements, the first pattern should begin with a pipe character. This rule also holds for other statements that use the pipe, such as `type`. The rule can be ignored if the statement is all on one line.
- No trailing whitespace at the end of lines
- Lines should not be much longer that 80 characters

C++ coding style

- Indentations should be tabs (hard tabs). Tabs are taken to be equivalent in width to 8 spaces.
- The opening curly brace of a compound statement (like `if` or `for`), should be at the end of the first line. The closing brace should go on a separate line.
- Braces in function declarations are different. Opening brace should be on separate line.
- Lines should not be much longer than 80 characters

4.6 Project Timeline

Date	Milestone
Sep. 15	Project proposal drafted
Oct. 07	Lexer completed
Oct. 25	Parser completed
Oct. 31	Language reference manual drafted
Nov. 05	Semantic checking and runtime library completed
Dec. 04	Code generation completed
Dec. 18	Final report completed

4.7 Team Responsibilities

Team Member	Responsibilities
Howard Mao (team leader)	Compiler Frontend, Runtime Library, Code Generation
Zachary Newman	Compiler Frontend, Code Generation, Test Suite Creation
Sidharth Shankar	Compiler Frontend, Code Generation, Semantic Checking
Jonathan Yu	Code Generation, Runtime Library
Harry Lee	Code Generation, Documentation

4.8 Development Environment

Because Vector targets the GPU, a uniform development environment is critical.

We use:

- Git for version control and host a shared repository on GitHub. This includes our source code, test suite, virtual environment specification, and documentation.
- Vagrant to mangage virtual development environments. We specify in a `Vagrantfile` that we will develop on an Ubuntu 12.04 virtual machine, and provision using a shell script to install dependencies.
- SCons as a build tool. Its Python syntax allows more flexibility than make. We build our compiler (including the parser and scanner) and run our test suite with SCons.
- OCaml (as well as `ocamllex` and `ocamlyacc`) to write our compiler.

- nvcc to compile generated CUDA code.
- gpuocelot to run PTX bytecode on the virtual x86 machines and other hardware without NVidia GPUs.

The developer only has to deal with installing Git and Vagrant; our build system abstracts the details of nvcc and gpuocelot away.

4.9 Project Log

Date	Milestone
Sep. 05	First team meeting and preliminary planning
Sep. 08	Language defined
Sep. 10	Development environment defined
Sep. 15	Project proposal drafted
Sep. 20	Language features defined
Oct. 06	Operations in grammar implemented
Oct. 07	Lexer completed
Oct. 09	Expressions in grammar implemented
Oct. 14	Control-flow statements in grammer implemented
Oct. 15	Regression test suite set up
Oct. 19	Statements in grammer implemented
Oct. 24	Basic code generator implemented
Oct. 25	Parser completed
Oct. 29	Scoping and symbol checking implemented for code generation
Oct. 30	Type inference implemented
Oct. 31	Language reference manual drafted
Nov. 05	Semantic checking and runtime library completed
Nov. 08	Higher order function ‘map’ implemented for code generation
Nov. 10	For loop and control flow implemented for code generation
Nov. 18	Higher order function ‘reduce’ implemented for code generation
Nov. 20	Inline macros implemented for code generation
Dec. 03	pfor statements implemented for code generation
Dec. 04	Code Generation completed

5. Architecture and Design

5.1 Architecture

The main components of the Vector compiler are the scanner, parser, code generator, and runtime library. We used the Ocaml libraries `ocammlex` and `ocamlyacc` to build our scanner and parser respectively. The code generator is also built in Ocaml and is implemented in the file `generator.ml`. The main entry point to the generator is a function called `generate_toplevel`, which takes in the abstract syntax tree returned by our parser as an argument and returns the generated CUDA code as a string, and a record called `environment`, which stores important global state about our program.

`generate_toplevel` steps through the AST of our program, and for each object in the AST it encounters, we call a corresponding generator function, which returns the CUDA code for that object in the AST. After a single pass through the AST, the compiler has generated the vector code for the input program.

However, not all the CUDA code can be generated in the same order that the corresponding objects appear in the AST. For example, some of the functions require forward declarations that need to appear at the beginning of the compiled CUDA file. To handle these, we do deferred generation for some of the generated code. As we encounter functions for which generation needs to be deferred, we add necessary information about that function to our `environment` record.

More information about the `environment` record and deferred generation follow in the next two sections.

5.2 Environment and Deferred Generation

We need to store global state in our compiler for the following reasons:

- Scope information
- Deferred Generation of functions

Information about scope is essential because the scope that we are in in the AST has impact on the validity of programs. For example, we we are declaring a variable `x`, but a variable `x` has already been declared in the same scope, this is an error, as variables can only be declared once.

The other reason we need global state is because not all of the code that our compiler generates is inline–for example, in our generated code we need to begin a file with the forward declarations of all the functions that will run on the GPU. Since we need to add these to the beginning of a file, we simply save these as global state as we collect more, and then when our compiler has finished a single pass through the AST of our program outputs the forward declarations before the generated code from our AST.

In order to keep track of this global state, we defined a record `environment` in Ocaml with the following fields:

- `kernel_invocation_functions`
- `kernel_functions`
- `func_type_map`
- `scope_stack`
- `pfor_kernels`
- `on_gpu`

In CUDA, the way we run functions on the GPU is through functions called kernels. Kernels are special functions, specified with the directive `__global__` and have special requirements. Kernels are invoked from code running on the CPU through specifying the number of blocks and threads we need to run for that function, and the function name. A typical kernel invocation looks like this:

```
kernel<<<grid_dim, block_dim>>>(*a, *b, size);  // *)
```

Where `a` and `b` are both pointers to arrays and size is the size of that array.

When we compile a Vector program, when high-level functions `@map` and `@reduce` and called, we need to invoke a kernel, and in addition, need to compile the function for the GPU as well.

Therefore, we need to generate two functions for each invocation of a high-level functions, one for the kernel itself, and one for the invocation of the kernel. We add information we need to generate the kernel invocation to the `kernel_invocation_list`, and the information the compiler needs to generate the GPU version of the function to the `kernel_functions` list in the environment. We the defer the generations of these functions because high-level functions can be called in other functions, and nested function definition is not permitted in CUDA.

To ensure that kernel functions are defined when the kernel invocation functions are compiled, we add forward declarations to our generated CUDA code for kernel functions.

`func_type_map` is a map that maps function identifiers to a tuple containing information about whether that file is meant to be compiled for the GPU, the return type of the function, and a list containing the types of its arguments. We use this for type inference, checking if a function call is valid, and for forward declarations.

`scope_stack` is a stack containing a list of variables that have been declared in the current scope. We push and pop from this stack as change scopes as we walk through the AST, and use this to check if variable declarations are valid.

`pfor_kernels` is a list of kernel functions, each of which corresponds to a `pfor` loop in the vector code. For each `pfor` loop encountered in our program, we need to generate a kernel and invoke it. Since a `pfor` loop could be called within another function, we need to defer the generation of this kernel, and therefore add the information that we need to the GPU.

`on_gpu` is a boolean flag that checks to see whether we should be compiling code for the GPU or for the CPU. It gets set to true when we start compiling functions for the GPU, and makes subtle changes, for example using our runtime library functions for array access in GPUs instead of normal array access.

5.3 Runtime Library

Vector's runtime library is implemented in C++ and contains supporting classes and functions that are called by the CUDA code generated by the Vector compiler. The runtime library is provided as a set of C++ headers.

The most important part of the runtime library is the VectorArray class. This is a templated C++ class that holds the information needed to provide an N-dimensional array on both the CPU and the GPU. The VectorArray class stores the number of dimensions, a pointer to a heap array containing the size along each dimension, a pointer to the elements of the array on the CPU, and a pointer to the elements on the GPU. VectorArrays are reference-counted, so when an array is assigned from one variable to another in Vector, the actual array elements are not copied, but rather shared between the two arrays. The array elements will be freed when all VectorArrays referencing it are destroyed. The VectorArray class also contains two boolean flags called h_dirty and d_dirty, which indicate whether or the array data has been modified on the CPU or the GPU, respectively. These flags will be set when vector code assigns to an element of an array. If the h_dirty flag is set, the next time an array element is read on the GPU, the contents of the CPU array will be copied to the GPU and the h_dirty flag will be cleared. The converse will occur if an array with d_dirty set is read on the CPU. This allows Vector code to synchronize the contents of an array on the CPU and GPU without requiring the user to specify such copying explicitly.

The vector runtime library also contains a range_iter struct and supporting functions which help simplify the code generated for for loops and pfor loops. Each range_iter struct corresponds to a single loop iterator. For instance, in the following two-variable for loop,

```
for (i in 0:10, j in 0:6)
    // for loop body
```

The loop variables i and j are each represented in the generated CUDA code by a range_iter struct. This Vector for-loop then compiles to a single C++ for-loop which iterates from 0 to 59. The supporting function `get_index_cpu` is then called in the for loop body to obtain the values for i and j. Similarly, in the pfor-loop, the range_iter structs are copied to the GPU, the thread index is computed using the expression `threadIdx.x + blockIdx.x * blockDim.x`, and the values of the iterator variables are computed using the range_iter struct, the thread index, and the device function `get_index_gpu`.

Finally, the runtime library contains the struct device_info, a templated C struct that can be copied to the GPU to support N-dimensional array accesses in GPU code. The struct contains a pointer to the device data, an integer storing the number of dimensions, and a pointer to the lengths along each dimension. Pointers to device_info structs are passed into the kernels generated for pfor statements. When compiling GPU code (i.e. when the on_gpu flag in the environment is set), device_info structs take the place of VectorArray instances in the generated code for array manipulations.

6. Test Plan

The test suite for Vector consisted of simple regression tests for language features, as well as longer tests to demonstrate target programs in the language.

6.1 Rationale

We omitted unit tests, trusting the OCaml type checker to detect major bugs (such as missing cases in pattern matching). Edge cases were simple to write in Vector, so our regression test suite includes the sort of edge cases likely to be included in a unit test suite.

By running tests frequently and before each check-in, we could catch any backwards-incompatible changes during development. This system also allows test-driven development, as the test suite can be run with tests that use unimplemented language features, simply failing until those features are implemented.

6.2 Mechanism

6.2.1 Components The components of the "Project Plan" section relevant to the test plan are

- **Vagrant** to manage virtual development environments
- **SCons** as a build tool
- **nvcc** to compile generated CUDA code
- **gpuocelot** to run PTX bytecode on the virtual x86 machines

6.2.2 Implementation We created a `test` folder in the Vector source repository with an `SConscript` file implementing the build procedure. Within a Vagrant virtual machine, run `scons test` to

1. Using Vector, compile all test programs (with a `.vec` suffix) to CUDA files
2. Using nvcc, compile the generated CUDA sources to ELF executables linked with the gpuocelot library
3. Run the executables and compare the outputs to the expected-result files (with a `.out` suffix). Any files with differing output are considered to be failing tests.

Add a base filename to the `test_cases` list in the `SConscript` file to add a new test case. Then, add `.vec` and `.out` for the test case.

6.3 Representative Programs

An example of a non-trivial program in vector is calculation of the mandelbrot set.

```
__device__ int mandelbrot(int xi, int yi, int xn, int yn,
    float left, float right, float top, float bottom)
{
    iter := 0;

    x0 := left + (right - left) / float(xn) * float(xi);
    y0 := bottom + (top - bottom) / float(yn) * float(yi);
    z0 := #(x0, y0);
    z := #(float(0), float(0));
```

```
    while (iter < 256 && abs(z) < 2) {
        z = z * z + z0;
        iter++;
    }

    return iter;
}

int vec_main()
{
    img_height := 256;
    img_width := 384;

    int shades[img_height, img_width];

    left := float(-2.0);
    right := float(1.0);
    top := float(1.0);
    bottom := float(-1.0);


    pfor (yi in 0:img_height, xi in 0:img_width) {
        shades[yi, xi] = mandelbrot(xi, yi, img_width, img_height,
                            left, right, top, bottom);
    }

    return 0;
}
```

The program launches a pfor thread for each pixel of the image which computes the number of iterations til convergence for that point on the complex plane.

6.4 Tests Used

- **arrays.vec** ensures that arrays can both be written to and read from
- **complex.vec** ensures that our native support for complex numbers works correctly
- **dotprod.vec** performs the dot product of two vectors on the GPU
- **float.vec** ensures that floating point arithmetic works
- **functions.vec** ensures that calling functions works correctly
- **hello.vec** ensures that basic print commands work
- **map.vec** tests the higher-order function `map`
- **reduce.vec** tests the higher-order function `reduce`
- **control_flow.vec** ensures that Vector's main control structures–`for`, `while`, and `if` work correctly
- **length.vec** ensures that the native function `length` works on arrays.
- **pfor.vec** ensures that our parallel structure `pfor` works.
- **strings.vec** ensures that string printing operations workk
- **inline.vec** ensures that our `inline` macro for injecting CUDA code works
- **logic.vec** ensures that boolean logic works correctly

6.5 Benchmarks

We benchmarked the performance of vector code on the CPU and GPU using the mandelbrot example shown earlier. For the GPU, we used the same code as above. For the CPU, we used similar code, except the pfor statement was replaced with a for statement, and the inner `mandelbrot` function was no longer a device function. By measuring how long it took to complete the computation for increasing image sizes using the `time` builtin function, we were able to compare how CPU and GPU code scaled with increasing workloads.

The benchmarks were performed on a desktop computer with a 2.5 GHz AMD Phenom processor and a NVIDIA GeForce 8400 GS GPU. The results are as follows

Width	Height	Time 1	Time 2	Time 3
640	480	5.16	5.16	5.16
800	600	8.06	8.07	8.07
1024	768	13.22	13.22	13.22
1152	864	16.72	16.73	16.74
1280	960	20.64	20.66	20.65

CPU Results

Width	Height	Time 1	Time 2	Time 3
640	480	0.19	0.19	0.19
800	600	0.28	0.28	0.28
1024	768	0.47	0.46	0.46
1152	864	0.53	0.53	0.53
1280	960	0.61	0.61	0.61

GPU Results

6.6 Responsibilities

Zachary was responsible for the initial configuration of the test suites. The implementers of language features were responsible for their own test suites. Howie wrote the Mandelbrot example and benchmarks.

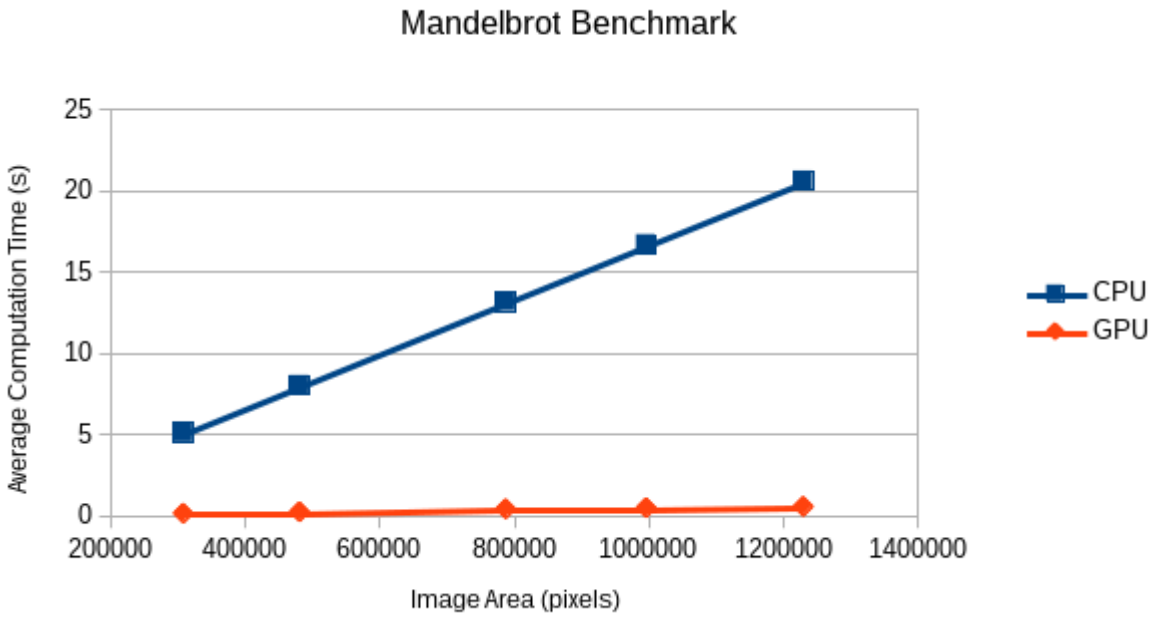

Figure 1: Benchmark Results

7. Lessons Learned

Harry Lee

- Group dynamics proved very important to finishing the project. Although mentioned from the beginning by Professor Edwards, having a leader for the project, Howie, was very helpful for setting up the initial vision and feature set, as well as keeping the development process consistent with the goals at hand. Having said that, active communication between members in weekly meetings was also extremely valuable in making informed decisions about implementing the language.
- For a large group working on a large project, focusing on feature requirements on a given week and to disperse the work instead of having one person to work on a given component was a great decision and one I would suggest future groups to make. In retrospect, this decisino was what gave each member some familiarity to each part, making it easier to make changes iteratively. Along the same lines, pairing for feature developments was also very helpful not only for team members to enforce one another to commit time, but also to have more than one person responsible for a part.

Howard Mao

- Communication with teammates is very important. Enforcing a consistent coding style (especially with respect to indentation) will avoid problems down the line.

Zachary Newman

- The Vector project went very well for several reasons. First, the team was organized. We met every week, and each team member left the meeting with a clear assignment to be completed by the next meeting. Second, the team started early. For this reason, the weekly assignments could be kept small. Third, the team communicated well. We established a mailing list early on. Fourth, the team set up a

standardized development environment: we used Git for version control, GitHub for hosting and issue tracking, and Vagrant to manage virtual development environments.

- The problems the group encountered were distribution of work. Sometimes, members would get excited and work ahead beyond their assigned tasks without consulting the group, leaving less for the others to do. At other times, group members sat idle while the others worked. These issues were mostly ironed out in the end by giving differently-sized assignments in later weeks. The other issue involved working together. The group sometimes assigned two members to one large task, but coordination on this matter was difficult and the task was done slowly. Pair programming or a more disciplined partnership would have mitigated this issue.

Sidharth Shankar

- It's better to segment building the compiler by feature than by phase of the compiler. It's very hard to predict exactly what the grammar should be before implementing code generation. We did this with each of our language features and this worked out well.
- OCaml has a very powerful toolset for building compilers
- Don't use a massive tuple for large amounts of data–after more than two elements, use OCaml product types.
- Don't try to generate all of the code in the same order as the source–from the beginning we should have planned to be able to defer the generation of code.

Jonathan Yu

- The OCaml tools (and the functional programming paradigm in general) are really great for writing compilers. Try to plan your code more before you write it to make it more reusable. Also, do your work early or Howie will do it for you.

Appendix A - Compiler Source Code Listing

A.1 compiler/scanner.mll

```
{ open Parser;; exception Illegal_identifier }

let decdigit = ['0'-'9']
let hexdigit = ['0'-'9' 'a'-'f' 'A'-'Z']
let letter = ['a'-'z' 'A'-'Z' '_']
let floating =
    decdigit+ '.' decdigit* | '.' decdigit+
  | decdigit+ ('.' decdigit*)? 'e' '-'? decdigit+
  | '.' decdigit+ 'e' '-'? decdigit+

rule token = parse
  | [' ' '\t' '\r' '\n'] { token lexbuf }
  | ';' { SC }
  | ':' { COLON }
  | '.' { DOT }
  | ',' { COMMA }
  | '@' { AT }
  | '(' { LPAREN }  | ')' { RPAREN }
  | '{' { LCURLY }  | '}' { RCURLY }
  | '[' { LSQUARE } | ']' { RSQUARE }
  | '=' { EQUAL } | ":=" { DECL_EQUAL }

  | '+' { PLUS } | '-' { MINUS } | '*' { TIMES } | '/' { DIVIDE }
  | '%' { MODULO }
  | "<<" { LSHIFT } | ">>" { RSHIFT }
  | '<' { LT } | "<=" { LTE }
  | '>' { GT } | ">=" { GTE }
  | "==" { EE } | "!=" { NE }
  | '&' { BITAND } | '^' { BITXOR } | '|' { BITOR }
  | "&&" { LOGAND } | "||" { LOGOR }

  | '!' { LOGNOT } | '~' { BITNOT }
  | "++" { INC } | "--" { DEC }

  | "+=" { PLUS_EQUALS } | "-=" { MINUS_EQUALS }
  | "*=" { TIMES_EQUALS } | "/-" { DIVIDE_EQUALS }
  | "%=" { MODULO_EQUALS }
  | "<<=" { LSHIFT_EQUALS } | ">>=" { RSHIFT_EQUALS }
  | "|=" { BITOR_EQUALS } | "&=" { BITAND_EQUALS } | "^=" { BITXOR_EQUALS }

  | '#' { HASH }

  | decdigit+ | "0x" hexdigit+
      as lit { INT_LITERAL(Int32.of_string lit) }
  | (decdigit+ | "0x" hexdigit+ as lit) 'L'
      { INT64_LITERAL(Int64.of_string lit) }
  | floating as lit { FLOAT_LITERAL(float_of_string lit) }
  | '"' (('\\' _ | [^ '"'])* as str) '"'
      { STRING_LITERAL(str) }
  | '\'' ('\\' _ | [^ '\''] | "\\x" hexdigit hexdigit as lit) '\''
```

```
      { CHAR_LITERAL((Scanf.unescaped(lit)).[0]) }

  | "bool" | "char" | "byte" | "int" | "uint"
  | "int8" | "uint8" | "int16" | "uint16"
  | "int32" | "uint32" | "int64" | "uint64"
  | "float" | "float32" | "double" | "float64"
  | "complex" | "complex64" | "complex128"
  | "void" | "string"
      as primtype { TYPE(primtype) }

  | "return" { RETURN }
  | "if" { IF }
  | "else" { ELSE }
  | "for" { FOR }
  | "while" { WHILE }
  | "pfor" { PFOR }
  | "in" { IN }

  | "__sym" decdigit+ "_" letter* { raise Illegal_identifier }
  | "__device__" { DEVICE }
  | letter (letter | decdigit)* as ident { IDENT(ident) }

  | "/*" { comments lexbuf }
  | "//" {inline_comments lexbuf}

  | eof { EOF }

and comments = parse
  | "*/"                { token lexbuf}
  | _                   { comments lexbuf}

and inline_comments = parse
  | "\n"  {token lexbuf}
  | _ {inline_comments lexbuf}
```

A.2 compiler/parser.mly

```
%{ open Ast %}

%token LPAREN RPAREN LCURLY RCURLY LSQUARE RSQUARE
%token DOT COMMA SC COLON AT
%token EQUAL DECL_EQUAL
%token PLUS_EQUALS MINUS_EQUALS TIMES_EQUALS DIVIDE_EQUALS MODULO_EQUALS
%token LSHIFT_EQUALS RSHIFT_EQUALS BITOR_EQUALS BITAND_EQUALS BITXOR_EQUALS
%token LSHIFT RSHIFT BITAND BITOR BITXOR LOGAND LOGOR
%token LT LTE GT GTE EE NE
%token PLUS MINUS TIMES DIVIDE MODULO
%token LOGNOT BITNOT DEC INC
%token IF ELSE WHILE FOR PFOR IN
%token RETURN
%token DEVICE
%token HASH
%token EOF
```

```
%token <int32> INT_LITERAL
%token <int64> INT64_LITERAL
%token <float> FLOAT_LITERAL
%token <string> IDENT TYPE STRING_LITERAL
%token <char> CHAR_LITERAL

%right EQUAL PLUS_EQUALS MINUS_EQUALS TIMES_EQUALS DIVIDE_EQUALS MODULO_EQUALS
    LSHIFT_EQUALS RSHIFT_EQUALS BITOR_EQUALS BITAND_EQUALS BITXOR_EQUALS
%right DOT
%left LOGOR
%left LOGAND
%left BITOR
%left BITXOR
%left BITAND
%left EE NE
%left LT LTE GT GTE
%left LSHIFT RSHIFT
%left PLUS MINUS
%left TIMES DIVIDE MODULO
%right UMINUS LOGNOT BITNOT
%nonassoc IFX
%nonassoc ELSE

%start top_level
%type <Ast.statement list> top_level

%%

ident:
    IDENT { Ident($1) }

datatype:
    TYPE { type_of_string $1 }
  | TYPE LSQUARE RSQUARE { ArrayType(type_of_string $1) }

lvalue:
  | ident { Variable($1) }
  | ident LSQUARE expr_list RSQUARE { ArrayElem($1, $3) }
  | expr DOT ident { ComplexAccess($1, $3) }

expr:
  | expr PLUS expr   { Binop($1, Add, $3) }
  | expr MINUS expr  { Binop($1, Sub, $3) }
  | expr TIMES expr  { Binop($1, Mul, $3) }
  | expr DIVIDE expr { Binop($1, Div, $3) }
  | expr MODULO expr { Binop($1, Mod, $3) }
  | expr LSHIFT expr { Binop($1, Lshift, $3) }
  | expr RSHIFT expr { Binop($1, Rshift, $3) }
  | expr LT expr     { Binop($1, Less, $3) }
  | expr LTE expr    { Binop($1, LessEq, $3) }
  | expr GT expr     { Binop($1, Greater, $3) }
  | expr GTE expr    { Binop($1, GreaterEq, $3) }
  | expr EE expr     { Binop($1, Eq, $3) }
  | expr NE expr     { Binop($1, NotEq, $3) }
```

```
  | expr BITAND expr { Binop($1, BitAnd, $3) }
  | expr BITXOR expr { Binop($1, BitXor, $3) }
  | expr BITOR expr  { Binop($1, BitOr, $3) }
  | expr LOGAND expr { Binop($1, LogAnd, $3) }
  | expr LOGOR expr  { Binop($1, LogOr, $3) }


  | lvalue PLUS_EQUALS expr   { AssignOp($1, Add, $3) }
  | lvalue MINUS_EQUALS expr  { AssignOp($1, Sub, $3) }
  | lvalue TIMES_EQUALS expr  { AssignOp($1, Mul, $3) }
  | lvalue DIVIDE_EQUALS expr { AssignOp($1, Div, $3) }
  | lvalue MODULO_EQUALS expr { AssignOp($1, Mod, $3) }
  | lvalue LSHIFT_EQUALS expr { AssignOp($1, Lshift, $3) }
  | lvalue RSHIFT_EQUALS expr { AssignOp($1, Rshift, $3) }
  | lvalue BITOR_EQUALS expr  { AssignOp($1, BitOr, $3) }
  | lvalue BITAND_EQUALS expr { AssignOp($1, BitAnd, $3) }
  | lvalue BITXOR_EQUALS expr { AssignOp($1, BitXor, $3) }

  | MINUS expr %prec UMINUS { Unop(Neg, $2) }
  | LOGNOT expr { Unop(LogNot, $2) }
  | BITNOT expr { Unop(BitNot, $2) }

  | lvalue DEC { PostOp($1, Dec) }
  | lvalue INC { PostOp($1, Inc) }

  | LPAREN expr RPAREN { $2 }

  | lvalue EQUAL expr { Assign($1, $3) }
  | lvalue            { Lval($1) }

  | INT_LITERAL                { IntLit($1) }
  | INT64_LITERAL              { Int64Lit($1) }
  | FLOAT_LITERAL              { FloatLit($1) }
  | HASH LPAREN expr COMMA expr RPAREN { ComplexLit($3, $5) }
  | STRING_LITERAL             { StringLit($1) }
  | CHAR_LITERAL               { CharLit($1) }
  | datatype LPAREN expr RPAREN { Cast($1, $3) }
  | LCURLY expr_list RCURLY    { ArrayLit($2) }

  | ident LPAREN RPAREN               { FunctionCall($1, []) }
  | ident LPAREN expr_list RPAREN { FunctionCall ($1, $3) }
  | AT ident LPAREN ident COMMA expr RPAREN
     { HigherOrderFunctionCall($2, $4, $6) }

expr_list:
  | expr COMMA expr_list { $1 :: $3 }
  | expr                 { [$1] }

decl:
  | ident DECL_EQUAL expr SC              { AssigningDecl($1, $3) }
  | datatype ident SC                     { PrimitiveDecl($1, $2) }
  | datatype ident LSQUARE RSQUARE SC     { ArrayDecl($1, $2, []) }
  | datatype ident LSQUARE expr_list RSQUARE SC { ArrayDecl($1, $2, $4) }
```

```
statement:
  | IF LPAREN expr RPAREN statement ELSE statement
      { IfStatement($3, $5, $7) }
  | IF LPAREN expr RPAREN statement %prec IFX
      { IfStatement($3, $5, EmptyStatement) }

  | WHILE LPAREN expr RPAREN statement { WhileStatement($3, $5) }
  | FOR LPAREN iterator_list RPAREN statement { ForStatement($3, $5) }
  | PFOR LPAREN iterator_list RPAREN statement { PforStatement($3, $5) }

  | LCURLY statement_seq RCURLY { CompoundStatement($2) }

  | expr SC { Expression($1) }
  | SC { EmptyStatement }
  | decl { Declaration($1) }

  | RETURN expr SC { ReturnStatement($2) }
  | RETURN SC { VoidReturnStatement }


iterator_list:
  | iterator COMMA iterator_list { $1 :: $3 }
  | iterator { [$1] }

iterator:
  | ident IN range { RangeIterator($1, $3) }
  | ident IN expr { ArrayIterator($1, $3) }

range:
  | expr COLON expr COLON expr { Range($1, $3, $5) }
  | expr COLON expr { Range($1, $3, IntLit(1l)) }
  | COLON expr COLON expr { Range(IntLit(0l), $2, $4) }
  | COLON expr { Range(IntLit(0l), $2, IntLit(1l)) }

top_level_statement:
  | datatype ident LPAREN param_list RPAREN LCURLY statement_seq RCURLY
      { FunctionDecl(false, $1, $2, $4, $7) }
  | DEVICE datatype ident LPAREN param_list RPAREN LCURLY statement_seq RCURLY
      { FunctionDecl(true, $2, $3, $5, $8) }
  | datatype ident LPAREN param_list RPAREN SC
      { ForwardDecl(false, $1, $2, $4) }
  | DEVICE datatype ident LPAREN param_list RPAREN SC
      { ForwardDecl(true, $2, $3, $5) }
  | decl { Declaration($1) }

param:
  | datatype ident { PrimitiveDecl($1, $2) }
  | datatype ident LSQUARE RSQUARE
      { ArrayDecl($1, $2, []) }

non_empty_param_list:
  | param COMMA non_empty_param_list { $1 :: $3 }
  | param { [$1] }
```

```
param_list:
  | non_empty_param_list { $1 }
  | { [] }

top_level:
  | top_level_statement top_level {$1 :: $2}
  | top_level_statement { [$1] }

statement_seq:
  | statement statement_seq {$1 :: $2 }
  | { [] }

%%
```

A.3 compiler/ast.ml

```
exception Invalid_type of string

type binop =
  | Add | Sub | Mul | Div | Mod
  | Lshift | Rshift
  | Less | LessEq | Greater | GreaterEq | Eq | NotEq
  | BitAnd | BitXor | BitOr | LogAnd | LogOr

type unop = Neg | LogNot | BitNot

type postop = Dec | Inc

type datatype =
  | Bool | Char | Int8
  | UInt8
  | Int16
  | UInt16
  | Int32
  | UInt32
  | Int64
  | UInt64
  | Float32
  | Float64
  | Complex64
  | Complex128
  | String
  | Void
  | ArrayType of datatype

type ident = Ident of string

type lvalue =
  | Variable of ident
  | ArrayElem of ident * expr list
  | ComplexAccess of expr * ident
and expr =
  | Binop of expr * binop * expr
```

```
  | AssignOp of lvalue * binop * expr
  | Unop of unop * expr
  | PostOp of lvalue * postop
  | Assign of lvalue * expr
  | IntLit of int32
  | Int64Lit of int64
  | FloatLit of float
  | ComplexLit of expr * expr
  | StringLit of string
  | CharLit of char
  | ArrayLit of expr list
  | Cast of datatype * expr
  | FunctionCall of ident * expr list
  | HigherOrderFunctionCall of ident * ident * expr
  | Lval of lvalue

type decl =
  | AssigningDecl of ident * expr
  | PrimitiveDecl of datatype * ident
  | ArrayDecl of datatype * ident * expr list

type range = Range of expr * expr * expr

type iterator =
  | RangeIterator of ident * range
  | ArrayIterator of ident * expr

type statement =
  | CompoundStatement of statement list
  | Declaration of decl
  | Expression of expr
  | EmptyStatement
  | IfStatement of expr * statement * statement
  | WhileStatement of expr * statement
  | ForStatement of iterator list * statement
  | PforStatement of iterator list * statement
  | FunctionDecl of bool * datatype * ident * decl list * statement list
  | ForwardDecl of bool * datatype * ident * decl list
  | ReturnStatement of expr
  | VoidReturnStatement

let type_of_string = function
  | "bool" -> Bool
  | "char" -> Char
  | "int8" -> Int8
  | "uint8" -> UInt8
  | "int16" -> Int16
  | "uint16" -> UInt16
  | "int" -> Int32
  | "int32" -> Int32
  | "uint" -> UInt32
  | "uint32" -> UInt32
  | "int64" -> Int64
  | "uint64" -> UInt64
```

```
  | "double" -> Float64
  | "float" -> Float32
  | "float32" -> Float32
  | "float64" -> Float64
  | "complex" -> Complex64
  | "complex64" -> Complex64
  | "complex128" -> Complex128
  | "void" -> Void
  | dtype -> raise (Invalid_type dtype)
```

A.4 compiler/symgen.ml

```
let sym_num = ref 0;;

let gensym () =
    let sym = "__sym" ^ string_of_int !sym_num ^ "_" in
        sym_num := (!sym_num + 1); sym
```

A.5 compiler/environment.ml

```
open Ast
open Symgen
(*
 * When a new scope is entered, we push to the VariableMap stack; when a scope
 * is left, we pop.
 * *)

(* Maps variable idents to type *)
module VariableMap = Map.Make(struct type t = ident let compare = compare end);;

(* Maps function idents to return type *)
module FunctionMap = Map.Make(struct type t = ident let compare = compare end);;

module FunctionDeclarationMap = Map.Make(struct type t = ident let compare = compare end);;

exception Already_declared
exception Invalid_environment
exception Invalid_operation
exception Not_implemented
exception Symbol_not_found of string

(* Kernel invocation functions are functions that invoke the kernel. The
 * kernel_invoke_sym refers to the function name of this kernel invocation
 * function, while the kernel_sym refers to the function name of the kernel
 * function being invoked.
 *)
type kernel_invocation_function = {
  kernel_invoke_sym: string;
  kernel_sym: string;
  func_type: string;
  higher_order_func: ident;
  };;
```

```
(* Kernel functions are the functions that are executed on the GPU.
 * The function_name is the name of the function in the vector code
 * that is being called, and kernel_symbol is the function name of the
 * generated global function that the kernel is compiled to (it needs to
 * have an additional __global__ directive and operate on arrays, which is
 * why it is a seperate function.  The hof is the type of higher-order-function
 * being invoked.
 *)
type kernel_function = {
  function_name: string;
  hof: ident;
  kernel_symbol: string;
  function_type: string;
};;

type pfor_kernel = {
    pfor_kernel_name: string;
    pfor_iterators: iterator list;
    pfor_arguments: decl list;
    pfor_statement: statement;
};;

type 'a env = {
  kernel_invocation_functions: kernel_invocation_function list;
  kernel_functions: kernel_function list;
  func_type_map: (bool * datatype * datatype list) FunctionMap.t;
  scope_stack: datatype VariableMap.t list;
  pfor_kernels: pfor_kernel list;
  on_gpu: bool;
}

type 'a sourcecomponent =
  | Verbatim of string
  | Generator of ('a -> (string * 'a))
  | NewScopeGenerator of ('a -> (string * 'a))

let create =
  {
    kernel_invocation_functions = [];
    kernel_functions = [];
    func_type_map = FunctionMap.empty;
    scope_stack = VariableMap.empty ::[];
    pfor_kernels = [];
    on_gpu = false;
  }

let update_env kernel_invocation_functions kernel_functions
  func_type_map var_map_list pfor_kernels on_gpu =
    {
      kernel_invocation_functions = kernel_invocation_functions;
      kernel_functions = kernel_functions;
      func_type_map = func_type_map;
      scope_stack = var_map_list;
      pfor_kernels = pfor_kernels;
```

```
      on_gpu = on_gpu;
    }

let var_in_scope ident env =
    let rec check_scope scopes =
        match scopes with
          | [] -> false
          | scope :: tail ->
                if VariableMap.mem ident scope then
                    true
                else check_scope tail
    in check_scope env.scope_stack

let get_var_type ident env =
  let rec check_scope scopes =
    match scopes with
     | [] -> let Ident(sym) = ident in raise (Symbol_not_found sym)
     | scope :: tail ->
         if VariableMap.mem ident scope then
           VariableMap.find ident scope
         else
           check_scope tail in
  let scope_stack = env.scope_stack in
  check_scope scope_stack

let is_var_declared ident env =
  match env.scope_stack with
   | [] -> false
   | scope :: tail -> VariableMap.mem ident scope

let set_var_type ident datatype env =
  let scope, tail = (match env.scope_stack with
                | scope :: tail -> scope, tail
                | [] -> raise Invalid_environment) in
  let new_scope = VariableMap.add ident datatype scope in
  update_env env.kernel_invocation_functions env.kernel_functions
    env.func_type_map (new_scope :: tail) env.pfor_kernels env.on_gpu

let update_scope ident datatype (str, env) =
  if is_var_declared ident env then
    raise Already_declared
  else
    (str, set_var_type ident datatype env)

let push_scope env =
  update_env env.kernel_invocation_functions env.kernel_functions
     env.func_type_map
    (VariableMap.empty :: env.scope_stack) env.pfor_kernels env.on_gpu

let pop_scope env =
  match env.scope_stack with
   | local_scope :: tail ->
      update_env env.kernel_invocation_functions env.kernel_functions
         env.func_type_map tail env.pfor_kernels env.on_gpu
```

```
    | [] -> raise Invalid_environment

let get_func_info ident env =
    FunctionMap.find ident env.func_type_map

let is_func_declared ident env =
  FunctionMap.mem ident env.func_type_map

let update_global_funcs function_type kernel_invoke_sym function_name
    hof kernel_sym (str, env) =
  let new_kernel_funcs = {
    kernel_invoke_sym = kernel_invoke_sym;
    higher_order_func = hof;
    kernel_sym = kernel_sym;
    func_type = function_type;
  } :: env.kernel_invocation_functions in

  let new_global_funcs = {
    function_name = function_name;
    hof = hof;
    kernel_symbol = kernel_sym;
    function_type = function_type;
  } :: env.kernel_functions in

  (str, update_env new_kernel_funcs new_global_funcs env.func_type_map
      env.scope_stack env.pfor_kernels env.on_gpu)

let update_pfor_kernels kernel_name iterators arguments statement (str, env) =
    let new_pfor_kernels = {
        pfor_kernel_name = kernel_name;
        pfor_iterators = iterators;
        pfor_arguments = arguments;
        pfor_statement = statement;
    } :: env.pfor_kernels in
    (str, update_env env.kernel_invocation_functions env.kernel_functions
            env.func_type_map env.scope_stack new_pfor_kernels env.on_gpu)

let set_on_gpu env =
    update_env env.kernel_invocation_functions env.kernel_functions
        env.func_type_map env.scope_stack env.pfor_kernels true

let clear_on_gpu env =
    update_env env.kernel_invocation_functions env.kernel_functions
        env.func_type_map env.scope_stack env.pfor_kernels false

let set_func_type ident device returntype arg_list env =
  let new_func_type_map =
      FunctionMap.add ident (device, returntype, arg_list) env.func_type_map in
  update_env env.kernel_invocation_functions env.kernel_functions
    new_func_type_map env.scope_stack env.pfor_kernels env.on_gpu

let update_functions ident device returntype arg_list (str, env) =
  if is_func_declared ident env then
    raise Already_declared
```

```
  else
    (str, set_func_type ident device returntype arg_list env)

let combine initial_env components =
    let f (str, env) component =
        match component with
         | Verbatim(verbatim) -> str ^ verbatim, env
         | Generator(gen) ->
           let new_str, new_env = gen env in
            str ^ new_str, new_env
         | NewScopeGenerator(gen) ->
           let new_str, new_env = gen (push_scope env) in
            str ^ new_str, pop_scope new_env in
    List.fold_left f ("", initial_env) components
```

A.6 compiler/detect.ml

```
open Environment
open Ast

exception Not_allowed_on_gpu

module IdentSet = Set.Make(
  struct
    let compare = (
      let ident_to_str = function Ident(s) -> s
      in (fun i1 i2 -> Pervasives.compare (ident_to_str i1) (ident_to_str i2))
    )
    type t = ident
  end
);;

let combine_detect_tuples tuplist =
    let combine_pair tup1 tup2 =
        let (in1, out1) = tup1 and
            (in2, out2) = tup2 in
        (IdentSet.union in1 in2, IdentSet.union out1 out2) in
    List.fold_left combine_pair (IdentSet.empty, IdentSet.empty) tuplist;;

let rec detect_statement stmt env =
    match stmt with
      | CompoundStatement(slst) -> detect_statement_list slst env
      | Declaration(decl) -> detect_decl decl env
      | Expression(expr) -> detect_expr expr env
      | IfStatement(expr, stmt1, stmt2) -> combine_detect_tuples
            [detect_expr expr env; detect_statement stmt1 env;
             detect_statement stmt2 env]
      | WhileStatement(expr, stmt) -> combine_detect_tuples
            [detect_expr expr env; detect_statement stmt env]
      | ForStatement(iter_list, stmt) -> combine_detect_tuples
            [detect_iter_list iter_list env; detect_statement stmt env]
      | EmptyStatement -> (IdentSet.empty, IdentSet.empty)
      | _ -> raise Not_allowed_on_gpu
```

```
and detect_decl decl env =
    match decl with
      | AssigningDecl(ident, expr) -> detect_expr expr env
      | _ -> (IdentSet.empty, IdentSet.empty)
and detect_expr expr env =
    match expr with
      | Binop(expr1, _, expr2) -> combine_detect_tuples
            [detect_expr expr1 env; detect_expr expr2 env]
      | AssignOp(lvalue, _, expr) -> combine_detect_tuples
            [detect_lvalue lvalue true true env ; detect_expr expr env]
      | Unop(_, expr) -> detect_expr expr env
      | PostOp(lvalue, _) -> detect_lvalue lvalue true true env
      | Assign(lvalue, expr) -> combine_detect_tuples
            [detect_lvalue lvalue false true env; detect_expr expr env]
      | Cast(_, expr) -> detect_expr expr env
      | FunctionCall(ident, elist) ->
            detect_expr_list elist env
      | Lval(lvalue) -> detect_lvalue lvalue true false env
      | _ -> (IdentSet.empty, IdentSet.empty)
and detect_expr_list elist env =
    let tuplist = List.map (fun expr -> detect_expr expr env) elist in
    combine_detect_tuples tuplist
and detect_statement_list slst env =
    let tuplist = List.map (fun stmt -> detect_statement stmt env) slst in
    combine_detect_tuples tuplist
and detect_lvalue lvalue ins outs env =
    match lvalue with
      | Variable(ident) ->
            if var_in_scope ident env then
                if outs then
                    (IdentSet.empty, IdentSet.empty)
                else if ins then
                    (IdentSet.singleton ident, IdentSet.empty)
                else (IdentSet.empty, IdentSet.empty)
            else (IdentSet.empty, IdentSet.empty)
      | ArrayElem(ident, indices) ->
            let (indins, indouts) as indtups =
                detect_expr_list indices env in
            if var_in_scope ident env then
                ((if ins then IdentSet.add ident indins else indins),
                 (if outs then IdentSet.add ident indouts else indouts))
            else indtups
      | ComplexAccess(expr, _) ->
            (match expr with
              | Lval(Variable(ident)) ->
                    if var_in_scope ident env then
                        ((if ins then IdentSet.singleton ident else IdentSet.empty),
                         (if outs then IdentSet.singleton ident else IdentSet.empty))
                    else (IdentSet.empty, IdentSet.empty)
              | _ -> detect_expr expr env)
and detect_iter iter env =
    match iter with
      | RangeIterator(_, range) ->
            detect_range range env
```

```
      | ArrayIterator(_, expr) ->
            detect_expr expr env
and detect_iter_list iter_list env =
    let tuplist = List.map (fun iter -> detect_iter iter env) iter_list in
    combine_detect_tuples tuplist
and detect_range range env =
    let Range(expr1, expr2, expr3) = range in
    combine_detect_tuples
        [detect_expr expr1 env; detect_expr expr2 env; detect_expr expr3 env]

let detect stmt env =
    let gpu_inputs, gpu_outputs = detect_statement stmt env in
    (IdentSet.elements gpu_inputs, IdentSet.elements gpu_outputs)

let dedup lst =
    IdentSet.elements (List.fold_left
        (fun set itm -> IdentSet.add itm set) IdentSet.empty lst)
```

A.7 compiler/generator.ml

```
open Ast
open Complex
open Environment
open Symgen
open Detect

module StringMap = Map.Make(String);;

exception Unknown_type
exception Empty_list
exception Type_mismatch of string
exception Not_implemented (* this should go away *)
exception Invalid_operation
exception Syntax_error of int * int * string

let rec infer_type expr env =
    let f type1 type2 =
        match type1 with
         | Some(t) -> (if t = type2 then Some(t)
                       else raise (Type_mismatch "wrong type in list"))
         | None -> Some(type2) in
    let match_type expr_list =
      let a = List.fold_left f None expr_list in
        match a with
         | Some(t) -> t
         | None -> raise Empty_list in
    match expr with
      | Binop(expr1, op, expr2) -> (match op with
         | LogAnd | LogOr | Eq | NotEq | Less | LessEq | Greater | GreaterEq ->
           Bool
         | _ -> match_type [infer_type expr1 env; infer_type expr2 env])
      | CharLit(_) -> Char
      | ComplexLit(re, im) ->
```

```
    (match (infer_type re env), (infer_type im env) with
      | (Float32, Float32) -> Complex64
      | (Float64, Float64) -> Complex128
      | (t1, t2) -> raise (Type_mismatch "expected complex"))
| FloatLit(_) -> Float64
| Int64Lit(_) -> Int64
| IntLit(_) -> Int32
| StringLit(_) -> String
| ArrayLit(exprs) ->
    let f expr = infer_type expr env in
    ArrayType(match_type (List.map f exprs))
| Cast(datatype, expr) -> datatype
| Lval(lval) -> (match lval with
    | ArrayElem(ident, _) ->
          (match infer_type (Lval(Variable(ident))) env with
            | ArrayType(t) -> t
            | _ -> raise (Type_mismatch "Cannot access element of non-array"))
    | Variable(i) -> Environment.get_var_type i env
    | ComplexAccess(expr1,ident) ->
        (match (infer_type expr1 env) with
          | Complex64 -> Float32
          | Complex128 -> Float64
          | _ -> raise Invalid_operation))
| AssignOp(lval, _, expr) ->
     let l = Lval(lval) in
     match_type [infer_type l env; infer_type expr env]
| Unop(op, expr) -> (match op with
     LogNot -> Bool
    | _ -> infer_type expr env
  )
| PostOp(lval, _) -> let l = Lval(lval) in infer_type l env
| Assign(lval, expr) ->
     let l = Lval(lval) in
     match_type [infer_type l env; infer_type expr env]
| FunctionCall(i, es) -> (match i with
    | Ident("len") | Ident("random") -> Int32
    | Ident("printf") | Ident("inline") | Ident("assert") -> Void
    | Ident("time") -> Float64
    | Ident("abs") -> (match es with
        | [expr] -> (match infer_type expr env with
            | Complex64 -> Float32
            | Complex128 -> Float64
            | t -> t)
        | _ -> raise (Type_mismatch "Wrong number of arguments to abs()"))
    | _ -> let (_,dtype,_) = Environment.get_func_info i env in dtype)

| HigherOrderFunctionCall(hof, f, expr) ->
    (match(hof) with
      | Ident("map") -> ArrayType(let (_,dtype,_) =
            Environment.get_func_info f env in dtype)
      | Ident("reduce") -> let (_,dtype,_) =
            Environment.get_func_info f env in dtype
      | _ -> raise Invalid_operation)
```

```
let generate_ident ident env =
  match ident with
    Ident(s) -> Environment.combine env [Verbatim(s)]

let rec generate_datatype datatype env =
    match datatype with
      | Bool -> Environment.combine env [Verbatim("bool")]
      | Char -> Environment.combine env [Verbatim("char")]
      | Int8 -> Environment.combine env [Verbatim("int8_t")]
      | UInt8 -> Environment.combine env [Verbatim("uint8_t")]
      | Int16 -> Environment.combine env [Verbatim("int16_t")]
      | UInt16 -> Environment.combine env [Verbatim("uint16_t")]
      | Int32 -> Environment.combine env [Verbatim("int32_t")]
      | UInt32 -> Environment.combine env [Verbatim("uint32_t")]
      | Int64 -> Environment.combine env [Verbatim("int64_t")]
      | UInt64 -> Environment.combine env [Verbatim("uint64_t")]
      | Float32 -> Environment.combine env [Verbatim("float")]
      | Float64 -> Environment.combine env [Verbatim("double")]

      | Complex64 -> Environment.combine env [Verbatim("cuFloatComplex")]
      | Complex128 -> Environment.combine env [Verbatim("cuDoubleComplex")]

      | String -> Environment.combine env [Verbatim("char *")]

      | ArrayType(t) -> Environment.combine env [
            Verbatim("VectorArray<");
            Generator(generate_datatype t);
            Verbatim(">")
        ]

      | _ -> raise Unknown_type

let generate_rettype dtype env =
    match dtype with
      | Void -> "void", env
      | _ -> generate_datatype dtype env

let rec generate_lvalue modify lval env =
    let rec generate_array_index array_id dim exprs env =
        let generate_mid_index array_id dim expr env =
            let typ = match (Environment.get_var_type array_id env) with
              | ArrayType(typ) -> typ
              | _ -> raise (Type_mismatch "Cannot index into non-array") in
            Environment.combine env [
                Verbatim("get_mid_index<");
                Generator(generate_datatype typ);
                Verbatim(">(");
                Generator(generate_ident array_id);
                Verbatim(", ");
                Generator(generate_expr expr);
                Verbatim(", " ^ string_of_int dim ^ ")");
            ] in
    match exprs with
      | [] -> raise Invalid_operation
```

```
      | [expr] ->
            generate_mid_index array_id dim expr env
      | expr :: tail ->
            Environment.combine env [
                Generator(generate_mid_index array_id dim expr);
                Verbatim("+");
                Generator(generate_array_index array_id (dim + 1) tail)
            ] in
  match lval with
    | Variable(i) ->
       Environment.combine env [Generator(generate_ident i)]
    | ArrayElem(ident, es) ->
       if env.on_gpu then
           Environment.combine env [
               Generator(generate_ident ident);
               Verbatim("->values[");
               Generator(generate_array_index ident 0 es);
               Verbatim("]")
           ]
       else
           Environment.combine env [
             Generator(generate_ident ident);
             Verbatim(".elem(" ^ (if modify then "true" else "false") ^ ", ");
             Generator(generate_expr_list es);
             Verbatim(")")
           ]
     | ComplexAccess(expr, ident) -> (
         let _op = match ident with
           Ident("re") -> ".x"
         | Ident("im") -> ".y"
         | Ident(sym) -> raise (Symbol_not_found sym) in
           Environment.combine env [
             Generator(generate_expr expr);
             Verbatim(_op)
           ]
      )
and generate_expr expr env =
  match expr with
    Binop(e1,op,e2) ->
      let datatype = (infer_type e1 env) in
      (match datatype with
        | Complex64 | Complex128 ->
           let func = match op with
             | Add -> "cuCadd"
             | Sub -> "cuCsub"
             | Mul -> "cuCmul"
             | Div -> "cuCdiv"
             | _ -> raise Invalid_operation in
           let func = if datatype == Complex128 then
               func else func ^ "f" in
           Environment.combine env [
               Verbatim(func ^ "(");
               Generator(generate_expr e1);
               Verbatim(",");
```

```
            Generator(generate_expr e2);
            Verbatim(")")
          ]
      | _ ->
       let _op = match op with
         | Add -> "+"
         | Sub -> "-"
         | Mul -> "*"
         | Div -> "/"
         | Mod -> "%"
         | Lshift -> "<<"
         | Rshift -> ">>"
         | Less -> "<"
         | LessEq -> "<="
         | Greater -> ">"
         | GreaterEq -> ">="
         | Eq -> "=="
         | NotEq -> "!="
         | BitAnd -> "&"
         | BitXor -> "^"
         | BitOr -> "|"
         | LogAnd -> "&&"
         | LogOr -> "||"
       in
       Environment.combine env [
         Verbatim("(");
         Generator(generate_expr e1);
         Verbatim(") " ^ _op ^ " (");
         Generator(generate_expr e2);
         Verbatim(")");
       ])

| AssignOp(lvalue, op, e) ->
    (* change lval op= expr to lval = lval op expr *)
    generate_expr (Assign(lvalue, Binop(Lval(lvalue), op, e))) env
| Unop(op,e) -> (
    let simple_unop _op e = Environment.combine env [
        Verbatim(_op);
        Generator(generate_expr e)
    ] in
    match op with
        Neg -> simple_unop "-" e
      | LogNot -> simple_unop "!" e
      | BitNot -> simple_unop "~" e
  )
| PostOp(lvalue, op) -> (
    let _op = match op with
        Dec -> "--"
      | Inc -> "++"
    in
    Environment.combine env [
      Generator(generate_lvalue true lvalue);
      Verbatim(_op)
    ]
```

```
  )
| Assign(lvalue, e) ->
    Environment.combine env [
      Generator(generate_lvalue true lvalue);
      Verbatim(" = ");
      Generator(generate_expr e)
    ]
| IntLit(i) ->
    Environment.combine env [Verbatim(Int32.to_string i)]
| Int64Lit(i) ->
    Environment.combine env [Verbatim(Int64.to_string i)]
| FloatLit(f) ->
    Environment.combine env [Verbatim(string_of_float f)]
| ComplexLit(re, im) as lit ->
    let make_func = (match (infer_type lit env) with
      | Complex64 -> "make_cuFloatComplex"
      | Complex128 -> "make_cuDoubleComplex"
      | t -> raise (Type_mismatch "Mismatch in ComplexLit")) in
    Environment.combine env [
      Verbatim(make_func ^ "(");
      Generator(generate_expr re);
      Verbatim(", ");
      Generator(generate_expr im);
      Verbatim(")")
    ]
| StringLit(s) ->
    Environment.combine env [Verbatim("\"" ^ s ^ "\"")]
| CharLit(c) ->
    Environment.combine env [
      Verbatim("'" ^ Char.escaped c ^ "'")
    ]
| ArrayLit(es) as lit ->
    let typ = (match (infer_type lit env) with
     | ArrayType(t) -> t
     | t -> raise (Type_mismatch "ArrayLit")) in
    let len = List.length es in
    Environment.combine env [
        Verbatim("VectorArray<");
        Generator(generate_datatype typ);
        Verbatim(">(1, (size_t) " ^ string_of_int len ^ ")");
        Generator(generate_array_init_chain 0 es)
    ]
| Cast(d,e) ->
    Environment.combine env [
      Verbatim("(");
      Generator(generate_datatype d);
      Verbatim(") (");
      Generator(generate_expr e);
      Verbatim(")")
    ]
| FunctionCall(i,es) ->
    Environment.combine env (match i with
      | Ident("inline") -> (match es with
          | StringLit(str) :: [] -> [Verbatim(str)]
```

```
    | _ -> raise (Type_mismatch "expected string"))
| Ident("printf") -> [
    Verbatim("printf(");
    Generator(generate_expr_list es);
    Verbatim(")");
  ]
| Ident("len") -> (match es with
    | expr :: [] -> (match (infer_type expr env) with
        | ArrayType(_) -> [
            Verbatim("(");
            Generator(generate_expr expr);
            Verbatim(").size()")
        ]
        | String -> [
            Verbatim("strlen(");
            Generator(generate_expr expr);
            Verbatim(")")
        ]
        | _ -> raise (Type_mismatch "cannot compute length"))
    | expr1 :: expr2 :: [] ->
        (match (infer_type expr1 env), (infer_type expr2 env) with
          | (ArrayType(_), Int32) -> [
              Verbatim("(");
              Generator(generate_expr expr1);
              Verbatim(").length(");
              Generator(generate_expr expr2);
              Verbatim(")");
          ]
          | _ -> raise (Type_mismatch
              "len() with two arguments must have types array and int"))
    | _ -> raise (Type_mismatch "incorrect number of parameters"))
| Ident("assert") -> (match es with
    | expr :: [] -> (match infer_type expr env with
        | Bool -> [
            Verbatim("if (!(");
            Generator(generate_expr expr);
            Verbatim(")) {printf(\"Assertion failed: ");
            Generator(generate_expr expr);
            Verbatim("\"); exit(EXIT_FAILURE); }");
        ]
        | _ -> raise (Type_mismatch "Cannot assert on non-boolean expression"))
    | _ -> raise (Type_mismatch "too many parameters"))
| Ident("random") -> (match es with
    | [] -> [ Verbatim("random()") ]
    | _ -> raise (Type_mismatch "Too many argument to random"))
| Ident("abs") -> (match es with
    | [expr] ->
        let absfunc = (match infer_type expr env with
          | Int8 | Int16 | Int32 | Int64 | Float32 | Float64 -> "abs"
          | Complex64 -> "cuCabsf"
          | Complex128 -> "cuCabs"
          | _ -> raise (Type_mismatch "abs() takes only numbers")) in
        [
            Verbatim(absfunc ^ "(");
```

```
                    Generator(generate_expr expr);
                    Verbatim(")");
                ]
            | _ -> raise (Type_mismatch "Wrong number of arguments to abs()"))
        | Ident("time") -> (match es with
            | [] -> [ Verbatim("get_time()") ]
            | _ -> raise (Type_mismatch "time() takes no arguments"))
        | _ -> let _, _, arg_list = Environment.get_func_info i env in
            let rec validate_type_list types expressions env =
                match (types, expressions) with
                  | typ :: ttail, expr :: etail ->
                        if (typ = infer_type expr env) then
                            validate_type_list ttail etail env
                        else false
                  | [], [] -> true
                  | [], _ -> false
                  | _, [] -> false in
            if (validate_type_list arg_list es env) then [
                Generator(generate_ident i);
                Verbatim("(");
                Generator(generate_expr_list es);
                Verbatim(")")
            ] else raise (Type_mismatch "Arguments of wrong type"))

  | HigherOrderFunctionCall(i1,i2,es) ->
      (match infer_type es env with
        | ArrayType(function_type) ->
            let function_type_str, _ = generate_datatype function_type env in
            let kernel_invoke_sym = Symgen.gensym () in
            let kernel_sym = Symgen.gensym () in
            let function_name, _ = generate_ident i2 env in
            Environment.update_global_funcs function_type_str kernel_invoke_sym
                function_name i1 kernel_sym (Environment.combine env [
                    Verbatim(kernel_invoke_sym ^ "(");
                    Generator(generate_expr es);
                    Verbatim(")")])
        | t -> let dtype, _ = generate_datatype t env in
            raise (Type_mismatch
                    ("Expected array as argument to HOF, got " ^ dtype)))
  | Lval(lvalue) ->
      Environment.combine env [Generator(generate_lvalue false lvalue)]
and generate_array_init_chain ind expr_list env =
    match expr_list with
      | [] -> "", env
      | expr :: tail ->
            Environment.combine env [
                Verbatim(".chain_set(" ^ string_of_int ind ^ ", ");
                Generator(generate_expr expr);
                Verbatim(")");
                Generator(generate_array_init_chain (ind + 1) tail)
            ]
and generate_expr_list expr_list env =
  match expr_list with
   | [] -> Environment.combine env []
```

```
   | lst ->
       Environment.combine env [Generator(generate_nonempty_expr_list lst)]
and generate_nonempty_expr_list expr_list env =
  match expr_list with
   | expr :: [] ->
       Environment.combine env [Generator(generate_expr expr)]
   | expr :: tail ->
       Environment.combine env [
         Generator(generate_expr expr);
         Verbatim(", ");
         Generator(generate_nonempty_expr_list tail)
       ]
   | [] -> raise Empty_list
and generate_decl decl env =
  match decl with
    | AssigningDecl(ident,e) ->
        let datatype = (infer_type e env) in
        Environment.update_scope ident datatype
            (Environment.combine env [
              Generator(generate_datatype datatype);
              Verbatim(" ");
              Generator(generate_ident ident);
              Verbatim(" = ");
              Generator(generate_expr e)
            ])
    | PrimitiveDecl(d,i) ->
        Environment.update_scope i d (Environment.combine env [
          Generator(generate_datatype d);
          Verbatim(" ");
          Generator(generate_ident i)
        ])
    | ArrayDecl(d,i,es) ->
        Environment.update_scope i (ArrayType(d))
            (if env.on_gpu then (match es with
              | [] -> Environment.combine env [
                  Verbatim("device_info<");
                  Generator(generate_datatype d);
                  Verbatim("> *");
                  Generator(generate_ident i);
              ]
              | _ -> raise Not_allowed_on_gpu)
            else (match es with
              | [] -> Environment.combine env [
                  Verbatim("VectorArray<");
                  Generator(generate_datatype d);
                  Verbatim("> ");
                  Generator(generate_ident i);
              ]
              | _ ->
                  Environment.combine env [
                    Verbatim("VectorArray<");
                    Generator(generate_datatype d);
                    Verbatim("> ");
                    Generator(generate_ident i);
```

```
                  Verbatim("(" ^ string_of_int (List.length es) ^ ", ");
                  Generator(generate_expr_list es);
                  Verbatim(")")
                ]))

let generate_range range env =
  match range with
   | Range(e1,e2,e3) ->
      Environment.combine env [
        Verbatim("range(");
        Generator(generate_expr e1);
        Verbatim(", ");
        Generator(generate_expr e2);
        Verbatim(", ");
        Generator(generate_expr e3);
        Verbatim(")")
      ]

let generate_iterator iterator env =
  match iterator with
   | RangeIterator(i, r) ->
       Environment.combine env [
         Generator(generate_ident i);
         Verbatim(" in ");
         Generator(generate_range r)
       ]
   | ArrayIterator(i, e) ->
       Environment.combine env [
         Generator(generate_ident i);
         Verbatim(" in ");
         Generator(generate_expr e)
       ]

let rec generate_iterator_list iterator_list env =
  match iterator_list with
   | [] -> Environment.combine env [Verbatim("_")]
   | iterator :: tail ->
       Environment.combine env [
         Generator(generate_iterator iterator);
         Verbatim(", ");
         Generator(generate_iterator_list tail)
       ]

let rec generate_nonempty_decl_list decl_list env =
  match decl_list with
   | decl :: [] ->
       Environment.combine env [Generator(generate_decl decl)]
   | decl :: tail ->
       Environment.combine env [
         Generator(generate_decl decl);
         Verbatim(", ");
         Generator(generate_nonempty_decl_list tail)
       ]
   | [] -> raise Empty_list
```

```
let generate_decl_list decl_list env =
  match decl_list with
   | [] -> Environment.combine env [Verbatim("void")]
   | (decl :: tail) as lst ->
       Environment.combine env [
         Generator(generate_nonempty_decl_list lst)
       ]

let rec generate_statement statement env =
    match statement with
      | CompoundStatement(ss) ->
          Environment.combine env [
            Verbatim("{\n");
            NewScopeGenerator(generate_statement_list ss);
            Verbatim("}\n")
          ]
      | Declaration(d) ->
          Environment.combine env [
            Generator(generate_decl d);
            Verbatim(";")
          ]
      | Expression(e) ->
          Environment.combine env [
            Generator(generate_expr e);
            Verbatim(";")
          ]
      | EmptyStatement ->
          Environment.combine env [Verbatim(";")]
      | IfStatement(e, s1, s2) ->
          Environment.combine env [
            Verbatim("if (");
            Generator(generate_expr e);
            Verbatim(")\n");
            NewScopeGenerator(generate_statement s1);
            Verbatim("\nelse\n");
            NewScopeGenerator(generate_statement s2)
          ]
      | WhileStatement(e, s) ->
          Environment.combine env [
            Verbatim("while (");
            Generator(generate_expr e);
            Verbatim(")\n");
            NewScopeGenerator(generate_statement s)
          ]
      | ForStatement(is, s) ->
          Environment.combine env [
            NewScopeGenerator(generate_for_statement (is, s));
          ]
      | PforStatement(is, s) ->
          Environment.combine env [
            NewScopeGenerator(generate_pfor_statement is s)
          ]
      | FunctionDecl(device, return_type, identifier, arg_list, body_sequence) ->
```

```
            let env = if device then set_on_gpu env else env in
            let str, env = Environment.combine env [
                NewScopeGenerator(generate_function device return_type
                                  identifier arg_list body_sequence)
              ] in
            let env = if device then clear_on_gpu env else env in
            let types = List.map (function x ->
              match x with
              |PrimitiveDecl(t, id) -> t
              |ArrayDecl(t, id, expr_list) -> ArrayType(t)
              | _ -> raise Invalid_operation
            ) arg_list in
            let new_str, new_env = Environment.update_functions identifier
              device return_type types (str, env) in
            new_str, new_env

        | ForwardDecl(device, return_type, ident, decl_list) ->
              let types = List.map (function x ->
                match x with
                |PrimitiveDecl(t, id) -> t
                |ArrayDecl(t, id, expr_list) -> ArrayType(t)
                | _ -> raise Invalid_operation
              ) decl_list in
              Environment.update_functions ident device return_type types
                  (Environment.combine env [
                      Verbatim(if device then "__device__ " else "");
                      Generator(generate_datatype return_type);
                      Verbatim(" ");
                      Generator(generate_ident ident);
                      Verbatim("(");
                      Generator(generate_decl_list decl_list);
                      Verbatim(");")
                  ])
        | ReturnStatement(e) ->
            Environment.combine env [
              Verbatim("return ");
              Generator(generate_expr e);
              Verbatim(";")
            ]
        | VoidReturnStatement ->
            Environment.combine env [Verbatim("return;")]

and generate_statement_list statement_list env =
    match statement_list with
     | [] -> Environment.combine env []
     | statement :: tail ->
         Environment.combine env [
           Generator(generate_statement statement);
           Verbatim("\n");
           Generator(generate_statement_list tail)
         ]

and generate_function device returntype ident params statements env =
    Environment.combine env [
```

```
        Verbatim(if device then "__device__ " else "");
        Generator(generate_rettype returntype);
        Verbatim(" ");
        Generator(generate_ident ident);
        Verbatim("(");
        Generator(generate_decl_list params);
        Verbatim(") {\n");
        Generator(generate_statement_list statements);
        Verbatim("}");
    ]

(* TODO: support multi-dimensional arrays *)
(* TODO: clean up this humongous mess *)
(* TODO: support negative integers in ranges *)
(* TODO: modify the type system so we can use size_t where appropriate *)
and generate_for_statement (iterators, statements) env =

  let iter_name iterator = match iterator with
    ArrayIterator(Ident(s),_) -> s
  | RangeIterator(Ident(s),_) -> s
  in

  (* map iterators to their properties
   * key on s rather than Ident(s) to save some effort... *)
  let iter_map =

    (* create symbols for an iterator's length and index.
     *
     * there is also a mod symbol - since we're flattening multiple
     * iterators into a single loop, we need to know how often each one
     * wraps around
     *
     * the output symbol is simply the symbol requested in the original
     * vector code
     *
     * for ranges, we have start:_:inc
     * for arrays, start = 0, inc = 1
     *)
    let get_iter_properties iterator =
      let len_sym = Ident(Symgen.gensym () ^ iter_name iterator ^ "_len") in
      let mod_sym = Ident(Symgen.gensym () ^ iter_name iterator ^ "_mod") in
      let div_sym = Ident(Symgen.gensym () ^ iter_name iterator ^ "_div") in
      let output_sym = match iterator with
        ArrayIterator(i,_) -> i
      | RangeIterator(i,_) -> i
      in
      (* start_sym and inc_sym are never actually used for array iterators...
       * the only consequence is that the generated symbols in our output code
       * will be non-consecutive because of these "wasted" identifiers *)
      let start_sym = Ident(Symgen.gensym () ^ iter_name iterator ^ "_start") in
      let inc_sym = Ident(Symgen.gensym () ^ iter_name iterator ^ "_inc") in
      (iterator, len_sym, mod_sym, div_sym, output_sym, start_sym, inc_sym)
    in
```

```
  List.fold_left (fun m i -> StringMap.add (iter_name i)
                  (get_iter_properties i) (m)) (StringMap.empty) (iterators)
in

(* generate code to calculate the length of each iterator
 * and initialize the corresponding variables
 *
 * also calculate start and inc *)
let iter_length_initializers =

  let iter_length_inits _ (iter, len_sym, _, _, _, start_sym, inc_sym) acc =
    match iter with
      ArrayIterator(_,e) -> [
        Declaration(
          AssigningDecl(len_sym, FunctionCall(Ident("len"), e :: [])))
      ] :: acc
    | RangeIterator(_,Range(start_expr,stop_expr,inc_expr)) -> (
        (* the number of iterations in the iterator a:b:c is n, where
         * n = (b-a-1) / c + 1 *)
        (* TODO: make sure we never get negative lengths *)
        let delta = Binop(stop_expr, Sub, Lval(Variable(start_sym))) in
        let delta_fencepost = Binop(delta, Sub, IntLit(Int32.of_int 1)) in
        let n = Binop(delta_fencepost, Div, Lval(Variable(inc_sym))) in
        let len_expr = Binop(n, Add, IntLit(Int32.of_int 1)) in
        [
          Declaration(AssigningDecl(start_sym, start_expr));
          Declaration(AssigningDecl(inc_sym, inc_expr));
          Declaration(AssigningDecl(len_sym, len_expr));
        ] :: acc
      )
  in

  List.concat (StringMap.fold (iter_length_inits) (iter_map) ([]))
in

(* the total length of our for loop is the product
 * of lengths of all iterators *)
let iter_max =
  StringMap.fold (fun _ (_, len_sym, _, _, _, _, _) acc ->
    Binop(acc, Mul, Lval(Variable(len_sym)))) (iter_map) (IntLit(Int32.of_int 1))
in

(* figure out how often each iterator wraps around
 * the rightmost iterator wraps the fastest (i.e. mod its own length)
 * all other iterators wrap modulo (their own length times the mod of
 * the iterator directly to the right) *)
let iter_mod_initializers =
  let iter_initializer iterator acc =
    let name = iter_name iterator in
    let (_, len_ident, mod_ident, div_ident, _, _, _) =
        StringMap.find name iter_map in
    match acc with
      [] -> [
        Declaration(AssigningDecl(mod_ident, Lval(Variable(len_ident))));
```

```
        Declaration(AssigningDecl(div_ident, IntLit(Int32.of_int 1)));
      ]
    | Declaration(AssigningDecl(prev_mod_ident, _)) :: _ -> (
        List.append [
          Declaration(AssigningDecl(mod_ident, Binop(
              Lval(Variable(len_ident)), Mul,
              Lval(Variable(prev_mod_ident)))));
          Declaration(AssigningDecl(div_ident, Lval(Variable(prev_mod_ident))));
        ] acc)
    | _ -> [] (* TODO: how do we represent impossible outcomes? *)
  in
  (* we've built up the list with the leftmost iterator first,
   * but we need the rightmost declared first due to dependencies.
   * reverse it! *)
  List.rev (List.fold_right (iter_initializer) (iterators) ([]))
in

(* initializers for the starting and ending value
 * of the index we're generating *)
let iter_ptr_ident = Ident(Symgen.gensym () ^ "iter_ptr") in
let iter_max_ident = Ident(Symgen.gensym () ^ "iter_max") in
let bounds_initializers = [
  Declaration(AssigningDecl(iter_ptr_ident, IntLit(Int32.of_int 0)));
  Declaration(AssigningDecl(iter_max_ident, iter_max));
] in

(* these assignments will occur at the beginning of each iteration *)
let output_assignments =
  let iter_properties s = match (StringMap.find s iter_map) with
    (_, _, mod_sym, div_sym, output_sym, start_sym, inc_sym) ->
      (mod_sym, div_sym, output_sym, start_sym, inc_sym)
  in
  let idx mod_sym div_sym =
    Binop(
      Binop(Lval(Variable(iter_ptr_ident)), Mod, Lval(Variable(mod_sym))),
      Div,
      Lval(Variable(div_sym)))
  in
  let iter_assignment iterator = match iterator with
    (* TODO: to avoid unnecessary copies, we really want to use pointers here *)
    ArrayIterator(Ident(s), Lval(Variable(ident))) -> (
      let mod_sym, div_sym, output_sym, _, _ = iter_properties s in
      (* TODO: we really should store the result of e in a variable, to
       * avoid evaluating it more than once *)
      Declaration(AssigningDecl(output_sym,
          Lval(ArrayElem(ident,[idx mod_sym div_sym])))))
  | RangeIterator(Ident(s),_) -> (
      let mod_sym, div_sym, output_sym, start_sym, inc_sym = iter_properties s in
      let offset = Binop(idx mod_sym div_sym, Mul, Lval(Variable(inc_sym))) in
      let origin = Lval(Variable(start_sym)) in
      let rhs = Binop(origin, Add, offset) in
      Declaration(AssigningDecl(output_sym, rhs))
    )
  | _ -> raise Not_implemented
```

```
    in
    List.map (iter_assignment) (iterators)
  in

  Environment.combine env [
    Verbatim("{\n");
    Generator(generate_statement_list iter_length_initializers);
    Generator(generate_statement_list iter_mod_initializers);
    Generator(generate_statement_list bounds_initializers);
    Verbatim("for (; ");
    Generator(generate_expr (Binop(Lval(Variable(iter_ptr_ident)), Less,
        Lval(Variable(iter_max_ident)))));
    Verbatim("; ");
    Generator(generate_expr (PostOp(Variable(iter_ptr_ident), Inc)));
    Verbatim(") {\n");
    Generator(generate_statement_list output_assignments);
    Generator(generate_statement statements);
    Verbatim("}\n");
    Verbatim("}\n");
  ]
and generate_pfor_statement iters stmt env =
    (* generate intermediate symbols for array iterators *)
    let rec get_array_ident_array ident_array index = function
      | ArrayIterator(_, expr) :: tl ->
          ident_array.(index) <- Symgen.gensym ();
          get_array_ident_array ident_array (index + 1) tl
      | _ :: tl -> get_array_ident_array ident_array (index + 1) tl
      | [] -> ident_array in
    let niters = List.length iters in
    let array_ident_array =
        get_array_ident_array (Array.make niters "") 0 iters in
    (* setup an array of iterator structs *)
    let gen_struct_mem iter_arr index mem_name expr env =
        if (infer_type expr env) == Int32 then
            Environment.combine env [
                Verbatim(iter_arr ^ "[" ^ string_of_int index ^ "]."
                            ^ mem_name ^ " = ");
                Generator(generate_expr expr);
                Verbatim(";\n")
            ]
        else raise (Type_mismatch "Iterator control must have type int32") in
    (* assign start, stop, and inc for each iterator *)
    let rec gen_iter_struct iter_arr index iters env =
        match iters with
          | RangeIterator(_, Range(start_expr, stop_expr, inc_expr)) :: tl ->
              Environment.combine env [
                  Generator(gen_struct_mem iter_arr index "start" start_expr);
                  Generator(gen_struct_mem iter_arr index "stop" stop_expr);
                  Generator(gen_struct_mem iter_arr index "inc" inc_expr);
                  Generator(gen_iter_struct iter_arr (index + 1) tl)
              ]
          | ArrayIterator(_, array_expr) :: tl ->
              let array_sym = Ident(array_ident_array.(index)) in
              Environment.combine env [
```

```
                Generator(generate_decl (AssigningDecl(
                    array_sym, array_expr)));
                Verbatim(";\n");
                Generator(gen_struct_mem iter_arr index "start" (IntLit(0l)));
                Generator(gen_struct_mem iter_arr index "stop"
                    (FunctionCall(Ident("len"),
                        [Lval(Variable(array_sym))])));
                Generator(gen_struct_mem iter_arr index "inc" (IntLit(1l)));
                Generator(gen_iter_struct iter_arr (index + 1) tl)
            ]
      | [] -> "", env in
(* turn the array into a list *)
let rec get_array_ident_list cur_list ident_array index n =
    if index < n then
        match ident_array.(index) with
            (* empty strings are from range iters, so ignore them *)
          | "" -> get_array_ident_list cur_list ident_array (index + 1) n
          | id -> get_array_ident_list (Ident(id) :: cur_list) ident_array
                        (index + 1) n
    else cur_list in
(* array arguments must have their device info pointer passed in
 * everything else can be passed in as-is *)
let generate_kernel_arg env id =
    let Ident(s) = id in
    match get_var_type id env with
      | ArrayType(_) -> s ^ ".devInfo()"
      | _ -> s in
let generate_ident_list ident_list env =
    let ident_str = String.concat ", " (List.map
                        (generate_kernel_arg env) ident_list) in
    (* put a leading comma if non-empty, otherwise just return "" *)
    if ident_str = "" then "", env else (", " ^ ident_str), env in
let generate_output_markings output_list =
    String.concat "" (List.map (function Ident(s) ->
                                s ^ ".markDeviceDirty();\n") output_list) in
let iter_arr = Symgen.gensym () and
    iter_devptr = Symgen.gensym () and
    total_iters = Symgen.gensym () and
    kernel_name = Symgen.gensym () and
    gpu_inputs, gpu_outputs = detect stmt env and
    array_ident_list = get_array_ident_list [] array_ident_array 0 niters in
let full_ident_list =
    Detect.dedup (gpu_outputs @ gpu_inputs @ array_ident_list) in
let gen_kernel_decl id =
    match Environment.get_var_type id env with
      | ArrayType(typ) -> ArrayDecl(typ, id, [])
      | typ -> PrimitiveDecl(typ, id) in
let kernel_args = List.map gen_kernel_decl full_ident_list in
Environment.update_pfor_kernels kernel_name iters kernel_args stmt
    (Environment.combine env [
        Verbatim("{\nstruct range_iter " ^ iter_arr ^
            "[" ^ string_of_int niters ^ "];\n");
        Generator(gen_iter_struct iter_arr 0 iters);
        Verbatim("fillin_iters(" ^ iter_arr ^ ", " ^
```

```
                    string_of_int niters ^ ");\n");
            Verbatim("struct range_iter *" ^ iter_devptr ^
                    " = device_iter(" ^ iter_arr ^ ", " ^
                    string_of_int niters ^ ");\n");
            Verbatim("size_t " ^ total_iters ^ " = total_iterations(" ^
                    iter_arr ^ ", " ^ string_of_int niters ^ ");\n");
            Verbatim(kernel_name ^ "<<<ceil_div(" ^ total_iters ^
                    ", BLOCK_SIZE), BLOCK_SIZE>>>(" ^ iter_devptr ^ ", " ^
                    string_of_int niters ^ ", " ^ total_iters);
            Generator(generate_ident_list full_ident_list);
            Verbatim(");\ncudaDeviceSynchronize();
                    checkError(cudaGetLastError());\n");
            Verbatim(generate_output_markings gpu_outputs);
            Verbatim("cudaFree(" ^ iter_devptr ^ ");\n}\n")
        ])

let generate_toplevel tree =
    let env = Environment.create in
    Environment.combine env [
        Generator(generate_statement_list tree);
        Verbatim("\nint main(void) { return vec_main(); }\n")
    ]

let generate_kernel_invocation_functions env =
  let rec generate_functions funcs str =
    match funcs with
    | [] -> str
    | head :: tail ->
        (match head.higher_order_func with
          | Ident("map") ->
            let new_str = str ^ "\nVectorArray<" ^ head.func_type ^ "> " ^
                head.kernel_invoke_sym ^ "(" ^
            "VectorArray<" ^ head.func_type ^ "> input){
              int inputSize = input.size();
              VectorArray<" ^ head.func_type ^ " > output = input.dim_copy();
              " ^ head.kernel_sym ^
              "<<<ceil_div(inputSize,BLOCK_SIZE),BLOCK_SIZE>>>" ^
              "(output.devPtr(), input.devPtr(), inputSize);
              cudaDeviceSynchronize();
              checkError(cudaGetLastError());
              output.markDeviceDirty();
              return output;
              }\n
            " in
            generate_functions tail new_str
          | Ident("reduce") ->
            let new_str = str ^ head.func_type ^ " " ^
                  head.kernel_invoke_sym ^
                      "(VectorArray<" ^ head.func_type ^ "> arr)
            {
                int n = arr.size();
                int num_blocks = ceil_div(n, BLOCK_SIZE);
                int atob = 1;
                int shared_size = BLOCK_SIZE * sizeof(" ^ head.func_type ^ ");
```

```
                    VectorArray<" ^ head.func_type ^ "> tempa(1, num_blocks);
                    VectorArray<" ^ head.func_type ^ "> tempb(1, num_blocks);

                    " ^ head.kernel_sym ^
                    "<<<num_blocks, BLOCK_SIZE, shared_size>>>" ^
                    "(tempa.devPtr(), arr.devPtr(), n);
                    cudaDeviceSynchronize();
                    checkError(cudaGetLastError());
                    tempa.markDeviceDirty();
                    n = num_blocks;

                    while (n > 1) {
                        num_blocks = ceil_div(n, BLOCK_SIZE);
                        if (atob) {
                            " ^ head.kernel_sym ^
                            "<<<num_blocks, BLOCK_SIZE, shared_size>>>" ^
                            "(tempb.devPtr(), tempa.devPtr(), n);
                            tempb.markDeviceDirty();
                        } else {
                            " ^ head.kernel_sym ^
                            "<<<num_blocks, BLOCK_SIZE, shared_size>>>" ^
                            "(tempa.devPtr(), tempb.devPtr(), n);
                            tempa.markDeviceDirty();
                        }
                        cudaDeviceSynchronize();
                        checkError(cudaGetLastError());
                        atob = !atob;
                        n = num_blocks;
                    }

                    if (atob) {
                        tempa.copyFromDevice(1);
                        return tempa.elem(false, 0);
                    }
                    tempb.copyFromDevice(1);
                    return tempb.elem(false, 0);
                }"  in
                generate_functions tail new_str
            | _ -> raise Invalid_operation) in
    generate_functions env.kernel_invocation_functions " "

let generate_kernel_functions env =
    let env = set_on_gpu env in
    let kernel_funcs = env.kernel_functions in
    let rec generate_funcs funcs str =
      (match funcs with
      | [] -> str
      | head :: tail ->
          (match head.hof with
           | Ident("map") ->
              let new_str = str ^
              "__global__ void " ^ head.kernel_symbol ^ "(" ^ head.function_type ^ "* output,
              " ^ head.function_type ^ "* input, size_t n){
                size_t i = threadIdx.x + blockDim.x * blockIdx.x;
```

```
            if (i < n)
              output[i] = " ^ head.function_name ^ "(input[i]);
          }\n"  in
          generate_funcs tail new_str
      | Ident("reduce") ->
          let new_str = str ^
          "__global__ void " ^ head.kernel_symbol ^ "( " ^ head.function_type ^
            " *output, " ^ head.function_type ^ " *input, size_t n) {
              extern __shared__ " ^ head.function_type ^ " temp[];

              int ti = threadIdx.x;
              int bi = blockIdx.x;
              int starti = blockIdx.x * blockDim.x;
              int gi = starti + ti;
              int bn = min(n - starti, blockDim.x);
              int s;

              if (ti < bn)
                  temp[ti] = input[gi];
              __syncthreads();

              for (s = 1; s < blockDim.x; s *= 2) {
                  if (ti % (2 * s) == 0 && ti + s < bn)
                      temp[ti] = " ^ head.function_name ^ "(temp[ti], temp[ti + s]);
                  __syncthreads();
              }

              if (ti == 0)
                  output[bi] = temp[0];
          }
          " in
          generate_funcs tail new_str

      | _ -> raise Invalid_operation)) in
  generate_funcs kernel_funcs ""

let generate_device_forward_declarations env =
    let gen_type_list types =
        if types = [] then "void"
        else String.concat ", "
            (List.map
                (fun typ -> fst (generate_datatype typ env)) types) in
    let gen_fdecl_if_needed (Ident(id)) (device, rettype, argtypes) genstr =
        if device then
            genstr ^ "__device__ " ^ fst (generate_rettype rettype env) ^ " " ^
            id ^ "(" ^ (gen_type_list argtypes) ^ ");\n"
        else genstr in
    FunctionMap.fold gen_fdecl_if_needed env.func_type_map ""

let generate_pfor_kernels env =
    let env = Environment.set_on_gpu env in
    let rec gen_iter_var_decls iter_arr index_var iter_index iterators env =
        match iterators with
          | [] -> "", env
```

```
        | ArrayIterator(Ident(id), expr) :: tail -> raise Not_implemented
        | RangeIterator(id, _) :: tail ->
            let _, env = Environment.update_scope id Int32 ("", env) in
            Environment.combine env [
                Verbatim("size_t ");
                Generator(generate_ident id);
                Verbatim(" = get_index_gpu(&" ^ iter_arr ^ "[" ^
                        string_of_int iter_index ^ "], " ^ index_var ^ ");\n");
                Generator(gen_iter_var_decls iter_arr index_var
                            (iter_index + 1) tail)
            ] in
    let rec gen_kernel pfor env =
        let iter_arr = Symgen.gensym () and
            niters = Symgen.gensym () and
            total_iters = Symgen.gensym () and
            index_var = Symgen.gensym () in
        Environment.combine env ([
            Verbatim("__global__ void " ^ pfor.pfor_kernel_name ^ "(");
            Verbatim("struct range_iter *" ^ iter_arr ^ ", ");
            Verbatim("size_t " ^ niters ^ ", ");
            Verbatim("size_t " ^ total_iters)
        ] @ (if pfor.pfor_arguments = [] then [Verbatim("){\n")] else [
            Verbatim(", ");
            Generator(generate_nonempty_decl_list pfor.pfor_arguments);
            Verbatim("){\n")
        ]) @ [
            Verbatim("size_t " ^ index_var ^
                    " = threadIdx.x + blockIdx.x * blockDim.x;\n");
            Verbatim("if (" ^ index_var ^ " < " ^ total_iters ^ "){\n");
            Generator(gen_iter_var_decls iter_arr index_var 0
                        pfor.pfor_iterators);
            Generator(generate_statement pfor.pfor_statement);
            Verbatim("}\n}\n");
        ]) in
    let rec generate_kernels str = function
      | [] -> str
      | pfor :: tail ->
            let newstr, _ = Environment.combine env
                [NewScopeGenerator(gen_kernel pfor)] in
            generate_kernels (str ^ newstr) tail in
    generate_kernels "" env.pfor_kernels ;;

let _ =
    let lexbuf = Lexing.from_channel stdin in
    let tree = try
        Parser.top_level Scanner.token lexbuf
    with except ->
        let curr = lexbuf.Lexing.lex_curr_p in
        let line = curr.Lexing.pos_lnum in
        let col = curr.Lexing.pos_cnum in
        let tok = Lexing.lexeme lexbuf in
        raise (Syntax_error (line, col, tok))
    in
    let code, env = generate_toplevel tree in
```

```
    let forward_declarations = generate_device_forward_declarations env in
    let kernel_invocations = generate_kernel_invocation_functions env in
    let kernel_functions  = generate_kernel_functions env in
    let pfor_kernels = generate_pfor_kernels env in
    let header =  "#include <stdio.h>\n\
                   #include <stdlib.h>\n\
                   #include <stdint.h>\n\
                   #include <libvector.hpp>\n\n" in
    print_string header;
    print_string forward_declarations;
    print_string kernel_functions;
    print_string kernel_invocations;
    print_string pfor_kernels;
    print_string code
```

A.8 rtlib/libvector.hpp

```
#ifndef __LIBVECTOR_H__
#define __LIBVECTOR_H__

#include "vector_array.hpp"
#include "vector_utils.hpp"
#include "vector_iter.hpp"
#include <cuComplex.h>
#endif
```

A.9 rtlib/vector_array.hpp

```
#ifndef __VECTOR_ARRAY_H__
#define __VECTOR_ARRAY_H__

#include <stdlib.h>
#include <stdarg.h>
#include "vector_utils.hpp"

using namespace std;

struct array_ctrl {
    int refcount;
    char h_dirty;
    char d_dirty;
};

template <class T>
struct device_info {
    T *values;
    size_t *dims;
    size_t ndims;
};

template <class T>
__device__ size_t get_mid_index(struct device_info<T> *info, size_t ind, size_t dim)
{
```

```
    size_t stride = 1;
    size_t i;

    for (i = dim + 1; i < info->ndims; i++)
        stride *= info->dims[i];

    return ind * stride;
}

template <class T>
class VectorArray {
    private:
        T *values;
        T *d_values;
        size_t ndims;
        size_t *dims;
        size_t *d_dims;
        size_t nelems;
        struct array_ctrl *ctrl;
        struct device_info<T> *dev_info;
        size_t bsize();
        void incRef();
        void decRef();
    public:
        VectorArray();
        VectorArray(size_t ndims, ...);
        VectorArray(const VectorArray<T> &orig);
        VectorArray<T> dim_copy(void);
                T &oned_elem(size_t ind);
        T &elem(bool modify, size_t first_ind, ...);
        VectorArray<T> &chain_set(size_t ind, T val);
        VectorArray<T>& operator= (const VectorArray<T> &orig);
        ~VectorArray();
        size_t size();
        size_t length(size_t dim = 0);
        void copyToDevice(size_t n = 0);
        void copyFromDevice(size_t n = 0);
        T *devPtr();
        struct device_info<T> *devInfo();
        void markDeviceDirty(void);
};

template <class T>
VectorArray<T>::VectorArray()
{
    this->ndims = 0;
    this->dims = NULL;
    this->d_dims = NULL;
    this->values = NULL;
    this->d_values = NULL;
    this->dev_info = NULL;
    this->nelems = 0;
    this->ctrl = (struct array_ctrl *) malloc(sizeof(struct array_ctrl));
    this->ctrl->refcount = 1;
```

```
    this->ctrl->h_dirty = 0;
    this->ctrl->d_dirty = 0;
}

template <class T>
VectorArray<T>::VectorArray(size_t ndims, ...)
{
    size_t i;
    va_list dim_list;
    cudaError_t err;
    struct device_info<T> h_dev_info;

    va_start(dim_list, ndims);

    this->ndims = ndims;
    this->dims = (size_t *) calloc(ndims, sizeof(size_t));
    this->nelems = 1;
    this->ctrl = (struct array_ctrl *) malloc(sizeof(struct array_ctrl));
    this->ctrl->refcount = 1;
    this->ctrl->h_dirty = 0;
    this->ctrl->d_dirty = 0;

    for (i = 0; i < ndims; i++) {
        this->dims[i] = va_arg(dim_list, size_t);
        this->nelems *= this->dims[i];
    }

    va_end(dim_list);

    this->values = (T *) calloc(this->nelems, sizeof(T));
    err = cudaMalloc(&this->d_values, bsize());
    checkError(err);

    err = cudaMalloc(&this->d_dims, sizeof(size_t) * ndims);
    checkError(err);
    err = cudaMemcpy(this->d_dims, this->dims, sizeof(size_t) * ndims,
            cudaMemcpyHostToDevice);
    checkError(err);

    h_dev_info.ndims = ndims;
    h_dev_info.dims = this->d_dims;
    h_dev_info.values = this->d_values;
    err = cudaMalloc(&this->dev_info, sizeof(h_dev_info));
    checkError(err);
    err = cudaMemcpy(this->dev_info, &h_dev_info, sizeof(h_dev_info),
            cudaMemcpyHostToDevice);
    checkError(err);
}

template <class T>
VectorArray<T>::VectorArray(const VectorArray<T> &orig)
{
    this->ndims = orig.ndims;
    this->dims = orig.dims;
```

```
    this->d_dims = orig.d_dims;
    this->nelems = orig.nelems;
    this->ctrl = orig.ctrl;
    this->values = orig.values;
    this->d_values = orig.d_values;
    this->dev_info = orig.dev_info;

    incRef();
}

template <class T>
VectorArray<T> VectorArray<T>::dim_copy(void)
{
    VectorArray<T> copy;
    cudaError_t err;

    copy.ndims = this->ndims;
    copy.dims = (size_t *) calloc(copy.ndims, sizeof(size_t));

    for (int i = 0; i < this->ndims; i++)
        copy.dims[i] = this->dims[i];

    copy.nelems = this->nelems;
    copy.values = (T *) calloc(this->nelems, sizeof(T));
    err = cudaMalloc(&copy.d_values, copy.bsize());
    checkError(err);

    return copy;
}

template <class T>
VectorArray<T>& VectorArray<T>::operator= (const VectorArray<T>& orig)
{
    // avoid self-assignment
    if (this == &orig)
        return *this;

    decRef();

    this->ndims = orig.ndims;
    this->dims = orig.dims;
    this->d_dims = orig.d_dims;
    this->nelems = orig.nelems;
    this->ctrl = orig.ctrl;
    this->values = orig.values;
    this->d_values = orig.d_values;
    this->dev_info = orig.dev_info;

    incRef();

    return *this;
}

template <class T>
```

```
void VectorArray<T>::incRef(void)
{
    this->ctrl->refcount++;
}

template <class T>
void VectorArray<T>::decRef(void)
{
    if (--(this->ctrl->refcount) > 0)
        return;
    free(this->ctrl);
    if (this->dims != NULL)
        free(this->dims);
    if (this->d_dims != NULL)
        cudaFree(this->d_dims);
    if (this->values != NULL)
        free(this->values);
    if (this->d_values != NULL)
        cudaFree(this->d_values);
    if (this->dev_info != NULL)
        cudaFree(this->dev_info);
}

template <class T>
T &VectorArray<T>::oned_elem(size_t ind)
{
    if (this->ctrl->d_dirty)
        copyFromDevice();

    this->ctrl->h_dirty = 1;
    return this->values[ind];
}

template <class T>
T &VectorArray<T>::elem(bool modify, size_t first_ind, ...)
{
    size_t ind = first_ind, onedind = first_ind;
    int i;
    va_list indices;

    if (this->ctrl->d_dirty)
        copyFromDevice();

    if (modify)
        this->ctrl->h_dirty = 1;

    va_start(indices, first_ind);

    for (i = 1; i < this->ndims; i++) {
        ind = va_arg(indices, size_t);
        onedind = onedind * this->dims[i] + ind;
    }

    va_end(indices);
```

```
    return this->values[onedind];
}

template <class T>
VectorArray<T>::~VectorArray()
{
    decRef();
}

template <class T>
VectorArray<T>& VectorArray<T>::chain_set(size_t ind, T value)
{
    oned_elem(ind) = value;
    return *this;
}

template <class T>
size_t VectorArray<T>::size()
{
    return this->nelems;
}

template <class T>
size_t VectorArray<T>::bsize()
{
    return sizeof(T) * this->nelems;
}

template <class T>
size_t VectorArray<T>::length(size_t dim)
{
    return this->dims[dim];
}

template <class T>
void VectorArray<T>::copyToDevice(size_t n)
{
        if (n == 0)
            n = size();
    cudaError_t err;
    err = cudaMemcpy(this->d_values, this->values,
            sizeof(T) * n, cudaMemcpyHostToDevice);
    checkError(err);
    this->ctrl->h_dirty = 0;
}

template <class T>
void VectorArray<T>::copyFromDevice(size_t n)
{
    cudaError_t err;
        if (n == 0)
            n = size();
        err = cudaMemcpy(this->values, this->d_values,
```

```
                        sizeof(T) * n, cudaMemcpyDeviceToHost);
    checkError(err);
    this->ctrl->d_dirty = 0;
}

template <class T>
T *VectorArray<T>::devPtr()
{
    if (this->ctrl->h_dirty)
        copyToDevice();
    return this->d_values;
}

template <class T>
struct device_info<T> *VectorArray<T>::devInfo()
{
    devPtr();
    return this->dev_info;
}

template <class T>
void VectorArray<T>::markDeviceDirty(void)
{
    this->ctrl->d_dirty = 1;
}

#endif
```

A.10 rtlib/vector_utils.hpp

```
#ifndef __VECTOR_UTILS_H__
#define __VECTOR_UTILS_H__

#include <sys/time.h>

static inline void _check(cudaError_t err, const char *file, int line)
{
    if (err != cudaSuccess) {
        fprintf(stderr, "CUDA error at %s:%d\n", file, line);
        fprintf(stderr, "%s\n", cudaGetErrorString(err));
        exit(err);
    }
}

static inline double get_time(void)
{
    struct timeval tv;

    gettimeofday(&tv, NULL);

    return (double) tv.tv_sec + ((double) tv.tv_usec) / 1000000.0;
}
```

```
#ifndef BLOCK_SIZE
#define BLOCK_SIZE 1024
#endif
#define checkError(err) _check((err), __FILE__, __LINE__)
#define ceil_div(n, d) (((n) - 1) / (d) + 1)
#define min(a, b) (((a) < (b)) ? (a) : (b))
#define max(a, b) (((a) > (b)) ? (a) : (b))

#endif
```

A.11 rtlib/vector_iter.hpp

```
#ifndef __VECTOR_ITER_H__
#define __VECTOR_ITER_H__

#include "vector_utils.hpp"

struct range_iter {
    size_t start;
    size_t stop;
    size_t inc;
    size_t len;
    size_t mod;
    size_t div;
};

void fillin_iters(struct range_iter *iters, size_t n)
{
    int i;
    size_t last_mod = 1;

    for (i = n - 1; i >= 0; i--) {
        iters[i].len = ceil_div(iters[i].stop - iters[i].start,
                    iters[i].inc);
        iters[i].div = last_mod;
        iters[i].mod = last_mod * iters[i].len;
        last_mod = iters[i].mod;
    }
}

inline size_t get_index_cpu(struct range_iter *iter, size_t oned_ind)
{
    return iter->start + (oned_ind % iter->mod) / iter->div * iter->inc;
}

__device__ size_t get_index_gpu(struct range_iter *iter, size_t oned_ind)
{
    return iter->start + (oned_ind % iter->mod) / iter->div * iter->inc;
}

size_t total_iterations(struct range_iter *iter, size_t n)
{
    int total = 1;
```

```
    size_t i;

    for (i = 0; i < n; i++)
        total *= iter[i].len;

    return total;
}

struct range_iter *device_iter(struct range_iter *iters, size_t n)
{
    cudaError_t err;
    struct range_iter *d_iters;

    err = cudaMalloc(&d_iters, n * sizeof(struct range_iter));
    checkError(err);
    err = cudaMemcpy(d_iters, iters, n * sizeof(struct range_iter),
            cudaMemcpyHostToDevice);
    checkError(err);

    return d_iters;
}

#endif
```

A.12 test/arrays.vec

```
int[] gen2d_array()
{
    int arr1[2, 2];

    arr1[0, 0] = 0;
    arr1[0, 1] = 1;
    arr1[1, 0] = 2;
    arr1[1, 1] = 3;

    return arr1;
}

void print_array(int8 arr[])
{
    for (num in arr)
        printf("%d\n", num);
}

int vec_main()
{
    arr1 := gen2d_array();
    int i;
    int j;

    i = 0;
    while (i < 2) {
        j = 0;
```

```
        while (j < 2) {
            printf("%d\n", arr1[i, j]);
            j++;
        }
        i++;
    }

    arr2 := {int8(arr1[0, 0]), int8(arr1[0,1])};
    print_array(arr2);

    return 0;
}
```

A.13 test/control_flow.vec

```
int vec_main()
{
    i := 0;
    arr := {0, 1, 2};

    while (i < 10) {
        if (i % 2 == 0)
            printf("%d\n", i);
        i++;
    }

    for (i in 0:5:2, j in 0:2)
        printf("%d %d\n", i, j);

    for (x in arr)
        printf("%d\n", x);

    return 0;
}
```

A.14 test/float.vec

```
double divide(double a, double b)
{
    return a/b;
}

int vec_main()
{
    j := divide(3., 4.);

    printf("%0.2f\n", abs(j));

    return 0;
}
```

A.15 test/hello.vec

```
int vec_main()
{
    printf("Hello, Vector!\n");
    return 0;
}
```

A.16 test/length.vec

```
int vec_main() {
    a := {1,2,3,4};
    b := len(a);

    c := len("length test");

    int d[3, 2];

    e := len(d, 1);

    printf("%d %d %d\n", b, c, e);

    return 0;
}
```

A.17 test/map.vec

```
__device__ float square(float x) {
 return x * x;
}

int vec_main()
{
    float inputs[1024, 3];
    int check;
    int expected;

    for (i in 0:1024, j in 0:3)
        inputs[i, j] = float(i * 3 + j);

    squares := @map(square, inputs);

    if (len(squares, 0) != 1024 || len(squares, 1) != 3) {
        printf("dimensions incorrect\n");
        return -1;
    }

    for (i in 0:1024, j in 0:3) {
        check = int(squares[i, j]);
        expected = i * 3 + j;
        expected = expected * expected;
        if (check != expected) {
            printf("incorrect\n");
```

```
            return -1;
        }
    }

    printf("correct\n");
    return 0;
}
```

A.18 test/reduce.vec

```
__device__ int add(int x, int y){
  return x + y;
}

int vec_main()
{
    int arr[1900];
    reference := 0;

    for (i in 0:1900)
        arr[i] = random();

    for (i in 0:1900)
        reference += arr[i];

    sum := @reduce(add, arr);

    if (sum == reference)
        printf("correct\n");
    else
        printf("incorrect\n");

    return 0;
}
```

A.19 test/time.vec

```
int vec_main()
{
    time1 := time();
    useless := 0;

    // add some arbitrary delay
    for (i in 0:100000)
        useless ^= (i | random());

    time2 := time();

    assert(time2 - time1 > 0.0);

    printf("correct\n");
```

```
    return 0;
}
```

A.20 test/complex.vec

```
complex128 add(complex128 a, complex128 b)
{
    return a + b;
}

int vec_main()
{
    i := #(float32(4.5), float32(2.3)) * #(float32(0.), float32(2.));
    j := add(#(3.,4.),#(5., 4.));

    k := #(float32(4.5), float32(2.3));

    k.im = float32(5);
    k += #(float32(0), float32(5));
    printf("%0.1f + %0.1fj\n", i.re, i.im);
    printf("%0.1f + %0.1fj\n", j.re, j.im);
    printf("%0.1f + %0.1fj\n", k.re, k.im);
    printf("%0.1f\n", abs(j));

    return 0;
}
```

A.21 test/dotprod.vec

```
__device__ int add(int a, int b)
{
    return a + b;
}

int vec_main()
{
    int x[2048];
    int y[2048];
    int prod[2048];
    expected := 0;

    for (i in 0:2048) {
        x[i] = random();
        y[i] = random();
    }

    pfor (i in 0:2048)
        prod[i] = x[i] * y[i];

    dp := @reduce(add, prod);

    for (i in 0:2048)
        expected += (x[i] * y[i]);
```

```
    if (dp == expected)
        printf("correct\n");
    else printf("incorrect\n");

    return 0;
}
```

A.22 test/functions.vec

```
void print_int(int d)
{
    printf("%d\n", d);
}

int add(int a, int b)
{
    return a + b;
}

int vec_main()
{
    j := add(3, 4);

    print_int(j);

    {
        d := j;
        j := d + 3;
        print_int(j);
    }

    return 0;
}
```

A.23 test/inline.vec

```
int vec_main()
{
    length := 10;
    value := 5;

    inline("int *array = (int *) malloc(length * sizeof(int));");
    for (i in 0:length) {
        inline("array[i] = value;");
    }
    for (i in 0:length) {
        int j;
        inline("j = array[i];");
        printf("%d\n", j);
    }
    inline("free(array)");
```

```
    return 0;
}
```

A.24 test/logic.vec

```
int vec_main()
{
    assert(3 == 3);
    assert(1 || 0);
    assert(1 && 1);
    assert(!(0 && 1));
    assert(3 < 5);
    assert(4 > 1);
    assert(2 <= 2);
    assert(3 >= 1);

    printf("All tests passed\n");

    return 0;
}
```

A.25 test/pfor.vec

```
int vec_main()
{
    int arr[1000, 2];
    scale := 2;

    pfor (i in 0:len(arr, 0), j in 0:len(arr, 1))
        arr[i, j] = 2 * i + j;

    pfor (i in 0:len(arr, 0), j in 0:len(arr, 1))
        arr[i, j] = scale * arr[i, j];

    for (i in 0:len(arr, 0), j in 0:len(arr, 1)) {
        if (arr[i, j] != (4 * i + 2 * j)) {
            printf("incorrect\n");
            return -1;
        }
    }

    printf("correct\n");

    return 0;
}
```

A.26 test/strings.vec

```
int vec_main()
{
    key := "answer";
    value := 42;
```

```
    printf("%s: %d\n", key, value);

    return 0;
}
```

A.27 bench/mandelbrot-bench.vec

```
left := float(-2.0);
right := float(1.0);
top := float(1.0);
bottom := float(-1.0);
ntrials := 3;

int mandelbrot_cpu(int xi, int yi, int xn, int yn,
    float left, float right, float top, float bottom)
{
    iter := 0;

    x0 := left + (right - left) / float(xn) * float(xi);
    y0 := bottom + (top - bottom) / float(yn) * float(yi);
    z0 := #(x0, y0);
    z := #(float(0), float(0));

    while (iter < 256 && abs(z) < 2) {
        z = z * z + z0;
        iter++;
    }

    return iter;
}

__device__ int mandelbrot_gpu(int xi, int yi, int xn, int yn,
    float left, float right, float top, float bottom)
{
    iter := 0;

    x0 := left + (right - left) / float(xn) * float(xi);
    y0 := bottom + (top - bottom) / float(yn) * float(yi);
    z0 := #(x0, y0);
    z := #(float(0), float(0));

    while (iter < 256 && abs(z) < 2) {
        z = z * z + z0;
        iter++;
    }

    return iter;
}

void bench_cpu(int img_width, int img_height, int ntrials)
{
    int shades[img_height, img_width];
```

```
    float64 runtimes[ntrials];
    float64 start;
    float64 finish;

    for (trial in :ntrials)
    {
        start = time();
        for (yi in 0:img_height, xi in 0:img_width) {
            shades[yi, xi] = mandelbrot_cpu(xi, yi, img_width, img_height,
                                left, right, top, bottom);
        }
        finish = time();
        runtimes[trial] = finish - start;
    }

    printf("CPU %dx%d: ", img_width, img_height);

    for (rt in runtimes)
        printf("%.2f ", rt);
    printf("\n");
}

void bench_gpu(int img_width, int img_height, int ntrials)
{
    int shades[img_height, img_width];
    float64 runtimes[ntrials];
    float64 start;
    float64 finish;

    for (trial in :ntrials)
    {
        start = time();
        pfor (yi in 0:img_height, xi in 0:img_width) {
            shades[yi, xi] = mandelbrot_gpu(xi, yi, img_width, img_height,
                                left, right, top, bottom);
        }
        finish = time();
        runtimes[trial] = finish - start;
    }

    printf("GPU %dx%d: ", img_width, img_height);

    for (rt in runtimes)
        printf("%.2f ", rt);
    printf("\n");
}

int vec_main()
{
    img_heights := {480, 600, 768, 864, 960};
    img_widths := {640, 800, 1024, 1152, 1280};

    printf("Benchmarking CPU\n");
    for (i in 0:len(img_heights))
```

```
        bench_cpu(img_widths[i], img_heights[i], ntrials);

    printf("Benchmarking GPU\n");
    for (i in 0:len(img_heights))
        bench_gpu(img_widths[i], img_heights[i], ntrials);

    return 0;
}
```

A.28 bench/mandelbrot-cpu.vec

```
int mandelbrot(int xi, int yi, int xn, int yn,
    float left, float right, float top, float bottom)
{
    iter := 0;

    x0 := left + (right - left) / float(xn) * float(xi);
    y0 := bottom + (top - bottom) / float(yn) * float(yi);
    z0 := #(x0, y0);
    z := #(float(0), float(0));

    while (iter < 256 && abs(z) < 2) {
        z = z * z + z0;
        iter++;
    }

    return iter;
}

void print_pgm(int shades[], int width, int height)
{
    printf("P2\n");
    printf("%d %d\n", width, height);
    printf("255\n");

    for (y in 0:height) {
        for (x in 0:width)
            printf("%d ", shades[y, x]);
        printf("\n");
    }
}

int vec_main()
{
    img_height := 512;
    img_width := 768;

    int shades[img_height, img_width];

    left := float(-2.0);
    right := float(1.0);
    top := float(1.0);
    bottom := float(-1.0);
```

```
    for (yi in 0:img_height, xi in 0:img_width) {
        shades[yi, xi] = mandelbrot(xi, yi, img_width, img_height,
                            left, right, top, bottom);
    }

    print_pgm(shades, img_width, img_height);

    return 0;
}
```

A.29 bench/mandelbrot-gpu.vec

```
__device__ int mandelbrot(int xi, int yi, int xn, int yn,
    float left, float right, float top, float bottom)
{
    iter := 0;

    x0 := left + (right - left) / float(xn) * float(xi);
    y0 := bottom + (top - bottom) / float(yn) * float(yi);
    z0 := #(x0, y0);
    z := #(float(0), float(0));

    while (iter < 256 && abs(z) < 2) {
        z = z * z + z0;
        iter++;
    }

    return iter;
}

void print_pgm(int shades[], int width, int height)
{
    printf("P2\n");
    printf("%d %d\n", width, height);
    printf("255\n");

    for (y in 0:height) {
        for (x in 0:width)
            printf("%d ", shades[y, x]);
        printf("\n");
    }
}

int vec_main()
{
    img_height := 512;
    img_width := 768;

    int shades[img_height, img_width];

    left := float(-2.0);
```

```
    right := float(1.0);
    top := float(1.0);
    bottom := float(-1.0);


    pfor (yi in 0:img_height, xi in 0:img_width) {
        shades[yi, xi] = mandelbrot(xi, yi, img_width, img_height,
                                    left, right, top, bottom);
    }

    print_pgm(shades, img_width, img_height);

    return 0;
}
```

A.30 compiler/SConscript

```
import subprocess as sp

Import('env')

AddOption('--ocamlc', dest='ocamlc', default='ocamlc',
          help = 'Ocaml compiler to use')
AddOption('--ocaml_flags', dest='ocaml_flags', default='-I compiler',
          help = 'Ocaml compiler flags')
AddOption('--native', dest='native', default=False, action='store_true',
          help = 'Compile native code instead of bytecode')

obj_ext = '.cmo'
ocamlc = GetOption('ocamlc')
ocaml_flags = GetOption('ocaml_flags')

if GetOption('native'):
    ocamlc = 'ocamlopt'
    obj_ext = '.cmx'

ocaml_object = Builder(action = '$OCAMLC -c $OCAML_FLAGS $SOURCE')
ocaml_program = Builder(action = '$OCAMLC $OCAML_FLAGS $SOURCES -o $TARGET')
ocamllex = Builder(action = 'ocamllex $SOURCE')
ocamlyacc = Builder(action = 'ocamlyacc $SOURCE')
env.Append(OCAMLC = ocamlc, OCAML_FLAGS = ocaml_flags)
env.Append(BUILDERS = {'OcamlObject': ocaml_object,
                       'OcamlProgram': ocaml_program,
                       'Ocamllex': ocamllex,
                       'Ocamlyacc': ocamlyacc})

parsers = ['parser']
lexers = ['scanner']
source_files = ['symgen', 'ast', 'parser', 'scanner',
                'environment', 'detect', 'generator']

for name in parsers:
    env.Ocamlyacc([name + '.ml', name + '.mli'], name + '.mly')
```

```
    env.OcamlObject(name + '.cmi', name + '.mli')

for name in lexers:
    env.Ocamllex(name + '.ml', name + '.mll')

for name in source_files:
    if name in parsers:
        target = name + obj_ext
        deps = [name + '.ml', name + '.cmi']
    else:
        target = [name + '.cmi', name + obj_ext]
        deps = name + '.ml'
    env.OcamlObject(target, deps)

env.Depends('ast' + obj_ext, 'symgen.cmi')
env.Depends('environment' + obj_ext, 'ast.cmi')
env.Depends('parser' + obj_ext, 'ast.cmi')
env.Depends('scanner' + obj_ext, 'parser.cmi')
env.Depends('detect' + obj_ext, ['environment.cmi', 'ast.cmi'])
env.Depends('generator' + obj_ext, ['environment.cmi', 'parser.cmi',
                                    'ast.cmi', 'scanner.cmi'])

env.OcamlProgram('generator',
        [source_file + obj_ext for source_file in source_files])
```

A.31 test/SConscript

```
Import('env')

cuda_gen = Builder(action='./compiler/generator < $SOURCE > $TARGET')
cuda_object = Builder(action='$NVCC -c $NVCC_FLAGS $SOURCE -o $TARGET')
run_test = Builder(action='./$SOURCE | diff ${SOURCE}.out -')

AddOption('--rtlib', dest='rtlib', default='ocelot',
            help='CUDA runtime library (ocelot or cudart)')
AddOption('--cuda_arch', dest='cuda_arch', default='sm_20',
            help='CUDA architecture')
AddOption('--cuda_lib', dest='cuda_lib', default='/usr/local/cuda/lib',
            help='Cuda lib directory')
AddOption('--8400gs', dest='8400gs',
            default=False, action="store_true",
            help="Build for the GeForce 8400 GS")

rtlib = GetOption('rtlib')
cuda_lib = GetOption('cuda_lib')
cuda_arch = GetOption('cuda_arch')
ccflags = "-Wall -g"

# This is just a convenience for me to quickly choose the options for my GPU
# If you're using ocelot, don't use this flag
if GetOption('8400gs'):
    rtlib = 'cudart'
    cuda_lib = '/opt/cuda/lib64'
```

```
    cuda_arch = 'sm_11'
    ccflags += " -DBLOCK_SIZE=256"

env['LINK'] = env['CXX']
env.Append(LIBS = [rtlib, 'm'])
env.Append(LIBPATH = [cuda_lib])
env.Append(CPPPATH = ['./rtlib'])
env.Append(CUDA_ARCH = cuda_arch)
env.Append(CCFLAGS = ccflags)
env.Append(NVCC = 'nvcc', NVCC_FLAGS = '-arch=$CUDA_ARCH -I$CPPPATH -Xcompiler $CCFLAGS')
env.Append(BUILDERS = {'CudaGen' : cuda_gen,
                       'CudaObject' : cuda_object,
                       'RunTest': run_test})

# Add the name of your example to this list to get it compiled
test_cases = ['hello', 'reduce', 'control_flow', 'arrays', 'functions', 'float',
              'complex', 'map', 'logic', 'inline', 'strings', 'length', 'pfor',
              'dotprod', 'time']
headers = ['libvector', 'vector_utils', 'vector_array', 'vector_iter']

for test_case in test_cases:
    env.CudaGen(test_case + '.cu', [test_case + '.vec'])
    env.CudaObject(test_case + '.o', [test_case + '.cu'] +
        ['../rtlib/' + h + '.hpp' for h in headers])
    env.Program(test_case, test_case + '.o')
    env.RunTest(test_case + '_test', test_case)
```

A.32 bench/SConscript

```
Import('env')

programs = ['mandelbrot-bench', 'mandelbrot-cpu', 'mandelbrot-gpu']
headers = ['libvector', 'vector_utils', 'vector_array', 'vector_iter']

for prog in programs:
    env.CudaGen(prog + '.cu', [prog + '.vec'])
    env.CudaObject(prog + '.o', [prog + '.cu'] +
        ['../rtlib/' + h + '.hpp' for h in headers])
    env.Program(prog, prog + '.o')
```

A.33 SConstruct

```
import os

env = Environment(ENV=os.environ)
Export('env')

env.SConscript(['compiler/SConscript'])
env.SConscript(['test/SConscript'])
env.SConscript(['bench/SConscript'])
env.SConscript(['docs/SConscript'])
Default('compiler')
```