Vector: A High-Level Programming Language for GPU Computing

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

The Problem

- GPUs have gained the ability to perform general-purpose computing tasks, so-called GPGPU
- GPGPU now the workhorse of High-Performance Computing
- Current GPGPU languages, CUDA and OpenCL, not very beginner-friendly and operate at low level of abstraction
  - Explicit copying of memory to and from GPU
  - Explicit choice of warp size
- GPU programming often follows common patterns, like map or reduce, but with no first-class functions, no way to implement patterns in reusable way

The Solution: Vector

- Memory implicitly copied to and from GPU on *ad-hoc* basis
- Automatic warp size selection
- Lightweight parallel-for syntax instead of defining kernels
- Map and Reduce implemented as higher order functions
- Compiles to CUDA

Syntax

- Mostly C-like syntax
- Extensions for GPU computing and some syntactic sugar

Arrays

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

x := a[i, j, k];

a[i, j, k] = x;
```

- Support for n-dimensional arrays
- Arrays created on both CPU and GPU
- Arrays are reference counted
- Data automatically copied to GPU if accessed in GPU statements
- Automatically copied back to CPU if accessed in CPU code

For and Parallel For (pfor)

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

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


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

- For loop uses iterator statements instead of explicit incrementing as in C, so “i=0; i<5; i+=2” becomes “i in 0:5:2”
- Pfor loop uses same syntax, but each iteration run in separate thread on GPU
- For loop also supports “for each” type syntax. Iterate over elements of array

Map and Reduce

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

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

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

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

- Higher order functions
- Must be generated at compile-time (function pointers not guaranteed to work in CUDA)
- Map takes function *f* and array *a*, returns array *b* where *b[i] = f(a[i])*
- Reduce takes function *f* and array *a*, returns the result of applying f to two pairs of elements in *a*, then applying it to pairs of the results, etc. The function f must be associative and commutative

Implementation Details

- Scanner/Parser in Ocamllex and Ocamlyacc
- Generator takes AST and produces CPU code inline
- Generation of GPU code is deferred until end
- Environment stores variables in scope and other state
- Runtime library implements arrays and iterators

Lessons Learned

- Group dynamics is important - good balance between leader and team members
- 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.
- Communication with teammates is very important. Enforcing a consistent coding style (especially with respect to indentation) will avoid problems down the line.
- OCaml tools (and the functional programming paradigm in general) are really great for writing compilers.
- Start early

And Now a Demo!!!

Mandelbrot set generator on CPU and GPU

CPU vs GPU performance