Vector: A High-Level Programming Language for the GPU

Howard Mao (zm2169@columbia.edu), Zachary Newman (zjn2101@columbia.edu), Sidharth Shanker (sps2133@columbia.edu), Jonathan Yu (jy2432@columbia.edu)

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 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.

We propose *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 choose to generate CUDA rather than the more platform-independent OpenCL due to our greater familiarity with CUDA. Also, most HPC systems use Nvidia Tesla GPUs. For development, we will test using GPU Ocelot, an emulator for Nvidia GPUs that runs PTX (the intermediate representation for CUDA) on the CPU.

Language Features

- basic type inference for assignments
- parallel for (`pfor`) loops: generate a CUDA `kernel` (or multiple `kernel`s) for CUDA to compile into PTX and run on the GPU on lots of data
- 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 (unless we compile kernels dynamically)
- anonymous functions
- easy FFI: call C functions using the same syntax as for Vector functions

- 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)

Sample Program

```
/*
 * Compute the dot product of two vectors
 */
int main(void) {
    x := int[]{1,2,3,4}
    y := int[]{3,5,7,9}
    int z[len(x)]
    pfor i in 0:len(x) {
        z[i] = x[i] * y[i]
        // each iteration of the for loop runs on a different thread in the
        // GPU; sync() pauses execution of the thread until all other
        // threads reach the same point
        sync()
        for s := 1; s < len(x); s*=2 {
            if i % (2 * s) == 0 {
                z[i] *= z[i+s]
            }
            sync()
        }
    }
    dp := z[0]
}
```

or, using language built-ins map/reduce (these need to happen at compile time because we have no GPU support for first-class functions at runtime):

```
/*
 * Compute the dot product of two vectors
 */
int main(void) {
    x := int[]{1,2,3,4}
    y := int[]{3,5,7,9}
    z := map((*), x, y)
    dp := reduce((+), z)
}
```

The equivalent CUDA program would look like this.

```
#include <stdlib.h>
#include <stdio.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);
    }
}

#define checkError(err) _check((err), __FILE__, __LINE__)

__global__
void dotprod_kernel(int *dotprod, int *x, int *y, size_t n)
{
    size_t i = threadIdx.x + blockDim.x * blockIdx.x;
    size_t s;

    // map
    if (i < n)
        dotprod[i] = x[i] * y[i];

    // reduce
    for (s = 1; s < n; s *= 2) {
        if (i % (2 * s) == 0 && i + s < n)
            dotprod[i] += dotprod[i + s];
        __syncthreads();
    }
}

int main(void)
{
    int x[4] = {1, 2, 3, 4};
    int y[4] = {3, 5, 7, 9};
    int dp;

    int *d_x, *d_y, *d_dp;

    cudaError_t err;

    // allocate GPU memory
    err = cudaMalloc(&d_x, sizeof(x));
    checkError(err);
    err = cudaMalloc(&d_y, sizeof(y));
```

```
    checkError(err);
    err = cudaMalloc(&d_dp, sizeof(x));
    checkError(err);

    // copy over x and y
    err = cudaMemcpy(d_x, x, sizeof(x), cudaMemcpyHostToDevice);
    checkError(err);
    err = cudaMemcpy(d_y, y, sizeof(y), cudaMemcpyHostToDevice);
    checkError(err);

    // launch the kernel
    dotprod_kernel<<<1, 4>>>(d_dp, d_x, d_y, 4);
    cudaDeviceSynchronize();
    checkError(cudaGetLastError());

    // copy the result into dp
    cudaMemcpy(&dp, d_dp, sizeof(dp), cudaMemcpyDeviceToHost);

    cudaFree(d_x);
    cudaFree(d_y);
    cudaFree(d_dp);

    return 0;
}
```

References

[1] GPU Ocelot project: http://gpuocelot.gatech.edu/