A Refactoring Tool to Extract GPU Kernels

Kosta Damevski

Department of Mathematics and Computer Science
Virginia State University

May 21, 2011
Some Computationally Intensive Tasks are Better Done on GPUs

- An order of magnitude increase in number of cores
  - 448 cores in NVIDIA Tesla C2070 GPU
- Slightly better memory latency compared to CPU
- Widely available
  - GPUs are already exist in many machines
- However, potential bottleneck in using GPUs: PCIe bus bandwidth (a few GB/s)
GPU Programming Has Gone Mainstream

A lot of novel uses have been proposed for GPUs

- Many scientific applications
  - The #1 supercomputer on the Top500 list uses NVIDIA GPUs as accelerators

- Virus protection (Kaspersky Lab)
- GPU accelerated software router (Han et al., SIGComm ’10)

Due, in large part, to the programmability of new generations of GPUs

- OpenCL
- NVIDIA’s Compute Unified Device Architecture (CUDA)
NVIDIA’s CUDA Programming Model

- Extension of C
  - Data parallel functions called *kernels*, executed with large numbers of threads
  - *cudaMemcpy*, *cudaMalloc* and *cudaFree* functions for GPU memory manipulation

- Typical CUDA workflow:
  1. CPU: Allocate and copy data to GPU memory
  2. CPU: Specify number of threads, launch GPU kernel
  3. GPU: Execute the kernel
  4. CPU: Copy results back from the GPU
  5. CPU: Free memory on the GPU
GPU Programming is Still Tedious

- Memory allocation and copying is error prone
  - Some errors can be hard to diagnose
- Special function calls are required to detect problems
  - e.g. Launching too many threads does not cause an error by default, just no computation
- And, parallelization is not easy either
A Refactoring Tool to Extract GPU Kernels

Kosta Damevski

GPU Programming Overview

Extract Kernel Refactoring

Future Plans and Work

A Simple AXPY loop

```c
for(i=0; i<N; i++) {
    y[i] = a*x[i] + y[i];
}
```
A Refactoring Tool to Extract GPU Kernels

Kosta Damevski

GPU Programming Overview

Extract Kernel Refactoring

Future Plans and Work

AXPY GPU kernel

```c
__global__ void gpu_axpy(float *y, float a, float *x) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    y[idx] = a * x[idx] + y[idx];
}

int main() {
    //variable declarations omitted

    float *y_d;
    cudaMalloc((void **) &y_d, sizeof(float)*N);
    cudaMemcpy(y_d, y, sizeof(float)*N, cudaMemcpyHostToDevice);
    float *x_d;
    cudaMalloc((void **) &x_d, sizeof(float)*N);
    cudaMemcpy(x_d, x, sizeof(float)*N, cudaMemcpyHostToDevice);

    gpu_axpy<<<1,N>>>(y_d, a, x_d);

    cudaMemcpy(y, y_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
    cudaFree(y_d);
    cudaMemcpy(x, x_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
    cudaFree(x_d);
}
```
AXPY GPU kernel

```c
__global__ void gpu_axpy(float *y, float a, float *x) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    y[idx] = a * x[idx] + y[idx];
}

int main() {
    //variable declarations omitted

    float *y_d;
    cudaMalloc((void **) &y_d, sizeof(float)*N);
    cudaMemcpy(y_d, y, sizeof(float)*N, cudaMemcpyHostToDevice);
    float *x_d;
    cudaMalloc((void **) &x_d, sizeof(float)*N);
    cudaMemcpy(x_d, x, sizeof(float)*N, cudaMemcpyHostToDevice);

    gpu_axpy<<<1,N>>>(y_d, a, x_d);

    cudaMemcpy(y, y_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
    cudaFree(y_d);
    cudaMemcpy(x, x_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
    cudaFree(x_d);
}
```
A Refactoring Tool to Extract GPU Kernels

Kosta Damevski

GPU Programming Overview

Extract Kernel Refactoring

Future Plans and Work

AXPY GPU kernel

```c
__global__ void gpu_axpy(float *y, float a, float *x) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    y[idx] = a * x[idx] + y[idx];
}

int main() {
    //variable declarations omitted

    float *y_d;
    cudaMalloc((void **) &y_d, sizeof(float)*N);
    cudaMemcpy(y_d, y, sizeof(float)*N, cudaMemcpyHostToDevice);
    float *x_d;
    cudaMalloc((void **) &x_d, sizeof(float)*N);
    cudaMemcpy(x_d, x, sizeof(float)*N, cudaMemcpyHostToDevice);

    gpu_axpy«<1,N»>(y_d, a, x_d);

    cudaMemcpy(y, y_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
    cudaFree(y_d);
    cudaMemcpy(x, x_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
    cudaFree(x_d);
}
```
Refactoring Tool Overview

- Refactor loops into GPU kernels
- Only candidate loops that are parallelizable and pass certain preconditions are refactored
Refactoring Preconditions

1. No break or return statements in the candidate loop body
   - Cannot preserve meaning of these statements in a kernel

2. No method calls in the candidate loop body
   - Cannot make function calls from a CUDA kernel to a CPU function

3. Loop does not contain data races, and therefore is safely parallelizable
   - Ensure that data dependences do not exist between loop iterations
   - Large body of work exists on how to do this from the parallelizing compiler community
Process of Generating GPU Kernels

To Generate:

1. `cudaMemcpy`, `cudaMalloc` and `cudaFree` calls ➞
2. Number of threads to instantiate on GPU ➞

Need To Obtain Via Static Analysis:

1. Size of variables (symbolic or exact)
2. Number of iterations in loop (symbolic or exact)

Replace

1. Loop induction variable with the threadID
2. `continue` with `return`
Preliminary Results

Livermore Loops Benchmark

- 24 loops total representing parts of various scientific codes
- 10 successfully refactored
- 14 contained data dependencies and were rejected by the tool
Preliminary Results

Livermore Loops Benchmark

- 24 loops total representing parts of various scientific codes
- 10 successfully refactored
- 14 contained data dependencies and were rejected by the tool

Encouraging results, although the loops were fairly short (only a few lines of code within each)
The Extract Kernel refactoring tool is functional but relatively incomplete. This is a short list of features we would like to implement before distributing the plugin:

- A more complete dependence tester
  - We only consider single induction variable and zero induction variable array indices
- More (complicated) test cases
Other Nice Additions

We are also planning to incorporate most of these:

- Advise the user when the refactoring would yield worse performance (in progress)
  - Measure arithmetic intensity (operations per byte of data) and coalesced memory accesses
- Provide the opposite refactoring
- Handle function calls within the loop (by moving all the functions in the call graph to the GPU) or by inlining them into one method

Thanks to the anonymous reviewers of WRT for some of these ideas!!!
A Refactoring Tool to Extract GPU Kernels

Kosta Damevski

QUESTIONS?

Kosta Damevski
damevski@acm.org

Supported by U.S. Department of Energy and NASA