CUDA and Java

GPU Computing in a Cross Platform Application

Scot Halverson
sah@lanl.gov
What’s the goal?

- Run GPU code alongside Java code
  - Take advantage of high parallelization

- Utilize GPU technology with existing Java code
  - Minimal modifications
  - Cross platform
How?

- Separate processes communicating directly or indirectly
  - Shared memory, file I/O

- Take advantage of Java Native Interface directly

- Use an intermediate library
  - JCUDA, JOCL, etc
  - Provide software interfaces to device

- Use an additional processing/compiling tool
  - Rootbeer, aparapi, etc
  - Converts some Java code to device code
Separate Processes

- **Advantages**
  - Java code is still cross platform

- **Disadvantages**
  - C/CUDA side must be compiled for each platform
  - Interprocess communication can be awkward
Java Native Interface

**Advantages**
- No management of communication between processes
- All code can be contained in one project

**Disadvantages**
- Cross platform capabilities of Java are much more complicated
- Might require JNI code to be written for each platform supported
- Requires embedded C/C++ & CUDA code
Intermediate Library

- **Advantages**
  - Maintains ‘build once, run anywhere’ Java ideal
  - No recompiling for different platforms
  - Any platform that supports Java and CUDA should work

- **Disadvantages**
  - Requires bundled CUDA code
  - Requires bundled library
Additional Processing or Compiling tool

- **Advantages**
  - Write only Java/Java-like code
  - Fewer low-level concerns
    - Memory management, pointers, etc.

- **Disadvantages**
  - Relatively new & experimental
    - Existing support info may not apply
    - Few developers with experience
    - Long term availability
  - Less control
    - Kernels and interfacing code are written by tool
BeSSy – Motivation for CUDA and Java interoperability at LANL D-3

- BeSSy is a sensor siting tool
  - Used to find optimal/near optimal placement of a set of sensors

- Uses computationally expensive algorithms
  - Simulated Annealing
  - Genetic Algorithm (experimental)

- Written in Java

- We’d like to improve on this with GPU computing

- A significant amount of developer time has been spent on existing code
  - Rewriting everything is not an option
Optimization Example
Optimization Example

- same city, same weather, same scenario; only the number of collectors differs
BeSSy – Using Separate Processes

- **Advantages**
  - Code can be distributed and used independently

- **Disadvantages**
  - Multiple processes are cumbersome
  - Still requires C/CUDA executable to be recompiled and possibly modified upon platform change
BeSSy – Java Native Interface

**Advantages**
- Probably don’t need separate methods per platform
- Code all in one project

**Disadvantages**
- Still not platform independent
- Requires native code to be compiled for each supported platform
- Additional build steps
  - Producing C/C++ headers
  - Compiling C/C++ code
  - Compiling CUDA code
BeSSy – Intermediate Library

**Advantages**
- Platform independent
  - As long as library is supported on platform
- No intermediate C/C++ code between Java and CUDA

**Disadvantages**
- Requires library to be distributed
- Additional build step for CUDA code
BeSSy – Additional Processing or Compiling Tool

- **Advantages**
  - No C/C++ or CUDA
  - Write Java-like code to run on GPU

- **Disadvantages**
  - Requires additional build step
  - Requires use of tool(s) with which few people are familiar
  - Relatively untested systems
  - Longevity in question due to competing tools, openACC
  - Depending on tool, might still need to recompile per platform
BeSSy – GPU Parallelization

- Chose Intermediate library as best option
  - Specifically, JCUDA

- JCUDA is open source
  - As long as Java and CUDA are supported on a platform, JCUDA will probably work

- JCUDA is mature and well maintained
  - Frequent updates, supports CUDA 5.0 (latest at time of writing)
  - Working project since at least 2009 – an eternity in GPU Computing
BeSSy – GPU Parallelization

- No recompiling of our own code
  - Java is platform independent, and CUDA can be compiled to PTX supporting wide range of CUDA GPUs

- No need to distribute BeSSy source code
  - The only set of code that is platform dependent is the library

- Existing Java and CUDA developers are already familiar with the necessary tools
BeSSy – Traveling Salesman Analog

- BeSSy is a sensitive tool not to be utilized outside of official use cases.

- Traveling Salesman Problem is an effective analog
  - Both are NP-Hard problems
  - Simulated Annealing and Genetic Algorithms are reasonable approaches for generating solutions.

- Traveling Salesman Problem is well known in CS world.

- TSP is not sensitive to Gov’t work
  - Source code can be shown and distributed.
Traveling Salesman Problem

- $n$ number of cities
- Salesman must visit each city exactly once
- Goal is to produce an optimal or near optimal sequence of cities to path
  - Optimal defined as shortest overall distance traveled
- For $n$ cities, there are $\sim n!$ possible solutions

![Diagram showing cities and path]
TSP – Simulated Annealing

1. Generate set of solutions
2. Modify solutions based on some Temperature (T)
   - Higher T produces greater change on average
3. Evaluate solutions
4. Replace bad solutions with copies of good solutions
5. Repeat starting at 2, until some number of iterations is reached
TSP – Genetic Algorithm

- 1 – Generate set of parent solutions
- 2 – Generate child solutions for each parent solution
- 3 – Correct children solutions
- 4 – Find best between parent and children and replace parent
- 5 – Repeat starting at 2, until some number of unchanged iterations reached
Traveling Salesman Problem

- Ideal for GPU parallelization
  - Highly parallelizable
  - Low memory requirements
  - Low device $\rightarrow$ host and host $\leftarrow$ device transfers

- Algorithm porting from CPU to GPU is nearly direct

- Substantial Performance benefits
  - $\sim$20x improvement for Genetic Algorithm
  - $\sim$80x improvement for Simulated Annealing
TSP – Initializing the GPU

- Nearly identical to C/C++ GPU initialization
  - Create CUdevice object
  - Create CUcontext object using Cudevice object
  - Load PTX on device
  - Create device function pointers
  - Allocate memory on device
  - Copy data to device
TSP – Initializing the GPU

```cpp
// Initialize the driver
cuInit(0);

// Create a context for the first device
device = new CUdevice();
cuDeviceGet(device, 0);
context = new CUcontext();
cuCtxCreate(context, 0, device);

// Load the ptx file
module = new CUmodule();
cuModuleLoad(module, ptxFileName);

// Initialize CUDA global function pointers
initParents = new CUfunction();
cuModuleGetFunction(initParents, module, "initParents");

data = new CUDeviceptr();
data = new CUDeviceptr();

// Allocate device memory for arrays
cuMemAlloc(data, cityCount * Sizeof.FLOAT);
cuMemAlloc(data, cityCount * Sizeof.FLOAT);

cuMemcpyHtoD(data, Pointer.to(x), cityCount * Sizeof.FLOAT);
cuMemcpyHtoD(data, Pointer.to(y), cityCount * Sizeof.FLOAT);
```

// Initialize device array pointers
// Allocate device memory for arrays
// Copy city positions to device
extern "C"
__global__ void reject(int *solutions, float *values, int cityCount, int solutionCount, int *mi, float *mv){
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if(index < solutionCount){
        float rejectionMultiplier = 1.15f;
        if(values[index] > rejectionMultiplier * mv[0]){  
            values[index] = mv[0];
            for(int a = 0; a < cityCount; a++){
                solutions[index * cityCount + a] = solutions[mi[0] * cityCount + a];
            }
        }
    }
}
TSP – Kernel Launch Setup

- Create Pointer to set of kernel arguments
- Copy any necessary data to device
- Call cuLaunchKernel
  - Set grid and block size
  - Set Shared memory utilization
  - Set kernel, extra parameters
- Copy any necessary data to host
TSP Kernel Launch Example

```c
//create set of parameters for call to reject on device
KernelParameters = Pointer.to(
    Pointer.to(d_solutions), // Solution array
    Pointer.to(d_values),     // Value array
    Pointer.to(new int[cityCount]), // Number of cities
    Pointer.to(new int[solutionCount]), // Number of solutions
    Pointer.to(d_mi),        // index of best solution
    Pointer.to(d_mv)         // value of best solution
);

// Call reject on device.
blockSizeX = 256;
gridSizeX = (int)Math.ceil((double)solutionCount / blockSizeX);
cuLaunchKernel(reject,
    gridSizeX, 1, 1, // Grid dimension
    blockSizeX, 1, 1, // Block dimension
    0, null, // Shared memory size and stream
    kernelParameters, null // Kernel- and extra parameters
)
```
Combining Java and CUDA – Notes

- Major programming concerns result from programming styles associated with two languages
  - Garbage collection in Java
  - Manual memory management in CUDA
  - Pointers are uncommon and avoided in Java
  - Pointers are critical in CUDA

- Typical CUDA parallelization issues
  - Setting up kernels rather than just calling host functions
  - Copying data to/from host
  - Designing code to run efficiently in parallel
Conclusion

- Number of options for programming CUDA & Java
  - Best option for BeSSy is intermediate library

- CUDA programming with Java is very similar to CUDA programming for C/C++
  - Most Functions behave identically or near identically

- Java programming model is somewhat different than C/C++ & CUDA
  - Some extra difficulty is introduced for programmers not familiar with either Java or CUDA
Source Code

Code soon to be available at:

http://code.google.com/p/travelingsalesmangpu