

## Compiling CUDA code along with other C++ code

7/10/2014

### 2 Comments

This post will focus mainly on how to get CUDA and ordinary C++ code to play nicely together. This seemed like a pretty daunting task when I tried it first, but with a little help from the others here at the lab and online forums, I got it to work.

As a starting point, I used the code that was shown in the previous post - The summation from 1 to n. This code was put into a class called GpuInterface in GpuSolver.cu and it also had a GpuSolver.h header file. The files are shown below:

GpuSolver.h

```
#ifndef EXAMPLE6_H
#define EXAMPLE6_H
class GpuInterface
{
public:
    int n[20];
    int y;
    int asize;

    GpuInterface();
    int calculateSum();
    void setY(int);
};
#endif
```

GpuSolver.cu

```
#include <iostream>
#include <cuda.h>
#include "GpuSolver.h"
```

\_\_global\_\_

```
void findSumToN(int *n, int limit)
{
    int tId = threadIdx.x;

    for (int i=0; i<=((int)log2((double)limit)); i++)
    {
        if (tId%(int)(pow(2.0,(double)(i+1)))) == 0){
            if (tId+(int)pow(2.0, (double)i) >= limit) break;
            n[tId] += n[tId+(int)pow(2.0, (double)i)];
        }
    }
}
```

```

        __syncthreads();
    }
}

GpuInterface::GpuInterface()
{
    y = 20;
    asize = y*sizeof(int);
    for (int i=0; i<y; i++)
        n[i] = i;
}

int GpuInterface::calculateSum()
{
    int *n_d;
    cudaMalloc( (void**)&n_d, asize );

    cudaMemcpy(n_d, n, asize, cudaMemcpyHostToDevice );

    dim3 dimBlock( y, 1 );
    dim3 dimGrid( 1, 1 );
    findSumToN<<<dimGrid, dimBlock>>>(n_d, y);
    cudaMemcpy(n, n_d, asize, cudaMemcpyDeviceToHost);
    cudaFree (n_d);
    return n[0];
}

void GpuInterface::setY(int newVal)
{
    y = newVal;
    asize = y*sizeof(int);
    for (int i=0; i<y; i++)
        n[i] = i;
}

```

And finally, I have a C++ file called main.cpp which just has a main function that creates a GpuSolver object and calls its functions, like so:

```

#include <iostream>
#include "GpuSolver.h"

int main()
{
    GpuInterface obj;
    obj.setY(16);
    std::cout << obj.calculateSum();
    return 0;
}

```

So the task now is to compile all of this into one executable. The key to understanding how we can do this is to understand how exactly the C++ compiler works. Professional computer science courses have a whole subject dedicated to Compilers, so I won't go into much detail. I'll just tell you that the compiler first converts the code (which is in a human readable format) into an intermediate code format. This is called assembly. The file created is called an object file. It contains essentially the same information as the source code file, but in a more machine-readable format. Separate object files are created for each source file in the project. Then, they are all linked together to form one single executable.

Typically, compilers automatically perform the assembly followed by the linking process. However, you can force it to stop after just the assembly, and then do the linking process later on. This is what we will have to do.

We run `g++` on `main.cpp` with the `-c` flag that instructs `g++` to stop compilation after the object files are generated. We also use the `-I.` flag to ask it to look for header files within the current folder. The `-o` flag asks the compiler to call the output as whatever string follows the flag (in this case `main.cpp.o`). The full command looks like:

```
g++ -c -I. main.cpp -o main.cpp.o
```

We do a similar compilation on `GpuSolver.cu` with the following command

```
nvcc -c -I. -I/usr/local/cuda/include GpuSolver.cu -o GpuSolver.cu.o
```

Apart from the fact that `g++` is replaced with `nvcc` (Nvidia CUDA Compiler) here, the only addition is the `"-I/usr/local/cuda/include"` flag. The path you see there contains some CUDA specific functionality that is required while compiling CUDA programs. So `nvcc` will need that library to compile the `.cu` file.

So now we have a bunch of files in our project directory:

- `main.cpp`
- `GpuSolver.cu`
- `GpuSolver.h`
- `main.cpp.o`
- `GpuSolver.cu.o`

We now need to link the two `.o` files into one executable. We do this with the following command:

```
g++ -o exec GpuSolver.cu.o main.cpp.o -L/usr/local/cuda/lib -lcudart
```

Firstly, notice how we are just using the normal `g++` call to perform the linking. `g++` is clever enough to know that only the linking step is required and skips the assembly automatically. `-o`, like before, ensures that the output is named 'exec'. The files that need to be linked are specified next, followed by a `-L` and `-l` flag. The `-L` flag asks the compiler to look at the directory specified in the flag for additional files that may need to be linked. `-l` specifies the exact file that needs to be linked. Again, this is specific to CUDA.

When all of this has been done, you get a neat little executable that will calculate the sum from 1 to `n`!

I packed all of these commands into a makefile, which I've put down here

```
CUDA_INSTALL_PATH := /usr/local/cuda
```

```
CXX := g++
```

```
CC := gcc
```

```
LINK := g++ -fPIC
```

```
NVCC := nvcc
```

```
# Includes
```

```
INCLUDES = -I. -I$(CUDA_INSTALL_PATH)/include
```

```
# Common flags
```

```
COMMONFLAGS += $(INCLUDES)
```

```
NVCCFLAGS += $(COMMONFLAGS)
```

```
CXXFLAGS += $(COMMONFLAGS)
```

```
CFLAGS += $(COMMONFLAGS)
```

```
LIB_CUDA := -L$(CUDA_INSTALL_PATH)/lib -lcudart
```

```
OBJS = GpuSolver.cu.o main.cpp.o
```

```
TARGET = exec
```

```
LINKLINE = $(LINK) -o $(TARGET) $(OBJS) $(LIB_CUDA)
```

```
.SUFFIXES: .c .cpp .cu .o
```

```
%.c.o: %.c
```

```
$(CC) $(CFLAGS) -c $< -o $@
```

```
%.cu.o: %.cu
```

```
$(NVCC) $(NVCCFLAGS) -c $< -o $@
```

```
%.cpp.o: %.cpp
```

```
$(CXX) $(CXXFLAGS) -c $< -o $@
```

```
$(TARGET): $(OBJS) Makefile
```

```
$(LINKLINE)
```

For those of you wondering what just happened, welcome to the world of Linux makefiles :) Makefiles are a way to script out the entire compilation and installation process for programs in linux. They have very weird syntax and there is no way I can explain all of it's details here. However, there are excellent tutorials on makefiles elsewhere on the internet, so I'd suggest doing some research.

The main point is that this makefile does almost exactly what I described above, with a bit of extra functionality for things like making sure the resulting executable can be further used in other programs, as opposed to needing to be run manually.

That's it for now! I'll leave the integration into MOOSE for next time.