Notes on using the nVidia 8800 GTX graphics card

Mike Giles and Su Xiaoke

Summary

These brief notes describe our experiences while developing a version of a LIBOR Monte Carlo code to execute on an nVidia 8800 GTX graphics card with 16 multi-processors, each of which has 8 cores.

It is assumed that the reader has already read at least the first three sections of the nVidia CUDA Programming Guide version 1.0.

1 LIBOR application

The LIBOR application involves calculating the evolution of 80 forward rates for 40 timesteps, after which a portfolio product is evaluated. The evolution is driven by an input vector of 40 random numbers, so a real Monte Carlo simulation calculates a very large number of paths using different random numbers and then averages the results. A second part of the application computes the so-called "Greeks" (the sensitivity of the payoff to changes in the initial forward rates) using an adjoint approach which requires the storing of all forward rates at all of the timesteps.

The key points from a computational point of view are that

- each path calculation is completely independent and can be handled by a separate thread
- each path requires roughly 100 working variables, 80 for the forward rates and 20 for other variables – in addition to this there are approximately 100 constants to be stored
- to compute the Greeks requires an additional 3200 variables per path

2 Experiences developing the code

The learning curve for programming within the CUDA development environment was roughly 2 weeks. This is less than for the Clearspeed Advance Board, but this was probably partly as a consequence of having tackled the Advance Board first. As with the Clearspeed product, the documentation is good and the example codes very helpful.

The basic programming model is clear and simple, at least for this trivially parallel application. Host code running on the main Intel or AMD processor initiates the execution of a "kernel" code on the graphics card. This execution is performed by multiple "blocks" each of which has multiple threads. Each block executes on a single multi-processor, and has a minimum of 32 threads, corresponding to 4 threads per core. However, for best performance one should have many more active threads; this is discussed further below.

The main thing for new users to understand is the memory hierarchy. There are several kinds of memory:

- host memory – the usual system memory
- device memory – GDDR3 memory on the graphics card
- constant and texture memory – off-chip memory with an on-chip cache
- shared memory – on-chip memory local to each multiprocessor
- registers – on chip-memory used as inputs/outputs of all arithmetic operations

For the purposes of trivially parallel applications we can ignore both the shared memory and the texture memory. The constant memory is ideal for storing the many constants that an application uses; the graphics multiprocessors have read-only access to this memory, while the host has write-only access. Any other data which has to be communicated between the host and the device is done through "global" variables which are allocated within the device memory; these can be read and written by both the host and the device. Finally, there are "local" variables which are specific to each thread. The compiler will keep some of these within the registers, but most will stored within the device memory.

The `nvcc` compiler has a useful flag `-cubin` which reports on the registers and other memory used by each thread in the kernel code. This can be input into an nVidia utility called the Occupancy Calculator to work out how many active threads can be supported, given the limited number of registers. There is then a programming choice. At one extreme, one can keep just one block per multiprocessor, and increase the number of threads to the maximum possible. Alternatively, at the other extreme, one can keep just 64 threads per block,

and use a large number of blocks; in this case the runtime system will decide how many blocks can run concurrently on each multiprocessor. In each case one ends up with a large number of active threads, and our experience is that there is less than 20% difference in overall performance.

The bottom line is in the timings reported in Table 1. The performance of the CUDA code on the 8800 GTX is exceptional. With the optimum number of blocks and threads, each one of the 128 graphics cores has greater throughput than a single Intel Xeon core.

		no Greeks	Greeks
original code	Visual Studio, one Xeon core	18.1	26.9
CUDA code	1500 blocks, 64 threads	0.048	0.20
CUDA code	750 blocks, 128 threads	0.048	0.19
CUDA code	500 blocks, 192 threads	0.045	0.19
CUDA code	375 blocks, 256 threads	0.047	0.19
CUDA code	300 blocks, 320 threads	0.046	0.18

Table 1: Program timings in seconds for 96000 paths

How is this possible? I think it is due to the efficient hardware/software architecture combined with the simple nature of this application which is ideally suited to that architecture. The large number of essentially identical path calculations makes it possible to use a large number of threads per core. This helps in two ways. The first is to achieve the maximum efficiency from the pipelined cores. With multi-stage pipelines efficiency is usually lost through pipeline stalls, waiting for inputs which are themselves the output of an earlier operation. With 24 active threads per core and each thread taking its turn in strict rotation, from the perspective of each individual thread it looks like a classic von Neumann architecture with the output of each operation available as an input for the very next operation.[1] The second benefit is in hiding memory latency. The bandwidth from the multiprocessors to the global GDDR3 memory on the graphics card is very high (80GB/s) but there is nevertheless a delay of 400 to 600 clock cycles. With a large number of threads, this latency can be largely hidden, though the results in the table suggest that memory bandwidth may still be a limiting feature; the Greeks computation requires roughly three times as much memory transfer as the computation without the Greeks.

[1]The vector operations on the Clearspeed card and hyperthreading in the Pentium 4 both try to address the same issue.

3 Notes on the code

The webpage contains the original C code and the new CUDA code.

The CUDA code is fairly similar to the original C code. The main change is in the "main" routine which has been split into two parts. The first part, which is again called "main", executes on the host to perform the following functions:

- copy all constants into the constant memory on the card
- allocate global memory for the payoffs computed on the card
- call the device kernel routines "Pathcalc_Portfolio_KernelGPU" and "Pathcalc_Portfolio_KernelGPU2" to do all of the work
- combine the output results and print the final values

The device kernel routines "Pathcalc_Portfolio_KernelGPU" and "Pathcalc_Portfolio_KernelGPU2" do the main work of calling "path_calc", "portfolio" and the other computational routines.

The makefile uses the makefile structure created by nVidia for their test programs. Specifically, they have a file common.mk which defines a generic makefile, and then the user's makefile just has to declare the names of the user's files. This also makes it possible to compile the code either with or without debugging (use dbg=1 for debugging and error checking) and with or without devide emulation (use emu=1 for device emulation).

4 Suggestions for other users

- Read through the most relevant examples supplied by nVidia, and use the toolkit and makefile structure they have developed for their test programs.
- Focus on the memory layout – for trivially parallel applications you just need to use local, constant and global memory on the card, in addition to the usual memory on the host.
- Define constants in the host code then copy them over to the card; allocate global memory for any data that needs to be communicated between the card and the host, and use local memory (the default) for all other data.
- Use the `nvcc` compiler option `-cubin` to tell you how many registers are used; input this into the Occupancy Calculator to get guidance on the maximum number of active threads.