M. Troyer
ETH Zürich, HIT G 31.8
CH-8093 Zürich

Set 9 - GPUs II

Issued: May 11, 2015
Hand in: May 18, 2015

Question 1: Diffusion on GPUs - Part II

In this exercise we will further improve the 2D diffusion code on GPUs.

a) Write a kernel for `GetMoment()` to further reduce the memory transfer needed between GPU and CPU by calculating (at least) paritial sums on the GPU.
Hint: An important part of the operation is a reduction.
While you can spend a full talk on how to optimize reductions[1], a simple GPU reduction will do here. It is called only every 100th update and will not dominate the execution time of our code. A simple reduction scheme within a block using the shared memory can be achieved by selecting only specific thread indices:

```
__shared__ float data[8];

...
if(threadIdx.x < 4)
    data[threadIdx.x] += data[threadIdx.x + 4];
__syncthreads();
if(threadIdx.x < 2)
    data[threadIdx.x] += data[threadIdx.x + 2];
__syncthreads();
if(threadIdx.x < 1)
    data[threadIdx.x] += data[threadIdx.x + 1];
```

A solution code is found in `solution/diffusion2d_cuda_shared.cu`, the achieved speed-up is listed in table 1.

b) Think about good choices for blocksPerGrid and threadsPerBlock of your kernels and explain your choice.
Since the NVIDIA's Fermi (GF) and Kepler (GK) GPUs schedule threads in groups of 32 threads (called a "warp"), `threadsPerBlock` should be a multiple of 32. Apart from this general rule, there are a few ways to obtain a good pair of `threadsPerBlock` and `blocksPerGrid`, such as using the NVIDIA Visual Profiler or using the CUDA occupancy calculator. We decided to use the latter in this solution.
Compiling our solution code with following additional options, reveals some properties of the compiled kernels, such as the number of required registers and shared memory usage:

[1]M. Harris, Optimizing Parallel Reduction in CUDA, 2007,
`http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf`

Version	time	speed-up
OpenMP	263.0s	1.0x
CUDA	161.8s	1.6x
CUDA with shared mem	119.3s	2.2x
getMoment() on GPU		
CUDA	130.6s	2.0x
CUDA with shared mem	89.2s	2.9x

Table 1: Execution time of the 2D diffusion solution for system size $2^{11} \times 2^{11}$.

```
nvcc -O3 -arch=sm_20 --ptxas-options=-v -o diffusion_cuda_shared diffusion2d_cuda_shared.cu
ptxas info    : Compiling entry function '_Z16diffusion_kernelPfPKffi' for 'sm_20'
ptxas info    : Function properties for _Z16diffusion_kernelPfPKffi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 12 registers, 1296+0 bytes smem, 56 bytes cmem[0]

ptxas info    : Compiling entry function '_Z17get_moment_kernelPfPKfffi' for 'sm_20'
ptxas info    : Function properties for _Z17get_moment_kernelPfPKfffi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 18 registers, 512+0 bytes smem, 60 bytes cmem[0]
```

We use this information in the CUDA Occupancy calculator `CUDA_Occupancy_Calculator.xls` of the CUDA toolkit. Given information about the Hardware ("compute capability") and the kernel requirements the spreadsheet calculates how many warps may be scheduled on one multiprocessor when launching the kernel. The occupancy is the number of active warps divided the number of warps supported by one of the multiprocessors of the GPU. If less warps are active, the multiprocessor will have more problems to hide latencies (e.g. memory access latencies), because it has less warps to schedule for execution while a warp is waiting for an operation to finish. It will be more likely that the multiprocessor is idle, because all warps/threads are waiting for some operations to finish at the same time. Hence, we try to optimize for highest occupancy to make best use of the multiprocessor.

We start by analyzing the `get_moment_kernel()` and make an initial guess of `threadsPerBlock=128`. Using the `get_compute_capability.cu` program we obtain the compute cababilty of the Fermi GPUs installed on Brutus, which is 2.0. Compiling our kernel for a device with compute capability 2.0 (`nvcc -arch=sm_20`), we require 18 registers per thread and 512 Bytes of shared memory per block, as shown in the compiler output above. Entering this data in the occupancy calculator, we obtain an occupancy 67%, which corresponds to only 32 warps of the 48 supported warps being active. The graphs on the right reveal the limiting factor is neither the register requirements of the threads nor the shared memory requirements of the thread blocks, but the chosen `threadsPerBlock`. Increasing the `threadsPerBlock=256`, which also increases the shared memory to 1024 Bytes, yields a perfect occupancy of 100%. (Note that we may also need to adapt the number of steps in our parallel reduction within the kernel if we change the number of threads per block.)

The same analysis of the `diffusion_kernel()` with our initial guess of $16 \times 16 = 256$ threads per block, using 12 registers per thread and 1296 Bytes of shared memory per block, yields already a perfect occupancy.

Summary

Summarize your answers, results and plots into a short PDF document. Furthermore, elucidate the main structure of the code and report possible code details that are relevant in terms of accuracy or performance. Send the PDF document and source code to your assigned teaching assistant.