CSE 746 - Parallel and High Performance Computing
Lecture 2 - Overview of CUDA

Pawel Pomorski, *HPC Software Analyst*

SHARCNET, University of Waterloo

ppomorsk@sharcnet.ca

http://ppomorsk.sharcnet.ca/

CSE 746 - Advanced Parallel and High Performance Computing

- Instructor: Dr. Pawel Pomorski (ppomorsk@sharcnet.ca)
- Course website: http://ppomorsk.sharcnet.ca/CSE_746.html
- Office: E6-2020 at University of Waterloo
- Office hours: please arrange meeting before/after lecture via email
- Lectures: Wednesday 2:30-5:30 in UH 101
- 12 lectures total, no lecture during winter break
- no course materials required
- Students need SHARCNET account, and should bring laptop to class for hands-on activities

CSE 746 - Advanced Parallel and High Performance Computing

- Evaluation:
  Final project: 40%
  Two assignments: 20% each
  Two in-class quizzes: 10% each
- Final project can be chosen by student in his/her area of interest or research. If that's not a good option, a topic can be selected in consultation with instructor.

Overview

- Introduction to GPU programming
- Introduction to CUDA
- CUDA example programs
- CUDA libraries
- OpenACC
- CUDA extensions to the C programming language
- Beyond the basics - initial discussion on optimizing CUDA

Linear algebra on the GPU

- Linear algebra on the CPU: BLAS, LAPACK
- GPU analogues: CUBLAS, CULA
- CUSPARSE library for sparse matrices
- Use of highly optimised libraries is always better than writing your own code, especially since GPU codes cannot yet be efficiently optimized by compilers to achieve acceptable performance
- Writing efficient GPU code requires special care and understanding the peculiarities of underlying hardware

CUBLAS

- Implementation of BLAS (Basic Linear Algebra Subprograms) on top of CUDA
- Included with CUDA (hence free)
- Workflow:
  1. allocate vectors and matrices in GPU memory
  2. fill them with data
  3. call sequence of CUBLAS functions
  4. transfer results from GPU memory to host
- Helper functions to transfer data to/from GPU provided

Error checks

- in following example most error checks were removed for clarity
- each CUBLAS function returns a status object containing information about possible errors
- It’s very important these objects to catch errors, via calls like this:

```
if (status != CUBLAS_STATUS_SUCCESS) {
print diagnostic information and exit}
```

Initialize program

```
#include <cuda.h> /* CUDA runtime API */
#include <cstdio>
#include <cublas_v2.h>


int main(int argc, char *argv[])
{
   float *x_host, *y_host; /* arrays for computation on host*/
   float *x_dev, *y_dev;   /* arrays for computation on device */

   int n = 32*1024;
   float alpha = 0.5f;
   int nerror;

   size_t memsize;
   int i;

   /*  could add device detection here */

   memsize = n * sizeof(float);
```

Allocate memory on host and device

```
     /* allocate arrays on host */

   x_host = (float *)malloc(memsize);
   y_host = (float *)malloc(memsize);

/* allocate arrays on device */

   cudaMalloc((void **) &x_dev, memsize);
   cudaMalloc((void **) &y_dev, memsize);

   /* initialize arrays on host */

   for ( i = 0; i < n; i++)
   {
      x_host[i] = rand() / (float)RAND_MAX;
      y_host[i] = rand() / (float)RAND_MAX;
   }

   /* copy arrays to device memory (synchronous) */

   cudaMemcpy(x_dev, x_host, memsize, cudaMemcpyHostToDevice);
   cudaMemcpy(y_dev, y_host, memsize, cudaMemcpyHostToDevice);
```

Call CUBLAS function

```
cublasHandle_t handle;
cublasStatus_t status;

status  = cublasCreate(&handle);

int stride = 1;
status = cublasSaxpy(handle,n,&alpha,x_dev,stride,y_dev,stride);

/* check if cublasSaxpy launched succesfully */

if (status != CUBLAS_STATUS_SUCCESS)
 {
   printf ("Error in launching CUBLAS routine \n");
   exit (20);
 }

status = cublasDestroy(handle);
```

Retrieve computed data and finish

```
   /* retrieve results from device (synchronous) */
   cudaMemcpy(y_host, y_dev, memsize, cudaMemcpyDeviceToHost);

  /* ensure synchronization (cudaMemcpy is synchronous in most cases, but not all) */

   cudaDeviceSynchronize();

   /* use data in y_host*/

   /* free memory */
   cudaFree(x_dev);
   cudaFree(y_dev);
   free(x_host);
   free(y_host);

   return 0;
}
```

OpenACC

- New standard for parallel computing developed by compiler makers. See: http://www.openacc-standard.org/
- Specified in late 2011, released in 2012
- SHARCNET has the PGI compiler on monk which supports it
- OpenACC works somewhat like OpenMP
- Goal is to provide simple directives to the compiler which enable it to accelerate the application on the GPU
- The tool is aimed at developers aiming to quickly speed up their code without extensive recoding in CUDA
- As tool is very new and this course focuses on CUDA, only a brief demo of OpenACC follows

SAXPY with OpenACC

```
...
#include <openacc.h>

void saxpy_openacc(float *restrict vecY, float *vecX, float alpha, int n)
{
     int i;
#pragma acc kernels
     for (i = 0; i < n; i++)
           vecY[i] = alpha * vecX[i] + vecY[i];
}

...
   /* execute openacc accelerated function on GPU */
   saxpy_openacc(y_shadow, x_host, alpha, n);
...
```

- OpenACC automatically builds a kernel function that will run on GPU
- Memory transfers between device and host handled by OpenACC and need not be explicit

Compiling SAXPY with OpenACC

```
[ppomorsk@mon54:~] module unload intel
[ppomorsk@mon54:~] module load pgi
[ppomorsk@mon54:~/CUDA_day1/saxpy] pgcc -acc -Minfo=accel -fast saxpy_openacc.c
saxpy_openacc:
     25, Generating copyin(vecX[0:n])
         Generating copy(vecY[0:n])
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
     26, Loop is parallelizable
         Accelerator kernel generated
         26, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
             CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes
             CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes
[ppomorsk@mon54:~/CUDA_day1/saxpy] export ACC_NOTIFY=1
[ppomorsk@mon54:~/CUDA_day1/saxpy] export PGI_ACC_TIME=1
[ppomorsk@mon54:~/CUDA_day1/saxpy] ./a.out
launch kernel  file=/home/ppomorsk/CUDA_day1/saxpy/saxpy_openacc.c function=saxpy_openacc
line=26 device=0 grid=128 block=256 queue=0

Accelerator Kernel Timing data
/home/ppomorsk/CUDA_day1/saxpy/saxpy_openacc.c
  saxpy_openacc
    25: region entered 1 time
        time(us): total=4241617 init=4240714 region=903
                  kernels=22 data=461
        w/o init: total=903 max=903 min=903 avg=903
        26: kernel launched 1 times
            grid: [128]  block: [256]
            time(us): total=22 max=22 min=22 avg=22
```

Is OpenACC always this easy?

- No, the loop we accelerated was particularly easy for the compiler to interpret. It was very simple, and each iteration was completely independent of the others
- If the accelerate directive is placed before a more complicated loop, the compiler refuse to accelerate the region, complaining of errors
- More specific compiler directives must hence be provided for more complicated functions
- Memory transfers must be handled explicitly if we don't want to transfer memory to/from device every time kernel is called
- For complex problems OpenACC grows as complex as CUDA, but it might get better in the future

Helpful tools

- CUDA 5+ includes Nsight, an Integrated Development Environment (IDE) for Linux/Mac based on Eclipse. IDE incorporates CUDA-aware editor, profiler and debugger in one close-integrated package. Try it out!
- There is a Visual Studio edition of Nsight for Windows
- On SHARCNET the DDT visual debugger has powerful GPU debugging capability

Further reading

- CUDA Programming Guide
- CUDA sample projects
  - many contain extended documentation
  - similarity to the matrix transpose, the reduction project is an excellent step-by-step walkthrough of how to optimize code for the hardware (read/write coalescing, shared memory, bank conflicts, etc.)
- Lots of documentation/presentations/tutorials online
- NVIDIA website - lots of materials

Introduction to GPU Programming: CUDA

CUDA EXTENSION TO THE C PROGRAMMING LANGUAGE

Storage class qualifiers

Functions

`__global__`	Device kernels callable from host (and from device on CC 3.x or higher
`__device__`	Device functions (only callable from device)
`__host__`	Host functions (only callable from host) - **default if not specified** **- can be combined with __device__**

Data

`__shared__`	Memory shared by a block of threads executing on a multiprocessor.
`__constant__`	Special memory for constants (cached)

CUDA data types

- C primatives:
  - `char`, `int`, `float`, `double`, …
- Short vectors:
  - `int2`, `int3`, `int4`, `uchar2`, `uchar4`, `float2`, `float3`, `float4`, …
  - no built-in vector math (although a utility header, `cutil_math.h`, defines some common operations)
- Special type used to represent dimensions
  - `dim3`
- Support for user-defined structures, e.g.:

```
struct particle
{
    float3 position, velocity, acceleration;
    float mass;
};
```

Library functions available to kernels

- Math library functions:
  - `sin`, `cos`, `tan`, `sqrt`, `pow`, `log`, ...
  - `sinf`, `cosf`, `tanf`, `sqrtf`, `powf`, `logf`, ...

- ISA intrinsics
  - `__sinf`, `__cosf`, `__tanf`, `__powf`, `__logf`, ...
  - `__mul24`, `__umul24`, ...

- Intrinsic versions of math functions are faster but less precise

Built-in kernel variables

`dim3 gradDim`

– number of blocks in grid

`dim3 blockDim`

– number of threads per block

`dim3 blockIdx`

– number of current block within grid

`dim3 threadIdx`

– index of current thread within block

printf inside kernels is supported (CC 2.x or higher)

```
[ppomorsk@mon54:~/] nvcc test_print.cu
saxpy_cuda_timed_print.cu(58): error: calling a __host__ function("printf") from a
__global__ function("saxpy_gpu") is not allowed

1 error detected in the compilation of "/tmp/
tmpxft_000004f7_00000000-6_saxpy_cuda_timed_print.cpp1.ii".
[ppomorsk@mon54:~/] nvcc -arch=sm_20  test_print.cu
```

CUDA kernels: limitations

- No recursion in __global__ functions
- Can have recursion in __device__ functions on cards with CC 2.x or higher
- No variable argument lists
- No dynamic memory allocation
- Function pointers to __device__ functions in device code only supported on CC 2.x or higher
- No static variables inside kernels (except __shared__)

Can have separate code for different CC

```
__host__ __device__ func()
{
#if __CUDA_ARCH__ >= 300
    // Device code path for compute capability 3.x
#elif __CUDA_ARCH__ >= 200
    // Device code path for compute capability 2.x
#elif __CUDA_ARCH__ >= 100
   // Device code path for compute capability 1.x
#elif !defined(__CUDA_ARCH__)
   // Host code path
#endif
}
```

Launching kernels

- Launchable kernels must be declared as '`__global__ void`'

```
__global__ void myKernel(paramList);
```

- Kernel calls must specify device execution environment
  - grid definition – number of blocks in grid
  - block definition – number of threads per block
  - optionally, may specify amount of shared memory per block (more on that later)

- Kernel launch syntax:

```
myKernel<<<GridDef, BlockDef>>>(paramList);
```

Thread addressing

- Kernel launch syntax:

```
myKernel<<<GridDef, BlockDef>>>(paramlist);
```

- `GridDef` and `BlockDef` can be specified as `dim3` objects
  - grids can be 1D, 2D or 3D
  - blocks can be 1D, 2D or 3D

- This makes it easy to set up different memory addressing for multi-dimensional data.

Thread addressing (cont.)

- 1D addressing example: 100 blocks with 256 threads per block:

```
dim3 gridDef1(100,1,1);
dim3 blockDef1(256,1,1);
kernel1<<<gridDef1, blockDef1>>>(paramList);
```

- 2D addressing example: 10x10 blocks with 16x16 threads per block:

```
dim3 gridDef2(10,10,1);
dim3 blockDef2(16,16,1);
kernel2<<<gridDef2, blockDef2>>>(paramList);
```

- Both examples launch the same number of threads, but block and thread indexing is different
  - `kernel1` uses `blockIdx.x`, `blockDim.x` and `threadIdx.x`
  - `kernel2` uses `blockIdx.[xy]`, `blockDim.[xy]`, `threadIdx.[xy]`

Thread addressing (cont.)

- One-dimensional addressing example:

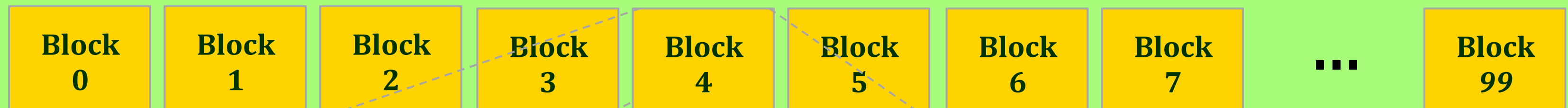
```
__global__ void kernel1(float *idata, float *odata)
{
    int i;

    i = blockIdx.x * blockDim.x + threadIdx.x;
    odata[i] = func(idata[i]);
}
```

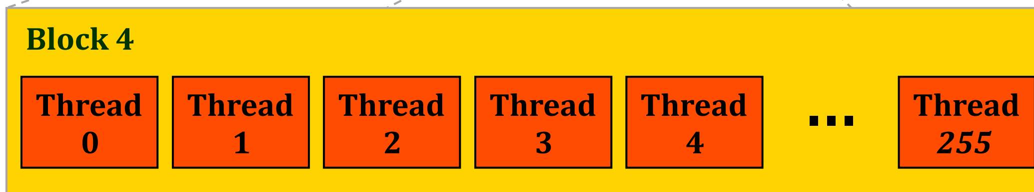
- Two-dimensional addressing example:

```
__global__ void kernel2(float *idata, float *odata, int pitch)
{
    int x, y, i;

    x = blockIdx.x * blockDim.x + threadIdx.x;
    y = blockIdx.y * blockDim.y + threadIdx.y;
    i = y * pitch + x;
    odata[i] = func(idata[i]);
}
```

Thread addressing (cont.)

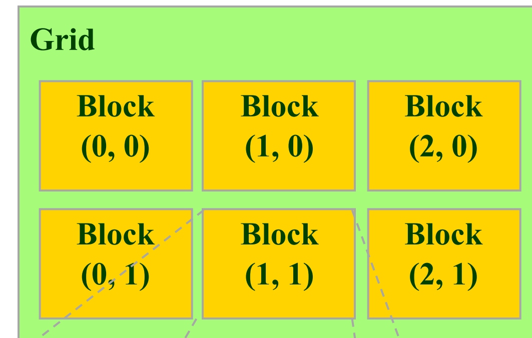
```
__global__ void kernel1(float *idata, float *odata)
{
    int i;

    i = blockIdx.x * blockDim.x + threadIdx.x;
    odata[i] = func(idata[i]);
}
...
dim3 gridDef1(100,1,1);
dim3 blockDef1(256,1,1);
kernel1<<<gridDef1, blockDef1>>>(paramList);
```





Thread addressing (cont.)

```
__global__ void kernel2(float *idata, float *odata, int pitch)
{
    int x, y, i;

    x = blockIdx.x * blockDim.x + threadIdx.x;
    y = blockIdx.y * blockDim.y + threadIdx.y;
    i = y * pitch + x;
    odata[i] = func(idata[i]);
}
...
dim3 gridDef2(10,10,1);
dim3 blockDef2(16,16,1);
kernel2<<<gridDef2, blockDef2>>>(paramList);
```



Block (1, 1)

Thread (0, 0)	Thread (1, 0)	Thread (2, 0)	Thread (3, 0)	Thread (4, 0)
Thread (0, 1)	Thread (1, 1)	Thread (2, 1)	Thread (3, 1)	Thread (4, 1)
Thread (0, 2)	Thread (1, 2)	Thread (2, 2)	Thread (3, 2)	Thread (4, 2)