GPU-programming

A GPU (Graphics Processing Unit) is used to generate images (rendering) to be shown on a computer display. Rendering for animated 3D-graphics can be very computationally intense so speed is essential. The computation is performed by a shader (a program), essentially on pixel level. The shader can be run in parallel in a multi-threaded SIMD-fashion, and a GPU is basically a parallel SIMD computer.
Since a GPU has excellent floating point performance, GPU:s can be used for scientific computing as well (at least for suitable applications). It used to be quite complicated to use GPU:s this way, but these last few years it has become less complicated.

The focus of this lecture is CUDA, (Compute Unified Device Architecture), Nvidia's GPU-architecture. Nvidia provides an SDK (software development kit) which makes it possible to use the GPU from C.

There are third party wrappers for other languages as well such as Fortran and Matlab. Matlab requires a GPU that supports double precision (CUDA compute capability ≥ 1.3). The GPU:s on the math computers only support single precision
(have compute capability 1.1).
An alternative to the CUDA SDK is OpenCL (see Wikipedia).

This lecture will skip most of the details, but if you need to know more, read a CUDA-book (> 300 pages).
See `http://developer.nvidia.com/cuda-books` .

A few are available as e-books form the Chalmers library, e.g.
R. Farber, CUDA Application Design and Development, Morgan Kaufmann, 2011 (Chalmers e-book from `sciencedirect.com`).

On the next page comes an *incomplete* description of the GPU:s in the student lab.

The lab computers have Nividia GeForce 9500 GT, GPU:s (see `/proc/driver/nvidia/gpus/0/information` or type `gpuDevice` in Matlab, for example), a mid-range GPU.

According to the `CUDA_C_Programming_Guide.pdf` (part of the SDK distribution), the GPU has four SM:s (streaming multiprocessors). Each SM has two SFU:s (special function units) and eight SP:s (shader or streaming processors), each with one FPU. Only single precision is supported. The $4 \cdot 8 = 32$ SP:s run at 1.4 GHz (`http://en.wikipedia.org/wiki/GeForce_9_Series`). The GPU has 512 Mbyte memory.

Each FPU can finish two MAD:s per cycle and each SFU can finish four function evaluations (or multiplications) per cycle. In `http://en.wikipedia.org/wiki/Nvidia_gpus` the theoretical floating point performance for GeForce 9500 GT is stated as 134.4 Gflop/s (not our usual definition of flop).
Counting one MAD or one SF as a flop we get:

$$4 \cdot (8 \cdot 2 \text{ MAD} + 2 \cdot 4 \text{ SF}) \cdot 1.4 \cdot 10^9/\text{s} = 134.4 \cdot 10^9 \text{ flops/s}$$

The dual core CPU is capable of $2 \cdot 4 \cdot 3.2 \cdot 10^9 = 25.6 \cdot 10^9$ (two threads, four + and four * (vector-single), equivalent to four MAD).

CPU:s are complicated: high clock frequency, large caches, branch prediction, multiple instruction streams, out of order execution, speculative execution, MIMD-applications, few threads.

GPU:s are "simple": lower clock frequency, no caches (but starting to appear on new high-end models), in-order execution, SIMD-applications. A GPU requires *many* threads to give good performance.

Thousands of threads making fairly simple operations at low speed can be much faster than a few threads performing complicated operations at high speed.

There are much more powerful (and expensive > 25 kkr) GPU:s. **`http://en.wikipedia.org/wiki/CUDA`** lists specifications for GPU:s based on their "compute capability-values".
The math compute server has a Tesla C1060-GPU, rather good, with 240 SP:s running at 1.296 GHz. Peak performance is 622 Gflop/s (single), 933 counting SF:s, and 77.8 Gflop/s (double) **`http://en.wikipedia.org/wiki/Nvidia_Tesla`**.

An easy way to use the GPU is to call a ready-made function (BLAS or FFT, for example) supplied with the SDK.

The following program uses **`cublasSgemm`** to compute **`C := alpha * A * B + beta * C`** where **`A`**, **`B`** and **`C`** are single precision 3×3-matrices. The code consists of the following main parts:

- Allocate and initialize the matrices in main memory.
- **`cudaMalloc`**: allocate space for the matrices on the GPU.
- **`cublasCreate`**: initialize CUBLAS.
- **`cublasSetMatrix`**: copy the matrices from main memory to the GPU.
- **`cublasSgemm`**: perform the matrix operation on the GPU.
- **`cublasGetMatrix`**: transfer the result from the GPU to main memory.
- **`cudaFree`**: deallocate matrices on the GPU.
- **`cublasDestroy`**: release CUBLAS resources.

A bottleneck is usually the transfer of data to and from the GPU (using the PCI-bus).

Here comes the program, it is rather long even though I have tried to shorten it, using a **`clean_up`**-routine for error messages and deallocation of memory.


```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda_runtime.h>
#include "cublas_v2.h"

// My own routine to shorten the code
void clean_up(float *, float *, float *,
              cublasHandle_t, int);

int main(void)
{
  cudaError_t cudaStat;
  cublasStatus_t stat;
  cublasHandle_t handle;

  // size and byt (number of bytes) to shorten the code
  int j, n = 3, size = sizeof(float),
      byt = n * n * size;

  // use malloc for large problems
  float A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  float B[] = {9, 8, 7, 6, 5, 4, 3, 2, 1};
  float C[] = {1, 1, 1, 1, 1, 1, 1, 1, 1};
  float *GPU_A, *GPU_B, *GPU_C;
  float alpha = 1.0, beta = -1.0;

  // To make clean_up work properly.
  GPU_A = GPU_B = GPU_C = 0;
  handle = (cublasHandle_t) 0; // eight byte zero

  // Allocate space on the GPU
  if (cudaMalloc((void **) &GPU_A, byt) != cudaSuccess)
    clean_up(GPU_A, GPU_B, GPU_C, handle, 0);
```

```
  if (cudaMalloc((void **) &GPU_B, byt) != cudaSuccess)
    clean_up(GPU_A, GPU_B, GPU_C, handle, 1);

  if (cudaMalloc((void **) &GPU_C, byt) != cudaSuccess)
    clean_up(GPU_A, GPU_B, GPU_C, handle, 2);

  // Initialize the CUBLAS library context
  if (cublasCreate(&handle) != CUBLAS_STATUS_SUCCESS)
    clean_up(GPU_A, GPU_B, GPU_C, handle, 3);

  // Copy the matrices to the GPU
  if (cublasSetMatrix(n, n, size, A, n, GPU_A, n) !=
      CUBLAS_STATUS_SUCCESS)
    clean_up(GPU_A, GPU_B, GPU_C, handle, 4);

  if (cublasSetMatrix(n, n, size, B, n, GPU_B, n) !=
      CUBLAS_STATUS_SUCCESS)
    clean_up(GPU_A, GPU_B, GPU_C, handle, 5);

  if (cublasSetMatrix(n, n, size, C, n, GPU_C, n) !=
      CUBLAS_STATUS_SUCCESS)
    clean_up(GPU_A, GPU_B, GPU_C, handle, 6);

  // Compute C := alpha * A * B + beta * C
  stat = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
                     n, n, n, &alpha, GPU_A, n, GPU_B,
                     n, &beta, GPU_C, n);

  if (stat != CUBLAS_STATUS_SUCCESS)
    clean_up(GPU_A, GPU_B, GPU_C, handle, 7);

  // Copy GPU_C to C
  if (cublasGetMatrix(n, n, size, GPU_C, n, C, n) !=
      CUBLAS_STATUS_SUCCESS)
    clean_up(GPU_A, GPU_B, GPU_C, handle, 8);
```



```
  // Free allocated resources on the GPU, no error
  clean_up(GPU_A, GPU_B, GPU_C, handle, 9);

//   In Matlab:
//   C = reshape(1:9, 3, 3) * reshape(9:-1:1, 3, 3) - 1
//   C =
//        89    53    17
//       113    68    23
//       137    83    29

  for (j = 0; j < n * n; j++)
    printf("%4.0f,", C[j]);
  printf("\n");

  return EXIT_SUCCESS;
}

void clean_up(float *GPU_A, float *GPU_B, float *GPU_C,
              cublasHandle_t handle, int err_no)
{
  char errors[9][37] = {
    "device memory allocation of A failed",
    "device memory allocation of B failed",
    "device memory allocation of C failed",
    "CUBLAS initialization failed",
    "data download of A failed",
    "data download of B failed",
    "data download of C failed",
    "cublasSgemm failed",
    "data upload of C failed"};

// It is safe to cudaFree(arg) if arg = 0.
  cudaFree(GPU_A); cudaFree(GPU_B); cudaFree(GPU_C);
  if(handle != 0)  // or it may crash
    cublasDestroy(handle);
```

```
  if ( err_no < 9 ) {
    printf("%s\n", errors[err_no]);
    exit(EXIT_FAILURE);
  }
}
```

We compile using the Nvidia C-compiler, **nvcc**, and link with **cublas**:

```
% nvcc simple_sgemm_ex_short.c -lcublas
% ./a.out
  89, 113, 137,  53,  68,  83,  17,  23,  29,
```

If you want to test CUDA on the lab-computers look at the non-compulsory CUDA-lab. You need to fetch and install the SDK yourself (easy, look at the lab) since I am prohibited by the Nvidia-license to install the SDK in the **unsup64**-area.

A drawback with using CUDA this way (CUBLAS) is that the transfer of data to and from the GPU is a bottleneck since the PCI-bus is rather slow. In the **sgemm**-example we need to transfer *four* matrices. So even though the floating point arithmetic may be much faster, the total execution time may no be reduced so much (or at all).

It is possible to write CUDA-code directly (instead of using existing libraries). This is rather more complicated as one has to keep track of threads and use the up to six different types of memory in an efficient way (register, local, shared, global, constant and texture).

For maximum performance you would like to create the data on the GPU, perform *many* flops and then transfer the result back (preferably just a few numbers) to the CPU. Using the graphics library OpenGL, **http://en.wikipedia.org/wiki/Opengl**, it is possible to display a graphical result on the monitor directly, although I have not tested this yet. See chapter 3.2.11 Graphics Interoperability in the NVIDIA CUDA C Programming Guide, for details.

Here comes a low-level example, but first some technical details. Threads are divided into *thread blocks*. Each SM can execute one or more thread blocks. Threads in a thread blocks can share data through shared memory in the SM. The GPU has a global scheduler which allocates one or more thread blocks to each SM, taking hardware limitations into account. The GF 9500 supports a maximum of 768 threads per SM (newer CUDA architectures allow for more). These 768 threads can be divided into three blocks of 256 threads each or six blocks of 128 threads.

It may be convenient to map the thread numbers, in a thread block, to one, two or three dimensions (much like using Cartesian grids in MPI). For this purpose the SDK supplies a 3-component vector, **threadIdx** (a C++-object, I think) with the three member variables, x, y and z (for the dimensions). The vector can be used in the GPU-code and different threads will get different values of the member variables. We define dimensions of the blocks when calling the GPU-code. Thread blocks are similarly organized in 1D, 2D or 3D-*grids* (though not 3D for the GeForce 9500). This is part of a code (it contains some new C++-statements):

```
...
  __global__ void gpu_func(parameter list)
  {
    compute ....
  }

int main()
{
  ...
  dim3 threadsPerBlock(4, 8, 2);  // C++-constructor
  dim3 blocksPerGrid(2, 4);    // only 2D for GF 9500

  gpu_func <<< blocksPerGrid, threadsPerBlock >>>
    (parameter list);    // start GPU-code
  ...
}
```

Inside the GPU-code we can access **blocksPerGrid** and **threadsPerBlock** using **blockDim** and **blockIdx**.
In this example **blockDim.x**, **blockDim.y** and **blockDim.z** equals 4, 8 and 2. **(blockIdx.x, blockIdx.y)** will take on the values (0, 0), (1, 0), (0, 1), (1, 1), (0, 2), (1, 2), (0, 3) and (1, 3).
For each block the **threadIdx**-vector will be (written in two columns to save space):

```
(0, 0, 0)     (0, 0, 1)
(1, 0, 0)     (1, 0, 1)
(2, 0, 0)     (2, 0, 1)
(3, 0, 0)     (3, 0, 1)

(0, 1, 0)     (0, 1, 1)
(1, 1, 0)     (1, 1, 1)
(2, 1, 0)     (2, 1, 1)
(3, 1, 0)     (3, 1, 1)

etc

(0, 7, 0)     (0, 7, 1)
(1, 7, 0)     (1, 7, 1)
(2, 7, 0)     (2, 7, 1)
(3, 7, 0)     (3, 7, 1)
```

The grid can we viewed as a 2×4-array where each element (a block) is a $4 \times 8 \times 2$-array. An element in a block is a thread.

Assume now we have a 1D-grid with two 1D-blocks consisting of four threads each. We can make the call:

```
// scalars (instead of dim3) will do in this case
gpu_func <<< 2, 4 >>> (parameter list);
```

and inside the GPU-code we use:

```
int i_am = blockIdx.x * blockDim.x + threadIdx.x;
```

to get a consecutive numbering of the threads.


```
i_am = blockIdx.x * blockDim.x + threadIdx.x;

blockDim = (4,  1,  1) constant in the GPU-code

blockIdx = (0,  0,  0)
threadId                        i_am
(0,  0,  0)                      0
(1,  0,  0)                      1
(2,  0,  0)                      2
(3,  0,  0)                      3

blockIdx = (1,  0,  0)
threadId
(0,  0,  0)                      4
(1,  0,  0)                      5
(2,  0,  0)                      6
(3,  0,  0)                      7
```

Now for the code. We approximate π by throwing darts, hitting in (x, y), $0 \le x, y \le 1$, x, y uniformly distributed on $(0, 1)$. The number of (x, y) with $x^2 + y^2 \le 1$ compared the number of darts thrown is proportional to $\pi/4$.

Standard C-routines for random numbers are not thread-safe (they save status information). There are thread-safe routines but they may not be called from the GPU-code, you get
error: calling a host function from a __device__/__global__ function("darts") is not allowed. There is a CUDA-library for random numbers but I copied some simple code from www instead.

Calling **printf** gives the same error (unless the GPU has compute capability 2.x). Nvidia supplies **cuPrintf**. Register as a CUDA-developer or google, for access. You need **cuPrintf.cu** and **cuPrintf.cuh**.

To call your own function from the GPU-code, use **__device__**, e.g. **__device__ float my_func(float x, float y)**

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

double csecond();

#define THROWS_PER_THREAD 2000000

__global__ void darts(uint *hits)
{
  const float r_max = 2.328306435454494e-10;
  uint m_z, m_w, u, n; // r_max should be double ...
  float x, y;          // automatic variables
  int i_am, k;         // in registers

  // thread ID
  i_am = blockDim.x * blockIdx.x + threadIdx.x;

  n = 0;
  m_z = 1 + i_am;   m_w = 1;

  for(k = 0; k < THROWS_PER_THREAD; k++) {
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    u = (m_z << 16) + m_w; // <<, >> left, right shift
    x = (u + 1.0f) * r_max;  // random number

    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    u = (m_z << 16) + m_w;
    y = (u + 1.0f) * r_max;  // random number

    if (x*x + y*y <= 1.0f)  // inside circle?
      n++;                  // 1.0f single prec one
  }                        // store in texture memory
  hits[i_am] = n;          // off chip, on board, slow
}
```



```
int main()
{
  int threadsPerBlock = 512 / 4, k;
  int blocksPerGrid = 4;
  size_t size = threadsPerBlock * blocksPerGrid *
                sizeof(uint);

  double t = csecond();

  // allocate space
  uint *cpu_hits = (uint *) malloc(size);
  uint *gpu_hits;
  cudaMalloc(&gpu_hits, size); // in texture memory

  // call GPU-code
  darts <<< blocksPerGrid, threadsPerBlock >>>
    (gpu_hits);

  // -------- error-checking (I had some problems)
  cudaThreadSynchronize();

  cudaError_t cudaLastError = cudaGetLastError();
  if(cudaLastError != cudaSuccess) {
    printf("error: %s\n",
            cudaGetErrorString(cudaLastError));
    cudaFree(gpu_hits);
    free(cpu_hits);
    return(EXIT_FAILURE);
  }
  // -------- end error-checking

  // fetch GPU-data
  cudaMemcpy(cpu_hits, gpu_hits, size,
             cudaMemcpyDeviceToHost);
```

```
  // add hits together
  double total_hits = 0.0;
  for(k = 0; k < threadsPerBlock * blocksPerGrid; k++)
    total_hits += cpu_hits[k];

  printf("time = %f\n", csecond() - t);

  // compute approximation
  printf("%e\n", 4.0 * total_hits /
    ((double) THROWS_PER_THREAD * threadsPerBlock *
              blocksPerGrid));

  // deallocate memory
  cudaFree(gpu_hits);
  free(cpu_hits);

  return 0;
}

% nvcc -O monte_carlo.cu csecond.c
% ./a.out
time = 1.180731
3.141572e+00

THROWS_PER_THREAD = 2000000
 threadsPerBlock   blocksPerGrid   wct
       512               1         3.93 s
       512/2             2         2.0  s
       512/4             4         1.2  s
       512/8             8         1.2  s

THROWS_PER_THREAD = 500000
       512               4         1.02 s
```

In the last cases all the four SP:s are used in parallel. The wct is 8.5 s for the equivalent non-parallel C-code. Note that the code is not so floating point intense, most of the instructions deal with integers.

Making longer runs leads to problems, the approximation becomes zero and one can see that the screen flickers. Here is the explanation (from the CUDA Release Notes):

(Windows and Linux) Individual GPU program launches are limited to a run time of less than 5 seconds on a GPU with a display attached. Exceeding this time limit usually causes a launch failure reported through the CUDA driver or the CUDA runtime. GPUs without a display attached are not subject to the 5 second runtime restriction. For this reason it is recommended that CUDA be run on a GPU that is NOT attached to a display and does not have the Windows desktop extended onto it. In this case, the system must contain at least one NVIDIA GPU that serves as the primary graphics adapter.

So one should have a compute server or an extra GPU.
I added the error-checking code and if you increase 2000000 enough you get: *error: the launch timed out and was terminated.*

Instead of computing the total number of hits (a reduction) using the CPU one could have done it in the GPU-code. This is a bit tricky, see the C Programming Guide for an example.

A potential problem is using control flow instructions (**for**, **if** etc) which may cause different threads to follow different execution paths. This leads to a significant slowdown, the code is serialized. I do not think this is a problem in the darts-code since all threads execute the same number of iterations.

It is possible to generate GPU-assembly code.

```
% nvcc -O -ptx monte_carlo.cu
```

generates **monte_carlo.ptx** (Parallel Thread Execution).
Here is part of the loop:

```
...
$Lt_0_2562:
 // <loop> Loop body line 20, nesting depth: 1,
 // iterations: 500000
...
 mov.f32       %f4, 0f2f800000;       // 2.32831e-10
 mul.f32       %f5, %f3, %f4;
...
 mov.f32       %f9, 0f2f800000;       // 2.32831e-10
 mul.f32       %f10, %f8, %f9;

 // %f11 = x * x    (TE's comment)
 mul.f32       %f11, %f10, %f10;

 // %f12 = %f11 + y * y    (TE)
 mad.f32       %f12, %f5, %f5, %f11;

 mov.f32       %f13, 0f3f800000;      // 1
 setp.le.f32   %p1, %f12, %f13;       // compare (TE)
...
 @%p2 bra      $Lt_0_2562;            // branch (TE)

 // hits[i_am] = n;    (TE)
 ld.param.u64  %rd1, [__cudaparm__Z5dartsPj_hits];
...
```

The SDK comes with a GUI-based profiler, **computeprof**. It provides execution times, usage of registers and hints.
Setting **blocksPerGrid** to two, for example, gives a warning: *Grid Size (2) is less than number of available SMs (4).*