Faculty of Science

Lab 1: A Gentle Introduction to CUDA

Cosmin E. Oancea

`cosmin.oancea@diku.dk`

After previous year slides of Rasmus Fonseca!

Department of Computer Science (DIKU)
University of Copenhagen

September 2023 PMPH Lab Slides

Get CUDA Up and Running

Option 1: Personal computer

- https://developer.nvidia.com/cuda-downloads
- Don't do this now!

Get CUDA Up and Running

Option 2: Same information is provided on Course's github:
Instructions for GPU servers are here:
https://github.com/diku-dk/howto/blob/main/servers.md

1. You need to be connected to VPN, if you are confused please see
   https://github.com/diku-dk/howto/blob/main/vpn.md

2. `$ ssh -l <ku-id> futharkhpa03fl.unicph.domain`

3. Type in you ku-id password
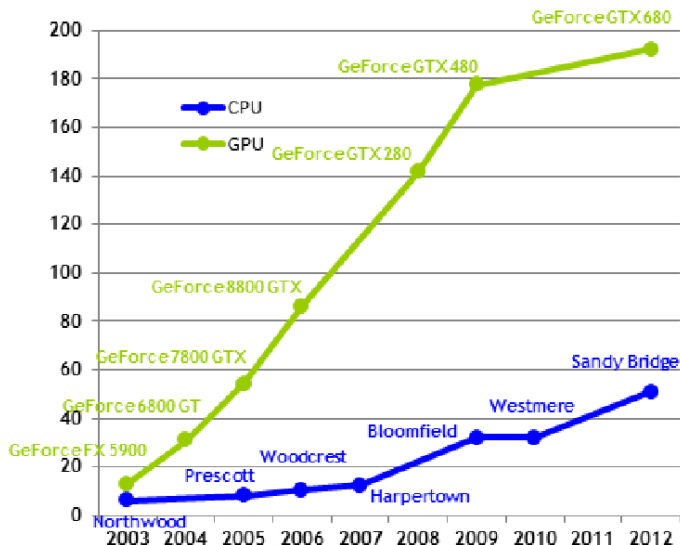
- Add the following to your `.bashrc` file:
  ```
  export CPATH=/usr/local/cuda/include:$CPATH
  export LIBRARY_PATH=/usr/local/cuda/lib64:$LIBRARY_PATH
  export LD_LIBRARY_PATH=/usr/local/cuda/lib64/$LD_LIBRARY_PATH
  export PATH=/usr/local/cuda/bin:$PATH
  ```

- Log out and in again, then you are ready to go:
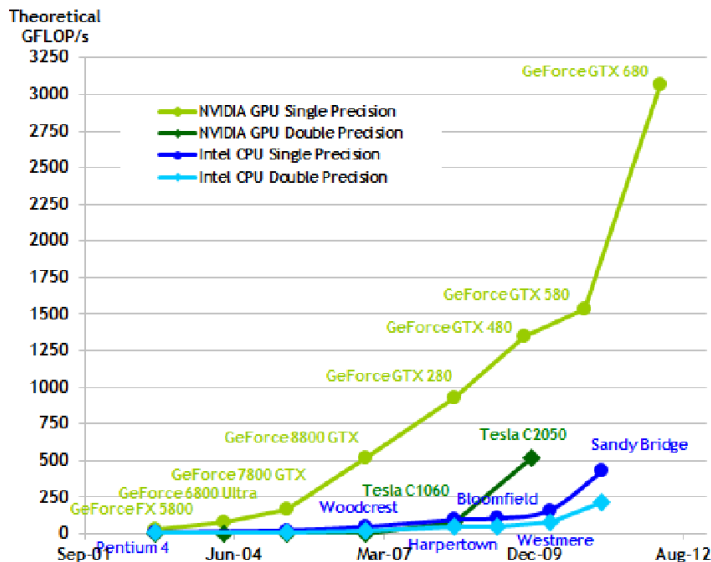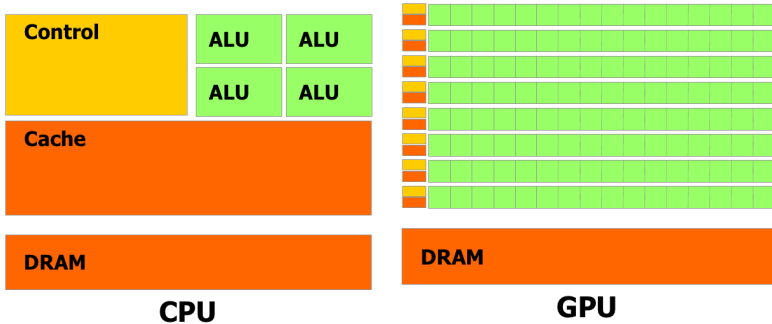 `$ nvcc ...`

Motivation for Using GPGPUs

Motivation for Using GPGPUs

Difficulties in Programming GPGPUs
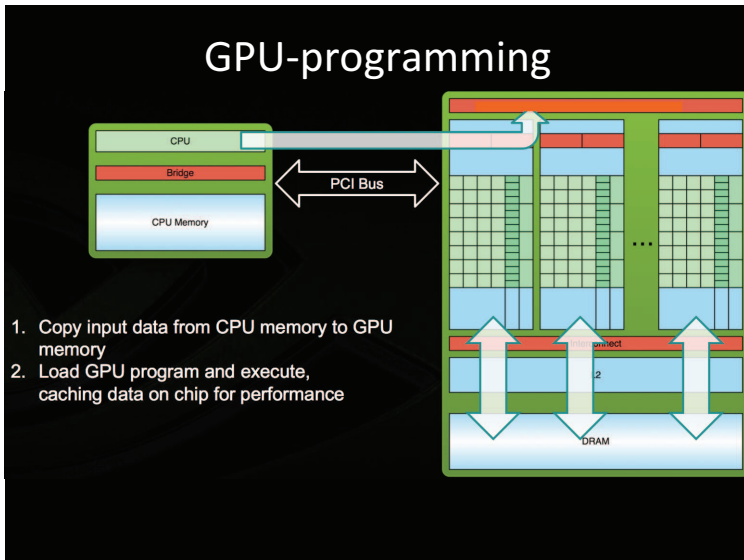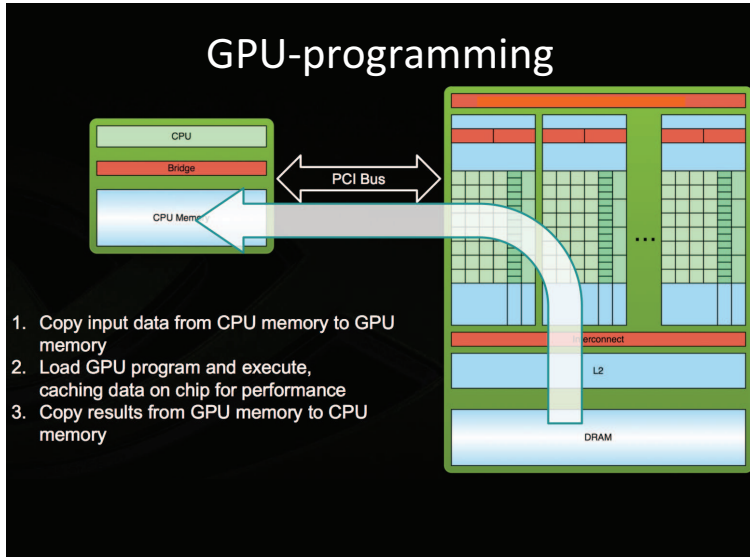
GPGPU programming

From http://on-demand.gputechconf.com/gtc-express/2011/presentations/GTC_Express_Sarah_Tariq_June2011.pdf

GPGPU programming

GPU-programming

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

GPGPU programming

GPU-programming

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

GPGPU programming

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

#include <cuda_runtime.h>

__global__ void squareKernel(float *d_in, float *d_out)
{
    const unsigned int tid = threadIdx.x; // access thread id
    d_out[tid] = d_in[tid]*d_in[tid];     // do computation
}

int main(int argc, char **argv)
{
    unsigned int num_threads = 32;
    unsigned int mem_size = sizeof(float) * num_threads;

    // allocate host memory
    float *h_in = (float *)malloc(mem_size);
    float *h_out = (float *) malloc(mem_size);

    // intialize the memory
    for (unsigned int i = 0; i < num_threads; ++i){
        h_in[i] = (float) i;
    }

    // allocate device memory
    float *d_in;
    float *d_out;
    cudaMalloc((void **) &d_in, mem_size);
    cudaMalloc((void **) &d_out, mem_size);

    // copy host memory to device
    cudaMemcpy(d_in, h_in, mem_size, cudaMemcpyHostToDevice);

    // execute the kernel
    squareKernel<<< 1, num_threads >>>(d_in, d_out);

    // copy result from device to host
    cudaMemcpy(h_out, d_out, sizeof(float) * num_threads, cudaMemcpyDeviceToHost);

    for (unsigned int i=0;i<num_threads; ++i){
        printf("%.1f\n",h_out[i]);
    }

    // cleanup memory
    free(h_in);
    free(h_out);
    cudaFree(d_in);
    cudaFree(d_out);

    return 0;
}
```

A Simple CUDA Program

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <cuda_runtime.h>

__global__ void squareKernel(float* d_in, float *d_out) {
    const unsigned int lid = threadIdx.x; // local id inside a block
    const unsigned int gid = blockIdx.x*blockDim.x + lid; // global id
    d_out[gid] = d_in[gid]*d_in[gid];      // do computation
}

int main(int argc, char** argv) {
    unsigned int N = 32;
    unsigned int mem_size = N*sizeof(float);

    // allocate host memory
    float* h_in  = (float*) malloc(mem_size);
    float* h_out = (float*) malloc(mem_size);

    // initialize the memory
    for(unsigned int i=0; i<N; ++i){
        h_in[i] = (float)i;
    }
```

A Simple CUDA Program (continuation)

```
    // allocate device memory
    float* d_in;
    float* d_out;
    cudaMalloc((void**)&d_in,  mem_size);
    cudaMalloc((void**)&d_out, mem_size);

    // copy host memory to device
    cudaMemcpy(d_in, h_in, mem_size, cudaMemcpyHostToDevice);

    // execute the kernel
    squareKernel<<< 1, N>>>(d_in, d_out);

    // copy result from ddevice to host
    cudaMemcpy(h_out, d_out, mem_size, cudaMemcpyDeviceToHost);

    // print result
    for(unsigned int i=0; i<N; ++i) printf("%.6f\n", h_out[i]);

    // clean-up memory
    free(h_in);        free(h_out);
    cudaFree(d_in);    cudaFree(d_out);
}
```

Save, Compile, Run

```
$ nvcc -O3 simpleCUDA.cu
```

On gpu01..3 you'll still need the -arch=compute_35 flag, i.e.,

```
$ nvcc -O3 -arch=compute_35 simpleCUDA.cu
```

```
$ ./a.out
```

Measuring Runtime

```
#include <sys/time.h>
#include <time.h>
int timeval_subtract(    struct timeval* result,
                         struct timeval* t2,struct timeval* t1) {
    unsigned int resolution=1000000;
    long int diff = (t2->tv_usec + resolution * t2->tv_sec) -
                    (t1->tv_usec + resolution * t1->tv_sec) ;
    result->tv_sec = diff / resolution;    result->tv_usec = diff % resolution;
    return (diff<0);
}
#define GPU_RUNS 100
int main() { ...
    unsigned long int elapsed; struct timeval t_start, t_end, t_diff;
    gettimeofday(&t_start, NULL);

    // execute the kernel
    for(int i=0; i<GPU_RUNS; i++) {
        squareKernel<<< 1, N>>>(d_in, d_out);
    } cudaThreadSynchronize();

    gettimeofday(&t_end, NULL);
    timeval_subtract(&t_diff, &t_end, &t_start);
    elapsed = (t_diff.tv_sec*1e6+t_diff.tv_usec) / GPU_RUNS;
    printf("Took %d microseconds (%.2fms)\n",elapsed,elapsed/1000.0);
...}
```

Trouble Ahead

This week assignment: *Write a CUDA program that maps the function* $(x/(x-2.3))^3$ *to the array* $[0,1,\ldots, 753411]$ *...*

This shouldn't be a problem with our program (adapt the kernel)

- GPU logical threads organized in a grid of blocks, in which the grid and the block can have up to three dimensions.
- However CUDA does not accept a block of size 32757
  - a *CUDA warp* is formed by 32 threads that execute SIMD.
  - a *a CUDA block* may contain up to 1024 threads (included); ideally the block size is a multiple of 32, but not necessarily.
  - Synchronization/communication is possible inside a CUDA block by means of barriers & scratchpad memory (shared memory).
  - Barrier synchronization is not possible across threads in different CUDA blocks, i.e., only by finishing the kernel!

Trouble Ahead

This week assignment: *Write a CUDA program that maps the function* $(x/(x-2.3))^3$ *to the array* $[0,1,...,\ 753411]$ *...*

This shouldn't be a problem with our program (adapt the kernel)

- GPU logical threads organized in a grid of blocks, in which the grid and the block can have up to three dimensions.
- However CUDA does not accept a block of size 32757
  - a *CUDA warp* is formed by 32 threads that execute SIMD.
  - a *CUDA block* may contain up to 1024 threads (included); ideally the block size is a multiple of 32, but not necessarily.
  - Synchronization/communication is possible inside a CUDA block by means of barriers & scratchpad memory (shared memory).
  - Barrier synchronization is not possible across threads in different CUDA blocks, i.e., only by finishing the kernel!
- Finally if the size of the computation does not matches exactly a multiple of block size, then you need to spawn extra threads, hence you need to add an `if` inside the kernel code, to make the extra threads iddle!

GPGPU in More Detail

- A set of Streaming Multiprocessors (SMs)

From deviceQuery:
 (15) Multiprocessors, (192) CUDA Cores/MP: 2880 CUDA Cores

- Each SM executes 1 'thread block' at a time.

- Each block has access to
  - Global memory (function arguments)

From deviceQuery:
 Total amount of global memory: 3072 MBytes

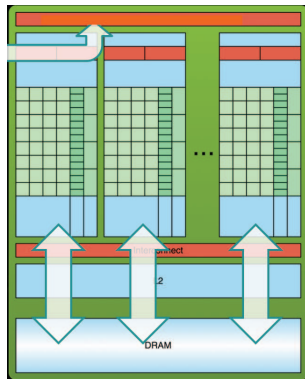 - Shared memory (`__shared__ int array[512]`)

From deviceQuery:
 Total amount of shared memory per block: 49152 bytes

  - Local memory (local variables)

From deviceQuery:
 Total number of registers available per block: 65536

Running Multiple Blocks

```
    unsigned int N = 32757;
    unsigned int mem_size   = N*sizeof(float);
    unsigned int block_size = 256;
    unsigned int num_blocks = ((N + (block_size - 1)) / block_size);

    // execute the kernel
    squareKernel<<< num_blocks, block_size>>>(d_in, d_out, N);

...

__global__ void squareKernel(float* d_in, float *d_out, int N) {
    const unsigned int lid = threadIdx.x; // local id inside a block
    const unsigned int gid = blockIdx.x*blockDim.x + lid; // global id
    if(gid < N) {
        d_out[gid] = d_in[gid]*d_in[gid];      // do computation
    }
}
```

Blank space for explanation

Blank space for explanation