HP-SEE

Introduction to GPU computing

www.hp-see.eu

Emanouil Atanassov
Grid Technologies and Applications Department
Institute of Information and Communication Technologies
Bulgarian Academy of Science
emanouil@parallel.bas.bg

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

The HP-SEE initiative is co-funded by the European Commission under the FP7 Research Infrastructures contract no. 261499

Outline

- What is GPGPU programming
- Advantages of using GPGPU programming
- How does it work
- Availability
- Problems and obstacles
- CUDA programming model
- GPU kernel example
- Conclusions

What is GPGPU programming

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- GPGPU means **General-purpose computing on graphics processing units** – a technique of using a GPU (a graphics card), to perform complex computations usually performed on CPU
- GPGPU leverages the high amount of transistors and high level of parallelism of contemporary graphics cards to achieve better overall efficiency.
- Contemporary graphics cards used for high-level gaming provide enormous amount of computational power, measured in Tflops
- This is achieved due to high number of processing elements, which are able to process high number of concurrent threads
- The two main producers are NVIDIA and AMD (ATI), who also provide speciliazed hardware for HPC installations (Tesla, Fermi, AMD FireStream) and software development tools
- Software development can be done using
  - OpenCL – cross-platform, can be used also on CPU
  - NVIDIA CUDA – only available for NVIDIA GPUs

Advantages of using GPGPU programming

- Due to high volume of sales of GPUs price is relatively low
- High power efficiency, low space requirements
- Example: TESLA M2050 - 448 CUDA cores, 3 GB memory, double Precision performance (peak) 515 Gflops, single precision performance (peak) 1.03 Tflops, memory bandwidth 148 GB/sec, Power Consumption 225W TDP
- Increasingly popular in top500 list, including the No 1 machine
- Improving support in popular libraries, applications and development suites
- Tools for automatic parallelisation become available

Problems and obstacles

- Memory size and bandwidth are limited and relatively low compared with the high number of concurrent threads executing.
- Porting a large piece of code is a daunting task
- Inter-node GPU – GPU communication not yet developed
- Synchronization and messaging between threads from different blocks not supported (yet).
- New features constantly added, some of them only supported on new hardware

Availability

- Most of the NVIDIA CPUs, including those found on laptops, support CUDA and OpenCL
- HPC cluster at IICT has 4 machines with NVIDIA GTX 295 (visible as 8 different computing devices). Users of the HPC cluster can access wn019.ipp.acad.bg with same username and password
- Latest installed version can be loaded with command
  - module load cuda
  - nvcc is the compiler
- Download and install the sdk containing many useful examples:
  - sh $CUDA_HOME/gpucomputingsdk_3.2.16_linux.run

How does it work

- CUDA introduces keywords that extend the C language
- GPU code organized in *kernels.*
- *When called, a kernel is N times in parallel by N different CUDA threads*
- A kernel is a C function, defined using the **__global__** declaration
- A kernel may call other functions, if they are defined with **__device__** declaration
- A kernel is invoked by the CPU code by specifying how many threads should be run in parallel.

How does it work

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- Example kernel definition:

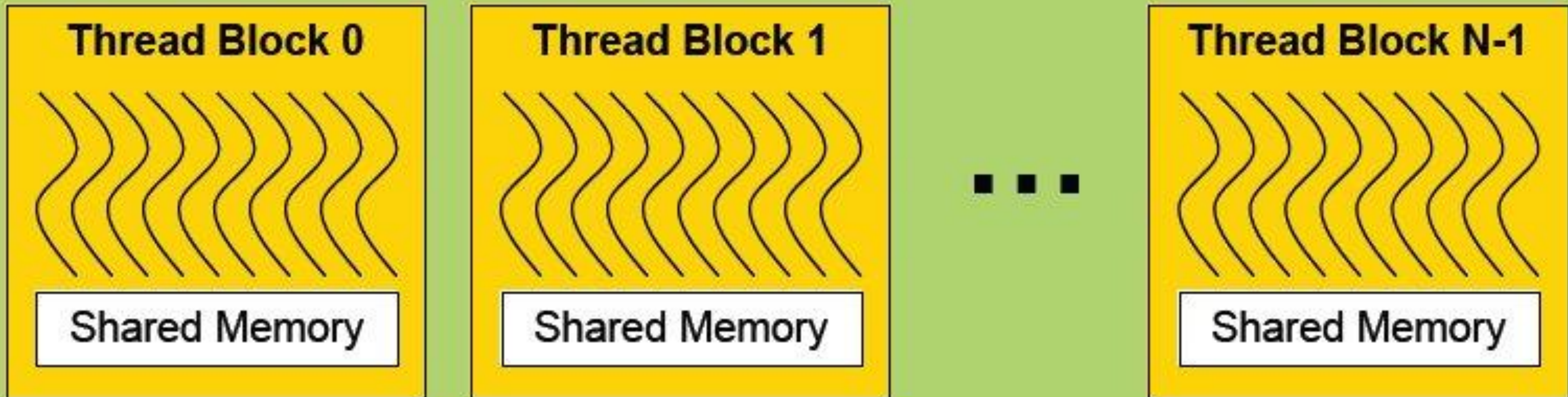
```
__global__ void multip(float A[N][N], float B[N][N],
float C[N][N]){
int i = threadIdx.x;
int j = threadIdx.y;
C[i][j] = A[i][j] * B[i][j];
}
int main() {
// 1 block of N * N * 1 threads
int blocks = 1;
dim3 threadsinblock(N, N);
multip<<<numBlocks, threadsinblock>>>(A, B, C);
}
```

Memory and execution model

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- The RAM dedicated to the GPU (**global** memory) has relatively high latency, but also high bandwidth
- There is no cache in the CPU sense, but there is fast "**shared**" memory "close" to the processing elements.
- Threads within the same block can use shared memory declared with __shared__ keyword to exchange data.
- **Texture** memory also available, optimised for 2D access.
- The **__synchthreads()** intrinsic function provides a barrier enabling synchronisation between threads from the same block.
- Several threads, e.g., 32, form a so-called *warp*.
- Threads within a warp are executed on one multiprocessor, following SIMD model.

Memory and execution model



Memory and execution model

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- Each thread has an ID that it uses to compute memory addresses and make control decisions:
  - float x = input[threadID];
- All __global__ and __device__ functions have access to these automatically defined variables
  - dim3 gridDim; - Dimensions of the grid in blocks (at most 2D)
  - dim3 blockDim; - Dimensions of the block in threads
  - dim3 blockIdx; - Block index within the grid
  - dim3 threadIdx; - Thread index within the block

Parallelization approach

HP-SEE
High-Performance Computing Infrastructure for South East Europe's Research Communities

- If computations inside a cycle are relatively independent from each other, the approach would be:
  - Copy data from CPU to GPU memory
  - Launch kernel with appropriate geometry
  - Copy data from GPU back to CPU
- Number of blocks and threads should be maximised, taking into account, however, that local variables are best put in shared memory and shared memory is limited.

Example program

```
#include <cuda.h>
__global__ void sum_test(int N,double *full_result, double*result){
        unsigned int tid = blockIdx.x * blockDim.x + threadIdx.x;
        unsigned int bid = blockIdx.x;
        double cf=exp((double)tid/(double)N);
        full_result[tid]=cf;
        __shared__ double  s_data[64];
        s_data[threadIdx.x]=cf;
        for ( int dist = blockDim.x/2; dist > 0; dist /= 2 ) {
           if ( threadIdx.x < dist ){
              s_data[ threadIdx.x ] += s_data[ threadIdx.x + dist ];
           }
              __syncthreads( );
        }
        if (threadIdx.x==0){
              result[bid]=s_data[0];
        }
```

```
int main(int argc,char**argv){
	int N=1280;	int gridsize=20;	int numthreads=64;
	dim3 grid=dim3(gridsize,1,1);	dim3 block=dim3(numthreads,1);
	double * full_results_h=(double*)malloc(sizeof(double)*N);
	double * full_results_d, *results_d;
	cudaMalloc(&full_results_d,sizeof(double)*N);
	double * results_h=(double*)malloc(sizeof(double)*gridsize);
	cudaMalloc(&results_d,sizeof(double)*gridsize);
	sum_test<<<grid,block>>>(N,full_results_d, results_d);
	cudaMemcpy(full_results_h, full_results_d,sizeof(double)*N, cudaMemcpyDeviceToHost);
	cudaMemcpy(results_h, results_d,sizeof(double)*gridsize, cudaMemcpyDeviceToHost);
	double full_s, s;
	int i;
	for (i=0,s=0.;i<gridsize;i++)  s+=results_h[i];
	for (i=0;i<N;i++)  full_s+=full_results_h[i];
	printf("%g %g \n",s/N,full_s/N);
	return 0;
}
```

Device management

- Example of device management:

```
int deviceCount;
cuDeviceGetCount(&deviceCount);
int device;
for (int device = 0; device < deviceCount; ++device){
  CUdevice cuDevice;
  cuDeviceGet(&cuDevice, device);
  int major, minor;
  cuDeviceComputeCapability(&major, &minor, cuDevice);
}
```

Compute Capability - above 1.3 allows **double**, above 2.0 – Fermi.

New features in CUDA 4.0

- Share GPUs between multiple CPU threads,e.g., with OpenMP
- Single thread can access all GPUs
- No-copy pinning of host RAM
- NVIDIA GPU Direct 1.0:
  - Direct access to GPU memory from other devices (Infiniband cards)
- NVIDIA GPU Direct 2.0:
  - Peer-to-peer access
  - Peer-to-peer transfers

Tools and software for CUDA

HP-SEE
High-Performance Computing Infrastructure
for South East Europe's Research Communities

- FFT: libcufft
- BLAS: libcublas
- Random numbers: libcurandorm
- Sparse matrix manipulations: libcusparse

- Debugger – cuda-dbg
- Application software: NAMD, ABINIT, parts of WRF,GROMACS

Most intense areas of application

- Financial mathematics
- DNA sequencing
- Oil and gas industry

- Up to date list at http://www.nvidia.com/object/cuda_app_tesla.html

Conclusions

- GPGPU codes become increasingly popular and GPGPU resources increase faster than CPU-based resources.
- Many of the widely used computational chemistry, linear algebra, fft, etc. codes are already ported and can be used without specific knowledge of GPGPU computing.
- Programming for the GPU has unique challenges, but there are large number of useful examples.