VRE for regional Interdisciplinary communities in Southeast Europe and the Eastern Mediterranean

Parallel programming with GPGPU coprocessors

Petar Jovanović
Institute of Physics Belgrade

The VI-SEEM project initiative is co-funded by the European Commission under the H2020 Research Infrastructures contract no. **675121**

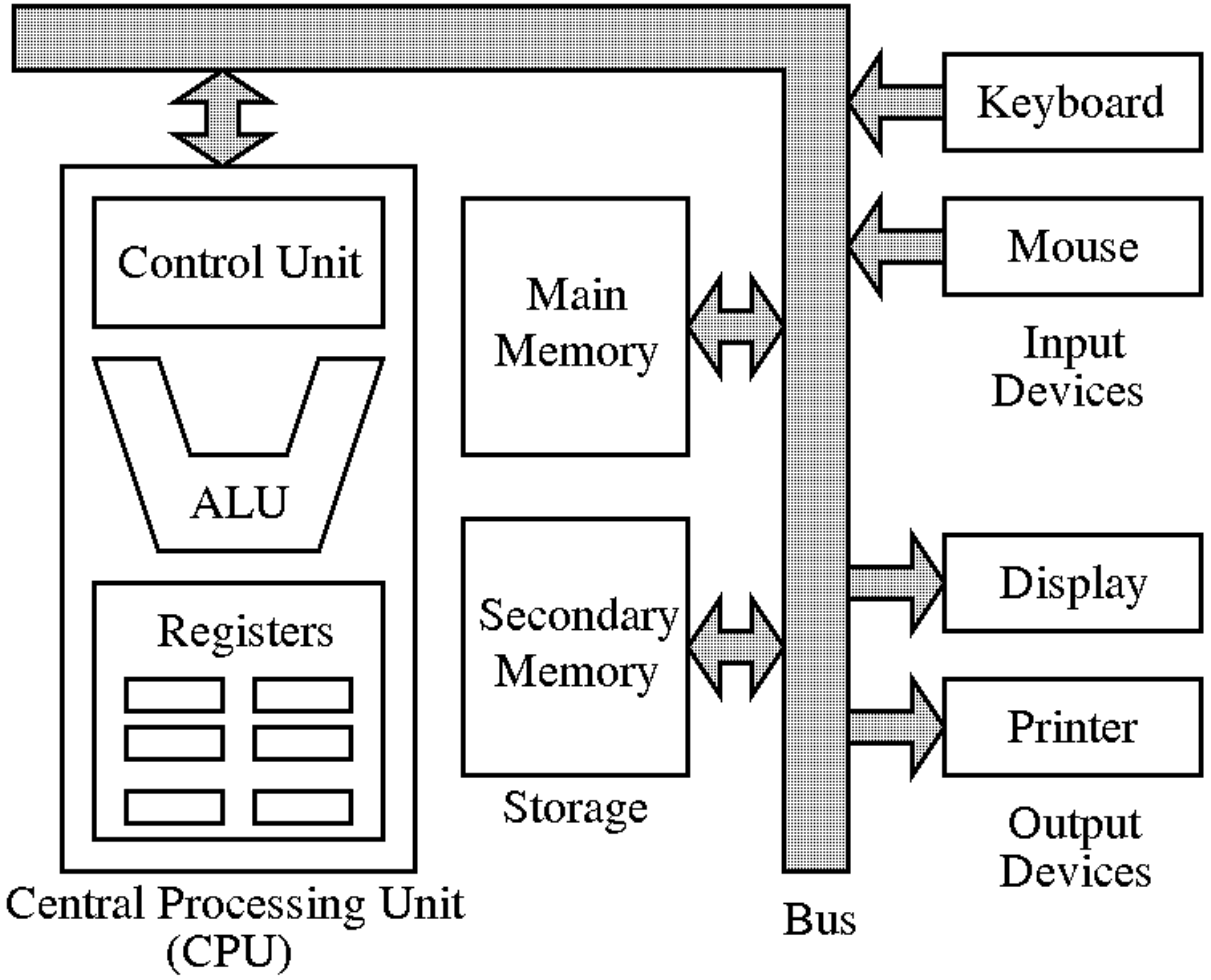
Agenda

- Introduction
- The von Neumann architecture
- CPU vs GPU architecture
- Heterogeneous execution model
- Code for GPUs
- CUDA kernel example
- GPU memory organization
- Matrix multiplication example

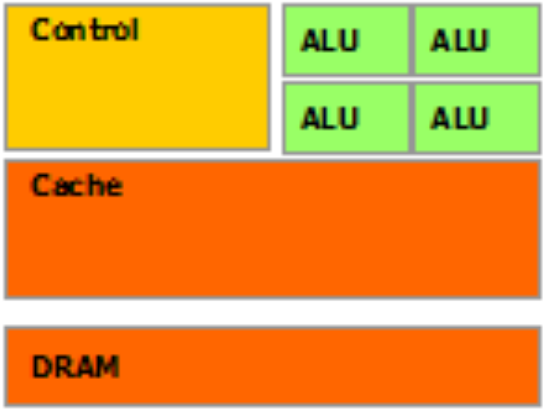
Introduction

- With the introduction of CUDA, graphical processing units (GPUs) became usable for general purpose computation.
- For some types of work GPU can bring significant speedup over traditional CPU.

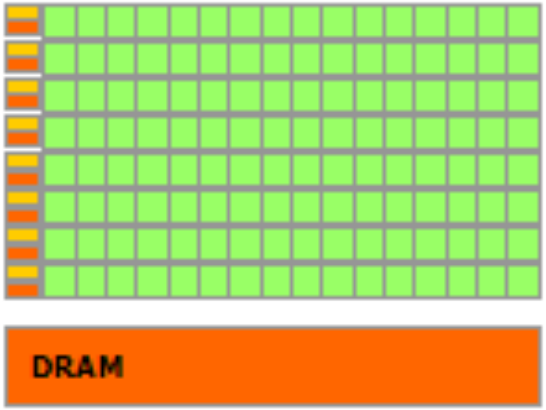
The von Neumann architecture



CPU vs GPU architecture



CPU



GPU

CPU (latency oriented design):

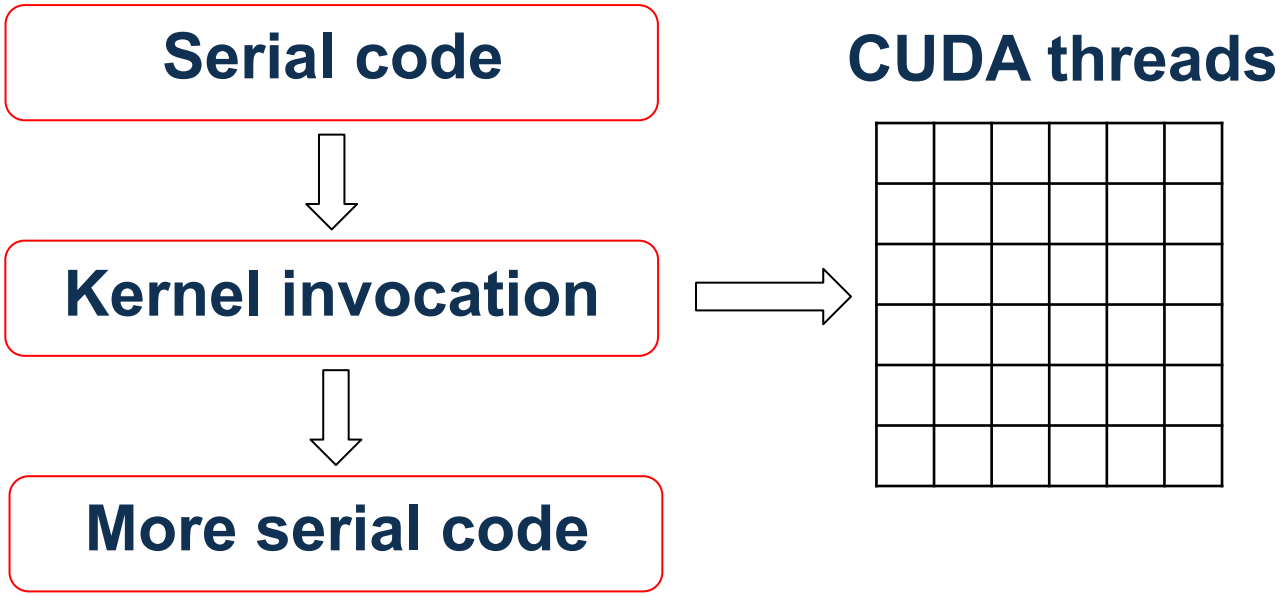
- Large caches
- Sophisticated control
- Powerful ALU

GPU (throughput oriented design):

- Small caches
- Simple control
- Energy efficient ALUs
- Latencies compensated by large number of threads

Heterogeneous execution model

- **Host** – a CPU which executes the main program in serial.
- **Device** – a GPU which executes parallel portions of the code.
- Memory spaces are separate*
  - Allocation and data movement is the responsibility of the programmer.



Code for GPUs

- CUDA C program is written as follows:
  - Serial parts in host C code
  - Parallel parts in device SIMD kernel C code
- Source code is compiled separately
  - Standard C/C++ code for the CPU
  - Device code in PTX – compiled just-in-time for the exact device
- Use the nvcc for compilation
  - PTX is an assembly format
  - Specific binary code for the GPU devices

CUDA kernel example

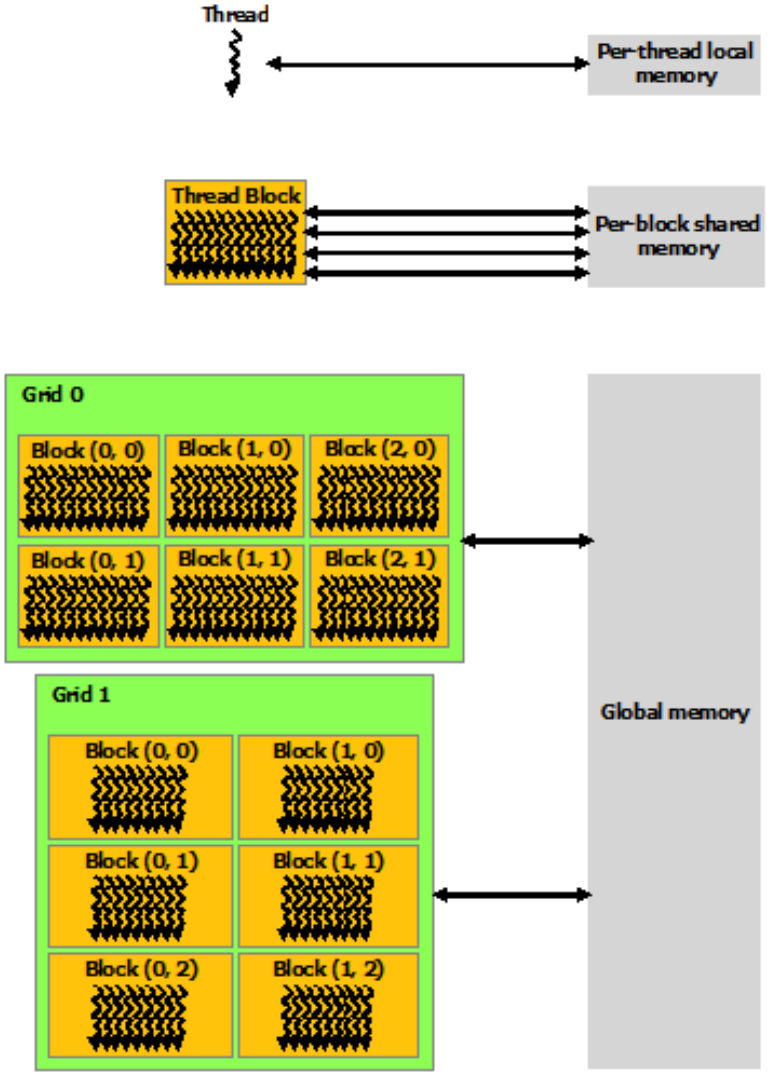
```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

GPU memory organization (1)

- Registers (local memory) are per-thread
  - **very low** latency, **very high** throughput
  - limited resource, used for automatic variables
- Shared memory (and L1 cache) is per-block
  - **low** latency, **high** throughput
  - can yield significant performance boost, depends on algorithm
  - programmer is responsible for its usage
  - shared/cache split can be controlled using the API
- Global memory is visible to all threads
  - **high** latency, **moderate** throughput
  - memory allocated with cudaMalloc is global
  - has the highest capacity

GPU memory organization (2)



Matrix multiplication example

- Simple version:

```
__global__
void matrixMulKernel(float* A, float* B, float* C, int width)
{
    int i;
    int row = blockIdx.y*blockDim.y+threadIdx.y;
    int col = blockIdx.x*blockDim.x+threadIdx.x;

    if ((row<width) && (col<width)) {
        float tmp = 0;
        for (i = 0; i < width; ++i)
            tmp += A[row*width+i]*B[i*width+col];
        C[row*width+col] = tmp;
    }
}
```

Matrix multiplication w/ shared memory

```
#define TILE_WIDTH 32

__global__
void matrixMulKernel(float* A, float* B, float* C, int width) {
          __shared__ float sA[TILE_WIDTH][TILE_WIDTH];
          __shared__ float sB[TILE_WIDTH][TILE_WIDTH];

          int bx=blockIdx.x, by=blockIdx.y;
          int tx=threadIdx.x, ty=threadIdx.y;
          int row = by*TILE_WIDTH+ty;
          int col = bx*TILE_WIDTH+tx;
          float tmp = 0;

          for (int i = 0; i < width/TILE_WIDTH; ++i) {
                    sA[ty][tx] = A[row*width+i*TILE_WIDTH+tx];
                    sB[ty][tx] = B[(i*TILE_WIDTH+ty)*width+col];
                    __syncthreads();
                    for (int j = 0; j < TILE_WIDTH; ++j) {
                              tmp += sA[ty][j]*sB[j][tx];
                    }
                    __syncthreads();
          }
          C[row*width+col] = tmp;
}
```

Questions

Thank you for your attention.