HPC.NRW

SEVERAL WAYS TO SAXPY

CUDA C/C++

Marius Neumann, Christian Schmidt

"CUDA C++ extends C++ by allowing the programmer to define C++ functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C++ functions."

NVIDIA, *CUDA C++ Programming Guide*

"CUDA C++ is C++ with GPU functionality."

Marius, *HPC.NRW GPU tutorials*

THE NVIDIA COMPILER NVCC

HPC.NRW

- part of NVIDIA Toolkit
- can compile standard C/C++ code
- standard suffix '.cu'

- `docs.nvidia.com/cuda/cuda-compiler-driver-nvcc`

Compiling and running:

```
$ nvcc -o gpu_code gpu_code.cu
$ ./gpu_code
```

Analogous to:

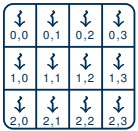
```
$ g++ -o cpu_only cpu_only.cpp
$ ./cpu_only
```

- data parallel function
  - executed on the device
  - called on the host
- new keywords and syntax
  - `__global__` modifier defining a kernel function
  - `<<<M,N>>>` kernel launch syntax

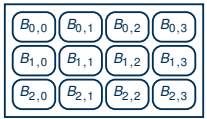
Launching a kernel:

```
__global__ void my_kernel(){
    doStuff();
}

int main(){
    my_kernel<<<M,N>>>();
}
```

Threads



Block: N Threads



Grid: M Blocks

LAUNCHING A KERNEL FUNCTION

- <<<M,N>>> starts kernel on *M* blocks with *N* threads per block
- kernel launched *MN* times
- each thread identified by threadIdx.x and blockIdx.x

```
void my_function(){
    for(int i=0; i<maxId; ++i){
        doStuff(i);
    }
}

int main(){
    my_function();
}
```

```
__global__ void my_kernel(){
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i<maxId) doStuff(i);
}

int main(){
    my_kernel<<<M,N>>>();
}
```

ALLOCATE MEMORY ON THE DEVICE

HPC.NRW

- data must be transferred to GPU
- memory allocation in GPU memory needed

```
cudaError_t cudaMalloc(void ** devPtr, size_t size)
```

- allocates memory on the device
- devPtr: Pointer to device memory
- size: allocation size in bytes
- returns error code cudaError_t

COPY DATA TO THE DEVICE


```
cudaError_t cudaMemcpy(void * dst, const void * src,
                       size_t count, enum cudaMemcpyKind kind)
```

- copies data between host and device
- dst/src: Pointer to destination/source, both may be host or device
- count: size in bytes to copy
- kind: type of transfer, e.g.: cudaMemcpyHostToDevice

WHY NOT BOTH?

HPC.NRW

```
cudaError_t cudaMallocManaged(void ** Ptr, size_t size)
```

- allocates memory on host and device
- automatic copying
- same syntax as cudaMalloc
- only available for Pascal architecture and later
- almost as fast as manual memory handling

THE SAXPY PROBLEM

- linear combination of two float arrays
- results written to third array

$$a \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_d \end{bmatrix} = \begin{bmatrix} a \cdot x_1 + y_1 \\ a \cdot x_2 + y_2 \\ \vdots \\ a \cdot x_d + y_d \end{bmatrix}$$

Typical C/C++ code:

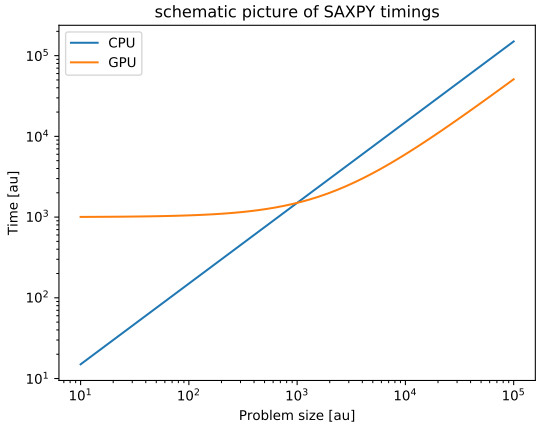
```
int main(){
    int N=6;
    float a=3.1415;
    float x[N]={1,2,3,4,5,6};
    float y[N]={7,8,9,0,1,2};
    float z[N];

    for(int i=0; i<N; i++){
        z[i]=a*x[i]+y[i];
    }
}
```

THE CUDA C/C++ SAXPY

HPC.NRW

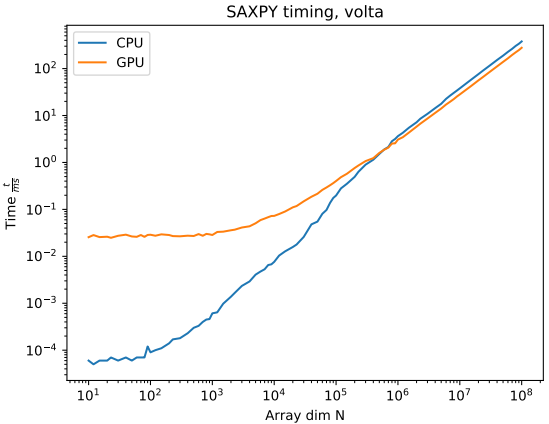
```
#include <cuda.h>
#define N 6

__global__ void saxpy_kernel(float a,
  float* x, float* y, float* z){
    int i = threadIdx.x;
    z[i] = a*x[i]+y[i];
}

int main(){
    float a = 3.1415;
    float x[N] = {1,2,3,4,5,6};
    float y[N] = {7,8,9,0,1,2};
    float z[N];
    float *d_x, *d_y, *d_z;
    cudaMalloc(&d_x, N*sizeof(float));
    cudaMalloc(&d_y, N*sizeof(float));
    cudaMalloc(&d_z, N*sizeof(float));

    cudaMemcpy(d_x, x, N*sizeof(float),
      cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(float),
      cudaMemcpyHostToDevice);

    saxpy_kernel<<<1,N>>>(a, d_x, d_y, d_z);

    cudaMemcpy(z, d_z, N*sizeof(float),
      cudaMemcpyDeviceToHost);
}
```

TIMINGS: MODEL



- CPU
  - linear
- GPU
  - small N: constant
  - large N: linear and fast

TIMINGS: REALITY



- CPU
  - almost linear
- GPU
  - small *N*: constant
  - large *N*: linear and fast
- varies with system configuration