COSC 4397
Parallel Computations
An Introduction to CUDA programming (I)

Edgar Gabriel
Spring 2010

Disclaimer

• Material for this lecture has been adopted based on various sources
  - Matt Heavener, CS, State Univ. of NY at Buffalo
    http://www.cse.buffalo.edu/faculty/miller/Courses/CSE710/heavner.pdf
CUDA overview

- CUDA program consists of one or more phases
  - sequential phases executed on CPU
  - data parallel phases executed on GPU
  - regular C code
  - compiler needs to know which portions have to be compiled for the CPU, which ones have to be compiled for the GPU
- Functions executed on the GPU are often called *kernels*
- Each kernel is supposed to be executed by a large number of *threads*
- Thread blocks are grouped into *thread blocks*.
- Thread blocks can be grouped in *grids*
- Blocks and grids may be 1-D, 2-D, or 3-D

CUDA Kernels (II)
Hello World: Vector Addition

- **Sequential code:**
  
  ```c
  int main ( int argc, char **argv )
  {
    int A[N], B[N], C[N];
    for ( i=0; i<N; i++ ) {
      C[i] = A[i] + B[i];
    }
    return (0);
  }
  ```

  CUDA: replace the loop by N threads each executing on element of the vector add operation

Hello World: Vector Addition

- **CUDA:** replace the loop by N threads each executing on element of the vector add operation
- **Question:** How does each thread know which elements to execute?
  - `threadIdx`: each thread has an id which is unique in the thread block
    - of type `dim3`, which is a
      ```c
      struct {
        int x, y, z;
      } dim3;
      ```
  - `blockDim`: Total number of threads in the thread block
    - remember that a thread block can be 1D, 2D or 3D
Hello World: Vector Addition

• Initial CUDA kernel:

```c
void vecadd ( int *d_A, int *d_B, int* d_C)
{
    int i = threadIdx.x;
    d_C[i] = d_A[i] + d_B[i];
    return;
}
```

Assuming a 1-D thread block
- only x-dimension used

• This code is limited by the maximum number of threads in a thread block
  - CUDA 1.3: 512 threads max.
  - if vector is longer, we have to create multiple thread blocks

How does the compiler now which code to compile for CPU and which one for GPU?

• Specifiers tell compiler where function will be executed
  -> compiler can generate code for corresponding processor

• Executed on CPU, called form CPU (default if not specified)
  __host__ void func(...)

• CUDA kernel to be executed on GPU, called from CPU
  __global__ void func(...);

• CUDA kernel to be executed on GPU, called from GPU
  __device__ void func(...);
Hello World: Vector Addition

so the CUDA kernel is in reality:

```c
__global__ void vecAdd (int *d_A, int *d_B, int * d_C)
{
    int i = threadIdx.x;
    d_C[i] = d_A[i] + d_B[i];
    return;
}
```

Note:
- d_A, d_B, and d_C are in global memory
- int i is in local memory of the thread

CUDA Memory model

- GPU can not use data from the memory of the CPU
- Data has to be copied explicitly into the GPU global memory
CUDA Memory Model (II)

- **cudaError_t cudaMalloc(void** * devPtr, size_t size)**
  - Allocates size bytes of device(global) memory pointed to by *devPtr
  - Returns cudaSuccess for no error
- **cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind)**
  - Dst = destination memory address
  - Src = source memory address
  - Count = bytes to copy
  - Kind = type of transfer ("cudaMemcpyHostToDevice", "cudaMemcpyDeviceToHost", "cudaMemcpyDeviceToDevice")
- **cudaError_t cudaFree(void** * devPtr)**
  - Frees memory allocated with cudaMalloc

```c
int main (int argc, char **argv) {
float a[N], b[N], c[N];
float *d_a, *d_b, *d_c;

cudaMalloc( &d_a, N*sizeof(float));
cudaMalloc( &d_b, N*sizeof(float));
cudaMalloc( &d_c, N*sizeof(float));
cudaMemcpy( d_a, a, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy( d_b, b, N*sizeof(float), cudaMemcpyHostToDevice);
vecAdd<<<1, N>>>(d_a, d_b, d_c);

cudaMemcpy(d_c, c, N*sizeof(float), cudaMemcpyDeviceToHost);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
}

__global__ void vecAdd ( int *d_A, int *d_B, int* d_C)
{ 
  int i = threadIdx.x;
  d_C[i] = d_A[i] + d_B[i];
  return;
}
```
CUDA Kernels (III)

- Kernels are called with the `<<<>>>` syntax
  `<<<Dg, Db, Ns, S>>>`

Where:
- $D_g =$ dimensions of the grid (type `dim3`)
- $D_b =$ dimensions of the block (type `dim3`)
- $N_s =$ number of bytes shared memory dynamically allocated / block (type `size_t`). 0 default
- $S =$ associated cudaStream. 0 default

- last two arguments often ignored/not given

---

How to create multiple thread blocks

```c
int main (int argc, char **argv) {
    float a[N], b[N], c[N];
    float *d_a, *d_b, *d_c;

    cudaMalloc( &d_a, N*sizeof(float));
    cudaMalloc( &d_b, N*sizeof(float));
    cudaMalloc( &d_c, N*sizeof(float));
    cudaMemcpy( d_a, a, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, b, N*sizeof(float), cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(512); // 1-D array of threads
    dim3 blocksPerGrid(N/512); // 1-D grid

    vecAdd<<<blocksPerGrid, threadsperBlock>>>(d_a, d_b, d_c);
    cudaMemcpy(d_c, c, N*sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
}
```
How to create multiple thread blocks (II)

```c
__global__ void vecAdd ( int *d_A, int *d_B, int* d_C)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    d_C[i] = d_A[i] + d_B[i];
    return;
}
```

ID of the thread block that this thread is part of

Number of threads in a thread block

Using more than one element per thread

```c
#define NUMELEMENTS 4

int main (int argc, char **argv) {
    ...
    dim3 threadsPerBlock(512); // 1-D array of threads
    dim3 blocksPerGrid(N/(512*NUMELEMENTS)); // 1-D grid
    vecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c);
    ...
}

__global__ void vecAdd ( int *d_A, int *d_B, int* d_C)
{
    int i, start = blockIdx.x * blockDim.x + threadIdx.x;
    for ( i=start*NUMELEMENTS; i<(start+1)*NUMELEMENTS; i++)
        d_C[i] = d_A[i] + d_B[i];
    return;
}  
```
Second example: Matrix multiply

```c
int main ( int argc, char **argv )
{
    float A[N][N], B[N][N], C[N][N];

    for ( i=0; i<N; i++) {
        for ( j=0; j<N; j++ ) {
            for ( k=0; k<N; k++ ) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    return (0);
}
```

CUDA: each thread determines the result for one element C[i][j]

---

Matrix Multiply Reformulated

- Mapping of the 2-D data structures to the 1-D address space
  - Assuming that the rows of each Matrix A, B, and C are contiguous in memory
  - Assuming row-major ordering

```c
float A[N][N], B[N][N], C[N][N];

for ( i=0; i<N; i++) {
    for ( j=0; j<N; j++ ) {
        float ctemp = 0.0;
        for ( k=0; k<N; k++ ) {
            ctemp += A[i*N+k] * B[k*N+j];
        }
        C[i*N+j] = ctemp;
    }
}
```
CUDA Kernel v1

```c
__global__ void MatrixMul ( float *A_d, float *B_d, float *C_d,
    int width) {

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    float pvalue = 0.0;

    for ( int k=0; k<width; k++) {
        pvalue += A_d[ty*width+k] * B_d[k*tx+ty];
    }
    C_d[ty*width+tx] = pvalue;

    return;
}
```

CUDA: 2-D thread block

 limitation: only 1 thread block,
 -> limits size of the matrices that can be used

CUDA Kernel v1 (II)

- And the corresponding part of the host code which calls the kernel:

```c
    cudaMalloc( &d_a, N*N*sizeof(float));
    cudaMalloc( &d_b, N*N*sizeof(float));
    cudaMalloc( &d_c, N*N*sizeof(float));
    cudaMemcpy( d_a, A, N*N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, B, N*N*sizeof(float), cudaMemcpyHostToDevice);

dim3 threadsPerBlock(N,N); // 2-D array of threads
dim3 blocksPerGrid(1); // 1 thread block specified as 1-D array
MatrixMul<<<dimGrid,dimBlock>>>( d_A, d_B, d_C, N);
    cudaMemcpy(d_C, C, N*N*sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
```
CUDA Kernel v2

- Use an arbitrary number of thread blocks
- Each thread calculates one element of the result

```c
__global__ void MatrixMul (float *A_d, float *B_d, float *C_d, 
    int width) {
    int tx = blockIdx.x*blockDim.x + threadIdx.x;
    int ty = blockIdx.y*blockDim.y + threadIdx.y;
    float pvalue=0;
    for (int k=0; k<width; k++) {
        pvalue += A_d[ty*width+k] * B_d[k*width+tx];
    }
    C_d[ty*width+tx] = pvalue;
    return;
}
```

- Example: 2-D grid of thread blocks, i.e.
  `gridDim.x = 2` and `gridDim.y = 2`
- Each thread block has 2-D array of threads, i.e.
  `blockDim.x = 2` and `blockDim.y = 2`
CUDA kernel v3

- Reformulate sequential code such that the result is calculated in chunks of TILE_WIDTH * TILE_WIDTH
- Each CUDA thread will calculate one such tile

```
for ( i=0; i<width; i+=TILE_WIDTH ) {
    for ( j=0; j<width; j+=TILE_WIDTH ) {
        for ( ii=i; ii<i+TILE_WIDTH; i++ ) {
            for ( jj=j; jj<j+TILE_WIDTH; j++ ) {
                for ( k=0; k<width; k++ ) {
                    c[ii][jj] += a[ii][k] * b[k][jj];
                }
            }
        }
    }
}
```

CUDA Kernel v3a

- Assuming one thread block

```
__global__ void MatrixMul ( float *A_d, float *B_d, float *C_d, int width) {
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    float pvalue;
    for ( int i=tx*TILE_WIDTH; i<(tx+1)*TILE_WIDTH; i++ ) {
        for ( int j=ty*TILE_WIDTH; j<(ty+1)*TILE_WIDTH; j++ ) {
            pvalue = 0.0;
            for ( int k=0; k<width; k++ ) {
                pvalue += A_d[i*width+k] * B_d[k*width+j];
            }
            C_d[i*width+j] = pvalue;
        }
    }
}
```
CUDA Kernel v3b

- Assuming multiple thread blocks

```
__global__ void MatrixMul(float *A_d, float *B_d, float *C_d, int width) {
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    float pvalue;
    for (int i=tx*TILE_WIDTH; i<(tx+1)*TILE_WIDTH; i++) {
        for (int j=ty*TILE_WIDTH; j<(ty+1)*TILE_WIDTH; j++) {
            pvalue = 0.0;
            for (int k=0; k<width; k++) {
                pvalue += A_d[i*width+k] * B_d[k*width+j];
            }
            C_d[i*width+j] = pvalue;
        }
    }
}
```

CUDA Kernel v3b (II)

- And the corresponding part of the host code which calls the kernel:

```
... 
cudaMalloc(&d_a, N*N*sizeof(float));
cudaMalloc(&d_b, N*N*sizeof(float));
cudaMalloc(&d_c, N*N*sizeof(float));
cudaMemcpy(d_A, A, N*N*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_B, B, N*N*sizeof(float),cudaMemcpyHostToDevice);

dim3 threadsPerBlock(20,20);// each thread block has 20x20 threads
dim3 blocksPerGrid(N/(20*TILE_WIDTH),N/(20*TILE_WIDTH));
MatrixMul<<<dimGrid,dimBlock>>>(d_A, d_B, d_C, N);
cudaMemcpy(d_C, C, N*N*sizeof(float),cudaMemcpyDeviceToHost);
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
... 
```
CUDA Kernel v3b (III)

- Trade off between TILE_WIDTH and Number of threads per thread block
  - typically driven by hardware limits

- E.g. for the GT200:
  - Maximum number of threads in a thread block: 512
  - Maximum number of threads on an SM: 1024
  - Maximum number of thread blocks on an SM: 8

CUDA Kernel v3b (IV)

- Example: should we use 8x8, 16x16 or 32x32 threads per thread block?

- 8x8:
  - 64 threads per thread block
  - One could have \(1024/(8\times8) = 12\) thread blocks per SM
     - exceeds the allowed limit of thread blocks per SM
  - Using 8 thread blocks with 8x8 threads = 512 threads
     - only half of the possible number of threads per SM
CUDA Kernel v3b (V)

- 16x16:
  - 256 threads per thread block
  - per SM we can have \( \frac{1024}{(16 \times 16)} \) thread blocks = 4 blocks
    - within the allowed limits
    - maximum number of threads per SM utilized.

- 32x32:
  - 1024 threads per thread block
  - exceeds the maximum number of threads per thread block (512)

CUDA kernel v4

- Problem with the CUDA kernels discussed so far for matrix multiply operations:
  - require loading two data items for each inner loop iteration
  - each inner loop iteration contains a multiply and add operation
  - 2 loads for 2 floating pointer operations - 1 load per op.

- GT200 Memory bandwidth: 141.7 GB/s
  - with 4 bytes per floating point number: ~35 GFLOPS

- GT200 Maximum peak performance: 933 GFLOPS
  - Memory bandwidth limits the maximum performance

CUDA kernel v4 (II)

- Same elements of $A_d$ and $B_d$ loaded multiple times
- To circumvent memory limitations
  - use caches on regular CPUs
  - use shared memory in CUDA architectures
- Shared memory:
  - memory that can be accessed and shared by all threads in a thread-block
  - Typically small, e.g. for GT200 16kB
  - much faster to access than global memory

CUDA variable type qualifiers

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic scalar variables</td>
<td>Register</td>
<td>Thread</td>
<td>Kernel</td>
</tr>
<tr>
<td>Automatic array variables</td>
<td>Local</td>
<td>Thread</td>
<td>Kernel</td>
</tr>
<tr>
<td><strong>device</strong>, <strong>shared</strong></td>
<td>Shared</td>
<td>Block</td>
<td>Kernel</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>Global</td>
<td>Grid</td>
<td>Application</td>
</tr>
<tr>
<td><strong>device</strong>, <strong>constant</strong>, <strong>constant</strong></td>
<td>Constant</td>
<td>Grid</td>
<td>Application</td>
</tr>
</tbody>
</table>
CUDA variable type qualifiers

```c
__global__ void MatrixMul ( float *A_d, float *B_d, float *C_d, 
    int width) {
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    float pvalue;
    ...
}
```

- `tx, ty, pvalue` are automatic scalar variables:
  - per thread, hold in a register each
- `A_d, B_d, C_d, width` global memory

CUDA kernel v4 (III)

- Idea: Copy rows/columes of `A_d` and `B_d` into shared memory
  - first access will be as expensive as accessing `A_d` and `B_d` in the previous code versions
  - subsequent access to the same row/column will be much faster
- Reuse can be achieved multiple ways
  - same thread reuses the same data, e.g. as in the CUDA kernel v3
    - difficult to maintain if threads do not share data, since size of shared memory is limited
  - multiple threads in a thread block reuse data of each other
- version 4: each thread calculates just 1 element, but uses shared memory and threads in a thread block coordinate access to shared data items.
```c
__global__ void MatrixMul ( float *A_d, float *B_d, float *C_d,
                           int width) {
    // Assuming
    // - one thread block with THREADS_PER_BLOCK x THREADS_PER_BLOCK
    // threads.
    // - width = THREADS_PER_BLOCK
    __shared__ float A_s[THREADS_PER_BLOCK][THREADS_PER_BLOCK];
    __shared__ float B_s[THREADS_PER_BLOCK][THREADS_PER_BLOCK];

    int tx = threadIdx.x; int ty = threadIdx.y;
    float pvalue=0.0;
    A_s[tx][ty] = A_d[tx*width + ty];
    B_s[tx][ty] = B_d[tx*width + ty];
    __syncthreads();
    for ( i=0; i< width; i++ ) {
        pvalue += A_s[tx][i] * B_s[i][ty];
    }
    C[tx*width+ty] = pvalue;
```

**CUDA kernel v4a**

- Number of read accesses to global memory:
  \[ 2 \times \text{width} \times \text{width} \]

- For comparison: number of accesses to global memory in CUDA kernel v2
  \[ 2 \times \text{width} \times \text{width} \times \text{width} \]
CUDA kernel v4b

- matrix will be typically larger than the number of threads held within a thread block
  - will require multiple thread blocks
  - each thread block has to load entire rows/columns of the matrixes into shared memory to be able to calculate the result of an element
    - not able to hold the entire row/column in shared memory due to size limitation
    - each thread will have to load more than one element per matrix
    - will require a multi-stage procedure

```c
__global__ void MatrixMul ( float *A_d, float *B_d, float *C_d, int width) {

    __shared__ float A_s[THREADS_PER_BLOCK][THREADS_PER_BLOCK];
    __shared__ float B_s[THREADS_PER_BLOCK][THREADS_PER_BLOCK];

    int tx = threadIdx.x; int ty = threadIdx.y;
    int bx = blockIdx.x; int by = blockIdx.y;
    int Row = bx * THREADS_PER_BLOCK + tx; // Row for this element
    int Col = by * THREADS_PER_BLOCK + ty; // Columns for this element
    float pvalue=0.0;

    for ( int m=0; m < width/THREADS_PER_BLOCK; m++ ) {
        A_s[tx][ty] = A_d[Row*width + (m*THREADS_PER_BLOCK + ty)];
        B_s[tx][ty] = B_d[(m*THREADS_PER_BLOCK + ty)*width + Col];
        __syncthreads();
        for ( i=0; i< THREADS_PER_BLOCK; i++ ) {
            pvalue += A_s[tx][i] * B[i][ty];
        }
        __syncthreads();
    }

    C[tx*width+ty] = pvalue;
}
```
e.g. for width=4 and THREADS_PER_BLOCK=2

<table>
<thead>
<tr>
<th>m=0</th>
<th>m=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_{0,0} A_{d_{0,0}}</td>
<td></td>
</tr>
<tr>
<td>↓ A_{s_{0,0}}</td>
<td></td>
</tr>
<tr>
<td>B_{d_{0,0}}</td>
<td></td>
</tr>
<tr>
<td>↓ B_{s_{0,0}}</td>
<td></td>
</tr>
<tr>
<td>pvalue_{0,0} +=</td>
<td></td>
</tr>
<tr>
<td>A_{s_{0,0}}*B_{s_{0,0}}</td>
<td></td>
</tr>
<tr>
<td>A_{s_{1,0}}*B_{s_{0,1}}</td>
<td></td>
</tr>
<tr>
<td>A_{d_{2,0}}</td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td></td>
</tr>
<tr>
<td>A_{s_{0,0}}</td>
<td></td>
</tr>
<tr>
<td>B_{s_{0,0}}</td>
<td></td>
</tr>
<tr>
<td>pvalue_{0,0} +=</td>
<td></td>
</tr>
<tr>
<td>A_{s_{0,0}}*B_{s_{0,0}}</td>
<td></td>
</tr>
<tr>
<td>A_{s_{1,0}}*B_{s_{0,1}}</td>
<td></td>
</tr>
<tr>
<td>T_{1,0} A_{d_{1,0}}</td>
<td></td>
</tr>
<tr>
<td>↓ A_{s_{1,0}}</td>
<td></td>
</tr>
<tr>
<td>B_{d_{1,0}}</td>
<td></td>
</tr>
<tr>
<td>↓ B_{s_{1,0}}</td>
<td></td>
</tr>
<tr>
<td>pvalue_{1,0} +=</td>
<td></td>
</tr>
<tr>
<td>A_{s_{0,0}}*B_{s_{1,0}}</td>
<td></td>
</tr>
<tr>
<td>A_{s_{1,0}}*B_{s_{0,1}}</td>
<td></td>
</tr>
<tr>
<td>A_{s_{1,0}}</td>
<td></td>
</tr>
<tr>
<td>B_{s_{1,0}}</td>
<td></td>
</tr>
<tr>
<td>pvalue_{1,0} +=</td>
<td></td>
</tr>
<tr>
<td>A_{s_{0,0}}*B_{s_{1,0}}</td>
<td></td>
</tr>
<tr>
<td>A_{s_{1,0}}*B_{s_{0,1}}</td>
<td></td>
</tr>
<tr>
<td>T_{0,1} A_{d_{0,1}}</td>
<td></td>
</tr>
<tr>
<td>↓ A_{s_{0,1}}</td>
<td></td>
</tr>
<tr>
<td>B_{d_{0,1}}</td>
<td></td>
</tr>
<tr>
<td>↓ B_{s_{0,1}}</td>
<td></td>
</tr>
<tr>
<td>pvalue_{0,1} +=</td>
<td></td>
</tr>
<tr>
<td>A_{s_{0,0}}*B_{s_{0,0}}</td>
<td></td>
</tr>
<tr>
<td>A_{s_{1,0}}*B_{s_{0,1}}</td>
<td></td>
</tr>
<tr>
<td>A_{s_{0,1}}</td>
<td></td>
</tr>
<tr>
<td>B_{s_{0,1}}</td>
<td></td>
</tr>
<tr>
<td>pvalue_{0,1} +=</td>
<td></td>
</tr>
<tr>
<td>A_{s_{0,0}}*B_{s_{0,0}}</td>
<td></td>
</tr>
<tr>
<td>A_{s_{1,0}}*B_{s_{0,1}}</td>
<td></td>
</tr>
<tr>
<td>T_{1,1} A_{d_{1,1}}</td>
<td></td>
</tr>
<tr>
<td>↓ A_{s_{1,1}}</td>
<td></td>
</tr>
<tr>
<td>B_{d_{1,1}}</td>
<td></td>
</tr>
<tr>
<td>↓ B_{s_{1,1}}</td>
<td></td>
</tr>
<tr>
<td>pvalue_{1,1} +=</td>
<td></td>
</tr>
<tr>
<td>A_{s_{0,0}}*B_{s_{1,0}}</td>
<td></td>
</tr>
<tr>
<td>A_{s_{1,0}}*B_{s_{0,1}}</td>
<td></td>
</tr>
<tr>
<td>A_{s_{1,1}}</td>
<td></td>
</tr>
<tr>
<td>B_{s_{1,1}}</td>
<td></td>
</tr>
<tr>
<td>pvalue_{1,1} +=</td>
<td></td>
</tr>
<tr>
<td>A_{s_{0,0}}*B_{s_{1,0}}</td>
<td></td>
</tr>
<tr>
<td>A_{s_{1,0}}*B_{s_{0,1}}</td>
<td></td>
</tr>
</tbody>
</table>

Shared Memory limitations

- each SM has 16KB of shared memory on the GT200
- each SM can execute up to 8 thread blocks that need to share the shared memory
  - -> 2 KB KB per thread block
- If a thread block has 16x16 threads:
  - A_s will take 16x16x4 bytes = 1 KB
  - B_s will take 16x16x4 bytes = 1 KB
- Similar limitations can occur with the number of registers available per thread
Summary

- CUDA: extensions to C that allow to utilize Nvidia GPUs
  - single instruction multiple data (SIMD) model
  - two-level hierarchy of threads (threads, thread-blocks)
  - easy/fast synchronization of threads in a thread-block
    - using __syncthreads()
  - explicit memory control:
    - using cudaMemcpy(), cudaMemcpy(), cudaMemcpy() for global memory
    - using specifiers to denote shared memory (__shared__) and constant memory (__constant__)
- Trade-offs required to deal with hardware limits
  - number of thread blocks per SM, number of threads per SM
  - amount of shared memory per thread-block

What else is there

- Improving performance of global memory through memory coalescing
- Dealing with if-then-else statements to avoid divergence of threads
- Special hardware trigonometry functions
- CUDA aware numerical libraries, e.g. cudaBLAS, cudaFFT
- CUDAstreams for asynchronous execution