



Graphic Processors in Computational Applications

Part 2 – CUDA Advances

dr inż. Krzysztof Kaczmarek
2021



Materiały sponsorowane przez:

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informatycznych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego



Goals for today:

- ▶ Understand advanced CUDA techniques
- ▶ Get familiar with pitfalls of parallel programming

Part 2 – CUDA Advances

- Warp threads scheduling
- Advanced synchronization
- Variables and Memory
 - Memory types
 - Global Memory Access
 - Shared Memory
 - Example of shared memory utilization – matrices
- Asynchronous operations
- Problems of parallelism
 - Race conditions
 - Volatile
- Time Measurements

Conditional blocks: idle and active threads

Warp threads scheduling

```

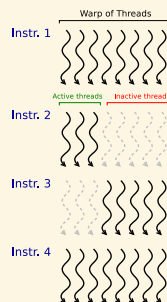
1 instruction 1
2 if (threadIdx.x<4)
3   instruction 2
4 else
5   instruction 3
6 instruction 4

```

Instr. for Threads 1-3: 1 2 (3) 4
Instr. for Threads 4-8: 1 (2) 3 4

Common mistake:
Instr 2. before Instr. 3

A single thread is assigned to a single ALU.
Waste of bandwidth – some ALUs do nothing.



Conditional blocks: warps scheduling

Warp threads scheduling

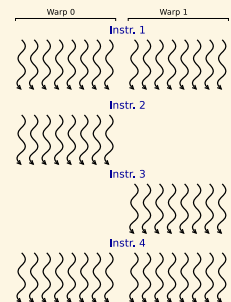
```

1 instruction 1
2 if (threadIdx.x<32)
3   instruction 2
4 else
5   instruction 3
6 instruction 4

```

Instr. for Warp 0: 1 2 4
Instr. for Warp 1: 1 3 4

Warp-level control saves bandwidth in conditional operations.



Part 2 – CUDA Advances

- Warp threads scheduling
- Advanced synchronization
- Variables and Memory
 - Memory types
 - Global Memory Access
 - Shared Memory
 - Example of shared memory utilization – matrices
- Asynchronous operations
- Problems of parallelism
 - Race conditions
 - Volatile
- Time Measurements

Advanced Threads Synchronization I

Advanced synchronization

Device side:

- ▶ `int __syncthreads_count(int predicate)`; is identical to `__syncthreads()` with the additional feature that it evaluates predicate for all threads of the block and returns the number of threads for which predicate evaluates to non-zero.
- ▶ `int __syncthreads_and(int predicate)`; similarly but evaluates predicate for all threads of the block and returns non-zero if and only if predicate evaluates to non-zero for all of them.
- ▶ `int __syncthreads_or(int predicate)`; ... similarly but returns non-zero if predicate evaluates to non-zero for any of the threads.
- ▶ `void __syncwarp(unsigned mask=0xffffffff)`; will cause the executing thread to wait until all warp lanes named in mask have executed a `__syncwarp()` (with the same mask) before resuming execution. All non-exited threads named in mask must execute a corresponding `__syncwarp()` with the same mask, or the result is undefined.

Advanced Threads Synchronization II

Advanced synchronization

Device side memory fence functions:

- ▶ `void __threadfence_block()`; waits until all global and shared memory accesses made by the calling thread before are visible to all threads in the thread block.
- ▶ `void __threadfence()`; waits until all global and shared memory accesses made by the calling thread prior to `__threadfence()` are visible to:
 - ▶ All threads in the thread block for shared memory accesses,
 - ▶ All threads in the device for global memory accesses.

9 / 60

Part 2 – CUDA Advances

Warp threads scheduling

Advanced synchronization

Variables and Memory

Memory types

Global Memory Access

Shared Memory

Example of shared memory utilization – matrices

Asynchronous operations

Problems of parallelism

Race conditions

Volatile

Time Measurements

10 / 60

Part 2 – CUDA Advances

Warp threads scheduling

Advanced synchronization

Variables and Memory

Memory types

Global Memory Access

Shared Memory

Example of shared memory utilization – matrices

Asynchronous operations

Problems of parallelism

Race conditions

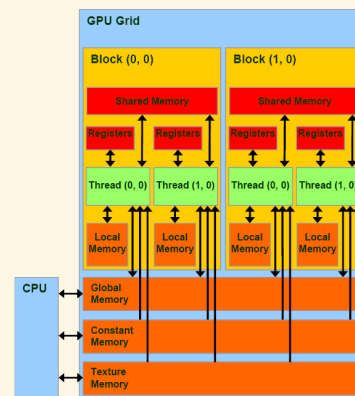
Volatile

Time Measurements

11 / 60

Accessing different types of memory

Variables and Memory



12 / 60

Variable Qualifiers (GPU side)

Variables and Memory

A variable declared in a kernel generally is stored in registers if possible. Exceptions (memory space specifiers):

- ▶ `__device__`
 - ▶ Stored in device global memory (large, high latency)
 - ▶ Accessible by all threads
 - ▶ Lifetime: application
- ▶ `__constant__`
 - ▶ Stored in constant memory space
 - ▶ Accessible by all threads
 - ▶ Lifetime: the CUDA context in which it is created
- ▶ `__shared__`
 - ▶ Stored in on-chip shared memory (very low latency)
 - ▶ Allocated by execution configuration or declared at compile time
 - ▶ Accessible by all threads in the same thread block
 - ▶ Lifetime: kernel execution
- ▶ `__managed__`
 - ▶ Can be referenced by both device and host
 - ▶ Lifetime: application

13 / 60

Part 2 – CUDA Advances

Warp threads scheduling

Advanced synchronization

Variables and Memory

Memory types

Global Memory Access

Shared Memory

Example of shared memory utilization – matrices

Asynchronous operations

Problems of parallelism

Race conditions

Volatile

Time Measurements

14 / 60

Global Memory Operations

Variables and Memory

- ▶ Memory operations are executed per warp
 - ▶ 32 threads in a warp provide memory addresses
 - ▶ Hardware determines into which lines those addresses fall
- ▶ Stores:
 - ▶ Invalidate L1, go at least to L2, 32-byte granularity
- ▶ Three types of loads:
 - ▶ Caching (default)
 - ▶ Non-caching
 - ▶ Read-only

15 / 60

Memory Load

Variables and Memory

- ▶ Caching (default mode)
 - ▶ Attempts to hit in L1, then L2, then GMEM
 - ▶ Load granularity is 128-byte line
- ▶ Non-caching
 - ▶ Compile with `-Xptxas -dlcm=cg` option to `nvcc`
 - ▶ Attempts to hit in L2, then GMEM (Does not hit in L1, invalidates the line if it's in L1 already)
 - ▶ Load granularity is 32 bytes
- ▶ Read-only
 - ▶ Loads via read-only cache: (Attempts to hit in Read-only cache, then L2, then GMEM)
 - ▶ Load granularity is 32 bytes

16 / 60

Coalesced Global Memory Access

Perhaps the most important optimization

Global memory loads and stores by threads of a warp are coalesced by the device into as few as possible transactions.

Compute capability ≥ 6.0 (since pascal)

The concurrent accesses of the threads of a warp will coalesce into a number of transactions equal to the number of 32-byte transactions necessary to service all of the threads of the warp.

Compute capability < 5.2 (before pascal)

L1-caching of accesses to global memory can be optionally enabled. If L1-caching is enabled on these devices, the number of required transactions is equal to the number of required 128-byte aligned segments.

17 / 60

Simple Access Pattern (cc ≥ 6.0)

Variables and Memory



- ▶ The k -th thread accesses the k -th word in a 32-byte aligned array.
- ▶ If the threads of a warp access adjacent 4-byte words
- ▶ ... and not all equally participate
- ▶ ... and/or random permuted access inside the block
- ▶ **then still only four 32-byte transactions would have been performed by a device.**

NVIDIA CUDA Toolkit. Cuda c++ best practices guide.

<https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>, 2020

18 / 60

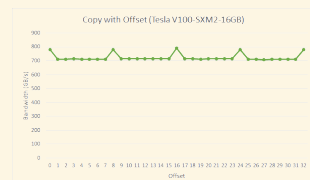
Misaligned Sequential Access Pattern (cc >6.0)

Variables and Memory



- ▶ The sequential threads accesses sequential memory but not aligned with a 32-byte segment,
- ▶ **then five 32-byte transactions would have been performed by a device.**

```
1 __global__ void offsetCopy(float *odata,
2 float *idata,
3 int offset)
4 {
5     int xid = blockIdx.x * blockDim.x +
6         threadIdx.x + offset;
7     odata[xid] = idata[xid];
8 }
```



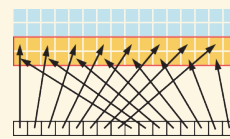
NVIDIA CUDA Toolkit. Cuda c++ best practices guide.

<https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>, 2020

19 / 60

Strided Access Pattern

Variables and Memory



- ▶ A stride of 2 results in a 50% of load/store efficiency since half the elements in the transaction are not used and represent wasted bandwidth.
- ▶ As the stride increases, the effective bandwidth decreases until the point where 32 32-byte segments are loaded for the 32 threads in a warp.

```
1 __global__ void strideCopy(float *odata,
2 float *idata,
3 int stride)
4 {
5     int xid = (blockIdx.x * blockDim.x +
6         threadIdx.x) * stride;
7     odata[xid] = idata[xid];
8 }
```



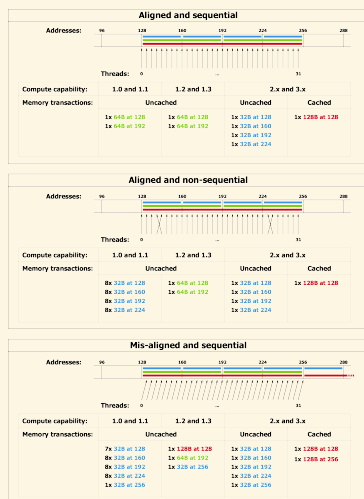
- ▶ Avoid non-unit-stride global memory accesses – **use shared memory.**

NVIDIA CUDA Toolkit. Cuda c++ best practices guide.

<https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>, 2020

20 / 60

In older architectures (not supported now)



21 / 60

Coalescing: Guidelines I

Variables and Memory

- ▶ Align data to fit equal segments in memory (arrays allocated with cudaMalloc... are positioned to appropriate addresses automatically)
- ▶ For single-dimensional arrays
 - ▶ array of type* accessed by BaseAddress + tid
 - ▶ type* must meet the size and alignment requirements
 - ▶ if size of type* is larger than 16 it must be treated with additional care
- ▶ For two-dimensional arrays
 - ▶ array of type* accessed by BaseAddress + width*tiy + tix
 - ▶ width is a multiply of 16
 - ▶ The width of the thread block is a multiple of half the warp size

22 / 60

Coalescing: Guidelines II

Variables and Memory

- ▶ If proper memory alignment is impossible:
 - ▶ Use structures of arrays instead of arrays of structures

AoS	x_1	y_1	z_1	w_1	x_2	y_2	z_2	w_2
	x_3	y_3	z_3	w_3	x_4	y_4	z_4	...
SoA	x_1	x_2	x_3	x_4
	y_1	y_2	y_3	y_4
	z_1	z_2	z_3	z_4
	w_1	w_2	w_3	w_4

- ▶ Use `__align(4)`, `__align(8)` or `__align(16)` in structure declarations

23 / 60

Coalescing example I

Variables and Memory

Misaligned memory access with float3 data

```
1 __global__ void accessFloat3(float3 *d_in, float3 *d_out)
2 {
3     int index = blockIdx.x * blockDim.x + threadIdx.x;
4     float3 a = d_in[index];
5     a.x += 2;
6     a.y += 2;
7     a.z += 2;
8     d_out[index] = a;
9 }
```

- ▶ Each thread reads 3 floats = 12 bytes
- ▶ Half warp reads $16 * 12 = 192$ bytes (three 64B non-contiguous segments)

NVIDIA. Cuda whitepapers. www.nvidia.com/cuda

24 / 60

Coalescing example II

Variables and Memory

Coalesced memory access with float3 data

```

1 __global__ void accessFloat3Shared(float *g_in, float *g_out)
2 {
3     int index = 3 * blockDim.x * blockIdx.x + threadIdx.x;
4     __shared__ float s_data[256*3];
5     s_data[threadIdx.x] = g_in[index];
6     s_data[threadIdx.x+256] = g_in[index+256];
7     s_data[threadIdx.x+512] = g_in[index+512];
8     __syncthreads();
9     float3 a = ((float3*)s_data)[threadIdx.x];
10    a.x += 2;
11    a.y += 2;
12    a.z += 2;
13    ((float3*)s_data)[threadIdx.x] = a;
14    __syncthreads();
15    g_out[index] = s_data[threadIdx.x];
16    g_out[index+256] = s_data[threadIdx.x+256];
17    g_out[index+512] = s_data[threadIdx.x+512];
18 }

```

NVIDIA. Cuda whitepapers. www.nvidia.com/cuda

25 / 60

Part 2 – CUDA Advances

Warp threads scheduling

Advanced synchronization

Variables and Memory

Memory types

Global Memory Access

Shared Memory

Example of shared memory utilization – matrices

Asynchronous operations

Problems of parallelism

Race conditions

Volatile

Time Measurements

26 / 60

Allocating shared memory

Variables and Memory

Static way

Device side:

```

1 __constant__ uint blockSize = 64;
2 __global__ void kernel(...)
3 {
4     ...
5     __shared__ short array0[blockSize];
6     __shared__ float array1[blockSize];
7     __shared__ int array2[blockSize];
8     ...
9 }

```

Host side:

```

1 kernel<<< nBlocks, blockSize >>>(...);

```

Dynamic way

Device side:

```

1 __constant__ uint blockSize = 64;
2 __global__ void kernel(...)
3 {
4     extern __shared__ float array[];
5     //All variables declared in this fashion,
6     //start at the same address in memory, so:
7     short* array0 = (short*)array;
8     float* array1 = (float*)array0[blockSize];
9     int* array2 = (int*)array1[blockSize];
10 }

```

Host side:

```

1 smBytes = blockSize*sizeof(float)
2 + blockSize*sizeof(short)
3 + blockSize*sizeof(int);
4 kernel<<< nBlocks, blockSize, smBytes >>>(...);

```

Note that pointers need to be aligned to the type they point to.
Error: array1 is not aligned to 4 bytes:

```

1 short* array0 = (short*)array;
2 float* array1 = (float*)array0[127];

```

27 / 60

Organization of shared memory

Variables and Memory

- ▶ Shared memory is divided into equally sized memory modules, called **banks**.
- ▶ Different banks can be accessed simultaneously.
- ▶ Read or write to n addresses in n banks multiplies bandwidth of a single bank by n .
- ▶ If many threads refers the same bank the access is serialized – hardware splits a memory request that has bank conflicts into as many separate conflict-free requests as necessary.
- ▶ There is one exception if all threads within a half-warp accesses the same address.

28 / 60

Bank conflicts

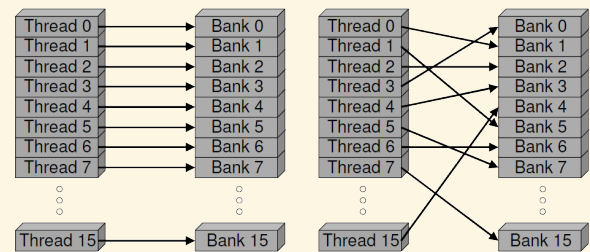
Variables and Memory

Shared memory banks are organized in such a way that successive 32-bit words are assigned to successive banks and each bank has a bandwidth of 32 bits per clock cycle. The bandwidth of shared memory is 32 bits per bank per clock cycle.

29 / 60

Access with no bank conflicts

Variables and Memory



left: stride = 1

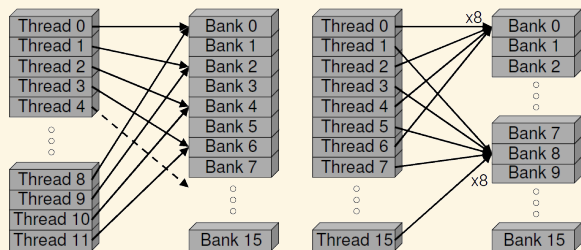
right: stride random

NVIDIA. Cuda whitepapers. www.nvidia.com/cuda

30 / 60

Access with bank conflicts

Variables and Memory



left: stride = 2 (2 way bank conflict)

right: stride = 8 (8 way bank conflict)

NVIDIA. Cuda whitepapers. www.nvidia.com/cuda

- ▶ Padding – adding extra space between array elements in order to brake cyclic access to same bank.

31 / 60

Example of bank conflicts removal in reduction I

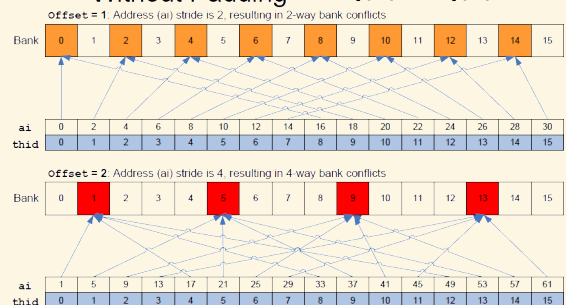
Variables and Memory

Addressing Without Padding

```

int ai = offset*(2*thid+1)-1;
int bi = offset*(2*thid+2)-1;
temp[bi] += temp[ai];

```



Mark Harris. Parallel prefix sum (scan) with CUDA. www.nvidia.com/cuda, 2007

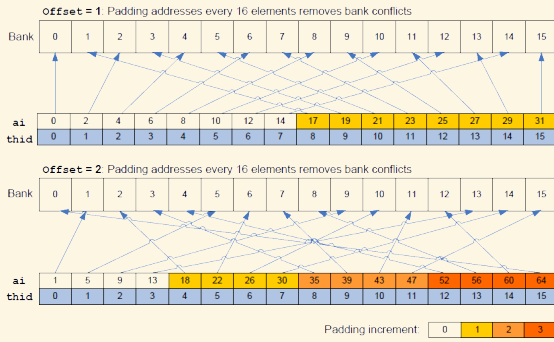
32 / 60

Example of bank conflicts removal in reduction II

Variables and Memory

Addressing With Padding

```
int ai = offset*(2*thid+1)-1;
int bi = offset*(2*thid+2)-1;
ai += ai / NUM_BANKS;
bi += bi / NUM_BANKS;
temp[bi] += temp[ai];
```



Mark Harris. Parallel prefix sum (scan) with CUDA. www.nvidia.com/cuda, 2007

33 / 60

Padding implementation I

Variables and Memory

We need more space in shared memory:

```
1 unsigned int extra_space = num_elements / NUM_BANKS;

Padding macro:

1 #define NUM_BANKS 16
2 #define LOG_NUM_BANKS 4
3
4 #ifndef ZERO_BANK_CONFLICTS
5 #define CONFLICT_FREE_OFFSET(index) ((index) >> LOG_NUM_BANKS \
6 + (index) >> (2 * LOG_NUM_BANKS))
7 #else
8 #define CONFLICT_FREE_OFFSET(index) ((index) >> LOG_NUM_BANKS)
9 #endif

Zero bank conflicts requires even more additional space:

1 #ifndef ZERO_BANK_CONFLICTS
2 extra_space += extra_space / NUM_BANKS;
3 #endif
```

Mark Harris. Parallel prefix sum (scan) with CUDA. www.nvidia.com/cuda, 2007

34 / 60

Padding implementation II

Variables and Memory

Loading data into shared memory:

```
1 int ai = thid, bi = thid + (n/2);
2
3 // compute spacing to avoid bank conflicts
4 int bankOffsetA = CONFLICT_FREE_OFFSET(ai);
5 int bankOffsetB = CONFLICT_FREE_OFFSET(bi);
6
7 TEMP(ai + bankOffsetA) = g_idata[ai];
8 TEMP(bi + bankOffsetB) = g_idata[bi];
```

Algorithm:

```
1 int ai = offset*(2*thid+1)-1;
2 int bi = offset*(2*thid+2)-1;
3
4 ai += CONFLICT_FREE_OFFSET(ai);
5 bi += CONFLICT_FREE_OFFSET(bi);
6
7 TEMP(bi) += TEMP(ai);
```

Mark Harris. Parallel prefix sum (scan) with CUDA. www.nvidia.com/cuda, 2007

35 / 60



Part 2 – CUDA Advances

Warp threads scheduling

Advanced synchronization

Variables and Memory

Memory types

Global Memory Access

Shared Memory

Example of shared memory utilization – matrices

Asynchronous operations

Problems of parallelism

Race conditions

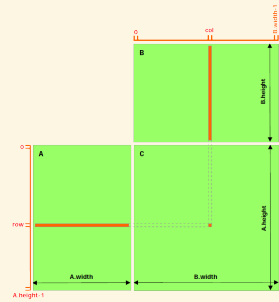
Volatile

Time Measurements

36 / 60

Matrix Multiplication

No shared memory used



```
1 // Matrix multiplication kernel called by MatMul()
2 __global__ void MatMulKernel(Matrix A,
3 Matrix B,
4 Matrix C)
5 {
6 // Each thread computes one element of C
7 // by accumulating results into Cvalue
8 float Cvalue = 0;
9 int row = blockIdx.y * blockDim.y + threadIdx.y;
10 int col = blockIdx.x * blockDim.x + threadIdx.x;
11 for (int e = 0; e < A.width; ++e)
12 Cvalue += A.elements[row * A.width + e]
13 * B.elements[e * B.width + col];
14 C.elements[row * C.width + col] = Cvalue;
15 }
```

37 / 60

Matrix Multiplication

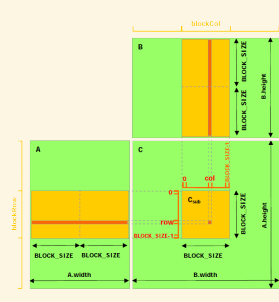
Host program for clarity

```
1 void MatMul(const Matrix A, 22 // Allocate C in device memory
2 const Matrix B, 23 Matrix d_C;
3 Matrix C) 24 d_C.width = C.width; d_C.height = C.height;
4 { 25 size = C.width * C.height * sizeof(float);
5 // Load A and B to device memory 26 cudaMalloc(&d_C.elements, size);
6 Matrix d_A; 27 // Invoke kernel
7 d_A.width = A.width; d_A.height = A.height; 28 dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
8 size_t size = A.width * A.height 29 dim3 dimGrid(B.width / dimBlock.x,
9 * sizeof(float); 30 A.height / dimBlock.y);
10 31 MatMulKernel<<<dimGrid, dimBlock>>
11 cudaMalloc(&d_A.elements, size); 32 (d_A, d_B, d_C);
12 cudaMemcpy(d_A.elements, A.elements, size, 33 cudaMemcpyHostToDevice);
13 cudaMemcpyHostToDevice); 34
14 Matrix d_B; 35 // Read C from device memory
15 d_B.width = B.width; 36 cudaMemcpy(C.elements, d_C.elements, size,
16 d_B.height = B.height; 37 cudaMemcpyDeviceToHost);
17 size = B.width * B.height * sizeof(float); 38
18 39 // Free device memory
19 cudaMalloc(&d_B.elements, size); 40 cudaFree(d_A.elements);
20 cudaMemcpy(d_B.elements, B.elements, size, 41 cudaFree(d_B.elements);
21 cudaMemcpyHostToDevice); 42 cudaFree(d_C.elements);
43 }
```

38 / 60

Matrix Multiplication with shared memory used I

with shared memory used I



```
1 typedef struct {
2 int width;
3 int height;
4 int stride;
5 float* elements;
6 } Matrix;
7
8 __device__ float GetElement(const Matrix A,
9 int row, int col)
10 {
11 return A.elements[row * A.stride + col];
12 }
13
14 __device__ void SetElement(Matrix A, int row,
15 int col, float value)
16 {
17 A.elements[row * A.stride + col] = value;
18 }
19
20 // Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub
21 // of A located col sub-matrices to the right and
22 // row sub-matrices down from the upper-left corner
23 // of A
24 __device__ Matrix GetSubMatrix(Matrix A, int row,
25 int col)
26 {
27 Matrix Asub;
28 Asub.width = BLOCK_SIZE;
29 Asub.height = BLOCK_SIZE;
30 Asub.stride = A.stride;
31 Asub.elements = &A.elements[A.stride
32 * BLOCK_SIZE * row
33 + BLOCK_SIZE * col];
34 }
```

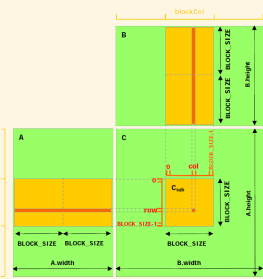
NVIDIA. Cuda c++ programming guide.

www.nvidia.com/cuda

39 / 60

Matrix Multiplication with shared memory used II

with shared memory used II



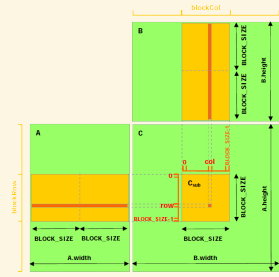
```
1 __global__ void MatMulKernel(Matrix A,
2 Matrix B,
3 Matrix C)
4 {
5 // Block row and column
6 int blockRow = blockIdx.y;
7 int blockCol = blockIdx.x;
8
9 // Each thread computes one sub-matrix
10 // Csub of C
11 Matrix Csub = GetSubMatrix(C, blockRow,
12 blockCol);
13
14 // Each thread computes one element of Csub
15 // by accumulating results into Cvalue
16 float Cvalue = 0;
17
18 // Thread row and column within Csub
19 int row = threadIdx.y;
20 int col = threadIdx.x;
```

NVIDIA. Cuda c++ programming guide.

www.nvidia.com/cuda

40 / 60

Matrix Multiplication with shared memory used III



NVIDIA. Cuda c++ programming guide.
www.nvidia.com/cuda

```

51 for (int m = 0; m < (A.width / BLOCK_SIZE); ++m)
52 {
53     // Get sub-matrix Asub of A
54     Matrix Asub = GetSubMatrix(A, blockRow, m);
55
56     // Get sub-matrix Bsub of B
57     Matrix Bsub = GetSubMatrix(B, m, blockCol);
58
59     // Shared memory used to store Asub and Bsub
60     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
61     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
62
63     // Each thread loads one element of each sub-
64     // matrix
65     As[row][col] = GetElement(Asub, row, col);
66     Bs[row][col] = GetElement(Bsub, row, col);
67
68     // Synchronize to make sure the sub-matrices
69     // are loaded before starting the
70     // computation
71     __syncthreads();
72
73     // Multiply Asub and Bsub together
74     for (int e = 0; e < BLOCK_SIZE; ++e)
75         Cvalue += As[row][e] * Bs[e][col];
76
77     // Synchronize to assure that the preceding
78     // computation is done before loading new
79     // sub-matrices of A and B
80     __syncthreads();
81 }
82 // Write Csub to device memory, one thread, one
83 // element
84 SetElement(Csub, row, col, Cvalue);
85 }

```

Part 2 – CUDA Advances

- Warp threads scheduling
- Advanced synchronization
- Variables and Memory
 - Memory types
 - Global Memory Access
 - Shared Memory
 - Example of shared memory utilization – matrices

Asynchronous operations

- Problems of parallelism
 - Race conditions
 - Volatile

Time Measurements

Streams API

Asynchronous operations

- Applications manage concurrency through streams.
- A stream is a sequence of commands that execute in order.
- Different streams may execute their commands out of order with respect to one another or concurrently.
- cudaStream_t – stream type
- cudaStreamCreate(&stream)
- cudaStreamDestroy(&stream) – waits for all tasks to complete before destroying a stream;
- cudaStreamQuery() – checks if all preceding commands in a stream have completed
- cudaStreamSynchronize() – forces the run-time to wait until all preceding commands in a stream have completed.
- cudaThreadSynchronize() – forces the run-time to wait until all preceding device tasks in all streams have completed

Streams API – example

Asynchronous operations

NVIDIA. Cuda c++ programming guide. www.nvidia.com/cuda

```

1 cudaStream_t stream[2];
2 for (int i = 0; i < 2; ++i)
3     cudaStreamCreate(&stream[i]);
4 float* hostPtr;
5 cudaMallocHost((void**)&hostPtr, 2 * size, cudaHostAllocDefault);
6 cudaMalloc((void**)&inputDevPtr, 2 * size);
7 cudaMalloc((void**)&outputDevPtr, 2 * size);
8 for (int i = 0; i < 2; ++i)
9     cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
10                    size, cudaMemcpyHostToDevice, stream[i]);
11 for (int i = 0; i < 2; ++i)
12     myKernel<<<100, 512, 0, stream[i]>>>
13     (outputDevPtr + i * size, inputDevPtr + i * size, size);
14 for (int i = 0; i < 2; ++i)
15     cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
16                    size, cudaMemcpyDeviceToHost, stream[i]);
17 cudaThreadSynchronize();
18 for (int i = 0; i < 2; ++i)
19     cudaStreamDestroy(&stream[i]);

```

Part 2 – CUDA Advances

- Warp threads scheduling
- Advanced synchronization
- Variables and Memory
 - Memory types
 - Global Memory Access
 - Shared Memory
 - Example of shared memory utilization – matrices

Asynchronous operations

- Problems of parallelism
 - Race conditions
 - Volatile

Time Measurements

Part 2 – CUDA Advances

- Warp threads scheduling
- Advanced synchronization
- Variables and Memory
 - Memory types
 - Global Memory Access
 - Shared Memory
 - Example of shared memory utilization – matrices

Asynchronous operations

- Problems of parallelism
 - Race conditions
 - Volatile

Time Measurements

Race conditions

Problems of parallelism

Simplest possible operation (*x is a global memory pointer)

Let int *x point to global memory. *x++ happens in 3 steps:

- Read the value in *x into a register.
- Add 1 to the value in the register.
- Write the result back to *x.

```

1 A: *x++
2 B: *x++

```

```

1 A: a = *x //a=7
2 B: b = *x //b=7
3 A: a++ //8
4 A: *x = a //8
5 B: b++ //8
6 B: *x = b //8

```

Race conditions

Problems of parallelism

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <cuda.h>
4 #include <cuda_runtime.h>
5
6 __global__ void colone1(int *d_a){
7     *d_a += 1;
8 }
9
10 int main(){
11     int a = 0, *d_a;
12     cudaMalloc((void**)&a_d, sizeof(int));
13     cudaMemcpy(d_a, &a, sizeof(int), cudaMemcpyHostToDevice);
14     float elapsedTime;
15     cudaEvent_t start, stop;
16     cudaEventCreate(&start);
17     cudaEventCreate(&stop);
18     cudaEventRecord( start, 0 );
19
20     colone1<<<1000,1000>>>(d_a);
21
22     cudaEventRecord( stop, 0 );
23     cudaEventSynchronize( stop );
24     cudaEventElapsedTime( &elapsedTime, start, stop );
25     cudaEventDestroy( start );
26     cudaEventDestroy( stop );
27     printf("GPU_Time: %f\n", elapsedTime);
28
29     cudaMemcpy(&a, d_a, sizeof(int), cudaMemcpyDeviceToHost);
30
31     printf("a=%d\n", a);
32     cudaFree(d_a);
33 }

```

Race condition results

Problems of parallelism

Output:

```
1 > nvcc race_condition.cu -o race_condition
2 > ./race_condition
3 GPU Time : 0.148 ms
4 a = 88
```

Modification:

```
1 __global__ void colonel(int *d_a){
2   atomicAdd(d_a, 1);
3 }
```

Output:

```
1 GPU Time : 14.85 ms
2 a = 1000000
```

Atomic functions can only be used in device functions.

49 / 60

Atomic operations I

(for all devices CC>2.0)

Device-wide atomics: atomic for all CUDA threads in the current program executing in the same compute device as the current thread:

```
1 atomicAdd()
2 atomicSub()
3 atomicMin()
4 atomicMax()
5 atomicInc()
6 atomicDec()
7 atomicAdd()
8 atomicExch()
9 atomicAnd()
10 atomicOr()
11 atomicXor()
12 int atomicCAS(int* address, int compare, int val); // Compare And Swap (returns old value)
```

An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory.

50 / 60

Atomic operations II

(for all devices CC≥6.0)

System-wide atomics: atomic for all threads in the current program including other CPUs and GPUs in the system. These are suffixed with `_system`. Like: `atomicAdd_system()`.

Block-wide atomics: atomic for all CUDA threads in the current program executing in the same thread block as the current thread. These are suffixed with `_block`. Like: `atomicAdd_block()`.

51 / 60



Part 2 – CUDA Advances

Warp threads scheduling

Advanced synchronization

Variables and Memory

Memory types

Global Memory Access

Shared Memory

Example of shared memory utilization – matrices

Asynchronous operations

Problems of parallelism

Race conditions

Volatile

Time Measurements

52 / 60

Volatile variables and specific compiler optimizations

Problems of parallelism

- ▶ One of the compiler's tricks: reuse references to memory location
- ▶ Result: A reused value may be changed by another thread in the background

```
1 // myArray is an array of non-zero integers
2 // located in global or shared memory
3 __global__ void myKernel(int* result)
4 {
5   int tid = threadIdx.x;
6   int ref1 = myArray[tid] * 1;
7   myArray[tid + 1] = 2;
8   int ref2 = myArray[tid] * 1;
9   result[tid] = ref1 * ref2;
10 }
```

- ▶ the first reference to `myArray[tid]` compiles into a memory read instruction
- ▶ the second reference does not as the compiler simply reuses the result of the first read

53 / 60



Part 2 – CUDA Advances

Warp threads scheduling

Advanced synchronization

Variables and Memory

Memory types

Global Memory Access

Shared Memory

Example of shared memory utilization – matrices

Asynchronous operations

Problems of parallelism

Race conditions

Volatile

Time Measurements

54 / 60

Timers API

Time Measurements

- ▶ `cudaEvent_t` – event type
- ▶ `cudaEventSynchronize()` – blocks CPU until given event records
- ▶ `cudaEventRecord()` – records given event in given stream
- ▶ `cudaEventElapsedTime()` – calculates time in milliseconds between events
- ▶ `cudaEventCreate()` – creates an event
- ▶ `cudaEventDestroy()` – destroys an event

55 / 60

Timers Example

Time Measurements

```
1 cudaEvent_t start, stop; float time;
2 cudaEventCreate(&start);
3 cudaEventCreate(&stop);
4
5 cudaEventRecord( start, 0 );
6
7 kernel<<<grid,threads>>> ( d_odata, d_idata, size_x, size_y);
8
9 cudaEventRecord( stop, 0 );
10
11 cudaEventSynchronize( stop );
12
13 cudaEventElapsedTime( &time, start, stop );
14
15 cudaEventDestroy( start );
16 cudaEventDestroy( stop );
```

56 / 60

Theoretical Bandwidth Calculation

Time Measurements

$$TB = (\text{Clock} \times 10^6 \times \text{MemInt} \times 2) / 10^9$$

- ▶ TB – theoretical bandwidth [GB/s]
- ▶ Clock – memory clock rate [MHz]
- ▶ MemInt – width of memory interface [B]
- ▶ 2 – DDR – Double Data Rate Memory

For NVIDIA GeForce GTX 280 we get:

$$(1107 \times 10^6 \times 512 / 8 \times 2) / 10^9 = 141.6 \text{ [GB/s]}$$

57 / 60

Effective Bandwidth Calculation





Time Measurements

$$EB = \frac{(B_r + B_w) \times 10^{-9}}{t}$$

- ▶ EB – effective bandwidth [GB/s]
- ▶ B_r – bytes read
- ▶ B_w – bytes written
- ▶ t – time of the test [s]

58 / 60

Bibliography

-  Mark Harris. Parallel prefix sum (scan) with CUDA. www.nvidia.com/cuda, 2007.
-  NVIDIA. Cuda c++ programming guide. www.nvidia.com/cuda.
-  NVIDIA. Cuda whitepapers. www.nvidia.com/cuda.
-  NVIDIA CUDA Toolkit. Cuda c++ best practices guide. <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>, 2020.

59 / 60

Materiały sponsorowane przez:

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Politechnika
Warszawska

Unia Europejska
Europejski Fundusz Społeczny



60 / 60