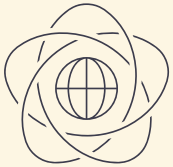


Faculty of Mathematics and Information Science

WARSAW UNIVERSITY OF TECHNOLOGY

Graphic Processors in Computational Applications

Part 4 – Extended CUDA Features

dr inż. Krzysztof Kaczmarski

2021

Fundusze Europejskie
Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Politechnika Warszawska

Unia Europejska
Europejski Fundusz Społeczny

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

Part 4 – Extended CUDA Features

Warp vote functions

Advanced Warp-level Functions

compute capability 7.x or higher

`__all_sync(unsigned mask, predicate)`: Evaluate predicate for all non-exited threads in mask and return non-zero if and only if predicate evaluates to non-zero for all of them.

Warp vote functions

Advanced Warp-level Functions

compute capability 7.x or higher

`__all_sync(unsigned mask, predicate)`: Evaluate predicate for all non-exited threads in mask and return non-zero if and only if predicate evaluates to non-zero for all of them.

`__any_sync(unsigned mask, predicate)`: Evaluate predicate for all non-exited threads in mask and return non-zero if and only if predicate evaluates to non-zero for any of them.

Warp vote functions

Advanced Warp-level Functions

compute capability 7.x or higher

`__all_sync(unsigned mask, predicate)`: Evaluate predicate for all non-exited threads in mask and return non-zero if and only if predicate evaluates to non-zero for all of them.

`__any_sync(unsigned mask, predicate)`: Evaluate predicate for all non-exited threads in mask and return non-zero if and only if predicate evaluates to non-zero for any of them.

`__ballot_sync(unsigned mask, predicate)`: Evaluate predicate for all non-exited threads in mask and return an integer whose Nth bit is set if and only if predicate evaluates to non-zero for the Nth thread of the warp and the Nth thread is active.

Warp vote functions

Advanced Warp-level Functions

compute capability 7.x or higher

`__all_sync(unsigned mask, predicate)`: Evaluate predicate for all non-exited threads in mask and return non-zero if and only if predicate evaluates to non-zero for all of them.

`__any_sync(unsigned mask, predicate)`: Evaluate predicate for all non-exited threads in mask and return non-zero if and only if predicate evaluates to non-zero for any of them.

`__ballot_sync(unsigned mask, predicate)`: Evaluate predicate for all non-exited threads in mask and return an integer whose Nth bit is set if and only if predicate evaluates to non-zero for the Nth thread of the warp and the Nth thread is active.

`__activemask()`: Returns a 32-bit integer mask of all currently active threads in the calling warp. The Nth bit is set if the Nth lane in the warp is active when it is called. Inactive threads are represented by 0 bits in the returned mask. Threads which have exited the program are always marked as inactive.

Warp match functions

Advanced Warp-level Functions

compute capability 7.x or higher

`__match_any_sync(unsigned mask, T value)`: Returns mask of threads that have same value of value in mask

`__match_all_sync(unsigned mask, T value, int *pred)`: Returns mask if all threads in mask have the same value for value; otherwise 0 is returned. Predicate pred is set to true if all threads in mask have the same value of value; otherwise the predicate is set to false.

T can be `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float`, `double`

Warp reduce functions

Advanced Warp-level Functions

compute capability 8.x

`T __reduce_*_sync(unsigned mask, T value)`: intrinsics perform a reduction operation on the data provided in value after synchronizing threads named in mask. `T` can be unsigned or signed for add, min, max and unsigned only for and, or, xor operations.

Warp shuffle functions

Advanced Warp-level Functions

compute capability 3.x or higher

`__shfl_sync`, `__shfl_*_sync`: exchange a variable between threads within a warp (up, down, xor).

```
1  \\ Direct copy from indexed lane
2  T __shfl_sync(unsigned mask, T var, int srcLane, int width=warpSize)
       ;
3  \\ Copy from a lane with lower ID relative to caller
4  T __shfl_up_sync(unsigned mask, T var, unsigned int delta, int width
       =warpSize);
5  \\ Copy from a lane with higher ID relative to caller
6  T __shfl_down_sync(unsigned mask, T var, unsigned int delta, int
       width=warpSize);
7  \\ Copy from a lane based on bitwise XOR of own lane ID
8  T __shfl_xor_sync(unsigned mask, T var, int laneMask, int width=
       warpSize);
```

Warp broadcast without shared memory

Advanced Warp-level Functions

```
1  __global__ void bcast(int arg) {
2      int laneId = threadIdx.x & 0x1f;
3      int value;
4      if (laneId == 0) // Note unused variable for
5          value = arg; // all threads except lane 0
6      value = __shfl_sync(0xffffffff, value, 0); // Synchronize all
           threads in warp, and get "value" from lane 0
7      if (value != arg)
8          printf("Thread␣%d␣failed.\n", threadIdx.x);
9  }
```

Inclusive scan across sub-partitions of 8 threads

Advanced Warp-level Functions

```
1  __global__ void scan4() {
2      int laneId = threadIdx.x & 0x1f;
3      // Seed sample starting value (inverse of lane ID)
4      int value = 31 - laneId;
5
6      // Loop to accumulate scan within my partition.
7      // Scan requires log2(n) == 3 steps for 8 threads
8      // It works by an accumulated sum up the warp
9      // by 1, 2, 4, 8 etc. steps.
10     for (int i=1; i<=4; i*=2) {
11         // We do the __shfl_sync unconditionally so that we
12         // can read even from threads which won't do a
13         // sum, and then conditionally assign the result.
14         int n = __shfl_up_sync(0xffffffff, value, i, 8);
15         if ((laneId & 7) >= i)
16             value += n;
17     }
18     printf("Thread %d final value = %d\n", threadIdx.x, value);
19 }
```

Reduction across a warp

Advanced Warp-level Functions

```
__global__ void warpReduce() {
    int laneId = threadIdx.x & 0x1f;
    // Seed starting value as inverse lane ID
    int value = 31 - laneId;

    // Use XOR mode to perform butterfly reduction
    for (int i=16; i>=1; i/=2)
        value += __shfl_xor_sync(0xffffffff, value, i, 32);

    // "value" now contains the sum across all threads
    printf("Thread␣%d␣final␣value␣=␣%d\n", threadIdx.x, value);
}
```

Warp matrix functions

Advanced Warp-level Functions

warp matrix operations leverage Tensor Cores to accelerate matrix problems of the form $D = A \cdot B + C$. These operations are supported on mixed-precision floating point data for devices of compute capability 7.0 or higher. This requires co-operation from all threads in a warp.

Warp matrix functions

Advanced Warp-level Functions

warp matrix operations leverage Tensor Cores to accelerate matrix problems of the form $D = A \cdot B + C$. These operations are supported on mixed-precision floating point data for devices of compute capability 7.0 or higher. This requires co-operation from all threads in a warp.
Sub-byte WMMA operations provide a way to access the low-precision capabilities of Tensor Cores. They are considered a preview feature i.e. the data structures and APIs for them are subject to change and may not be compatible with future releases.

Tensor cores matrix multiplication

Advanced Warp-level Functions

16x16x16 matrix multiplication in a single warp.

```
1  #include <mma.h>
2  using namespace nvcuda;
3  __global__ void wmma_ker(half *a, half *b, float *c) {
4     // Declare the fragments
5     wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major>
           a_frag;
6     wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major>
           b_frag;
7     wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;
8     // Initialize the output to zero
9     wmma::fill_fragment(c_frag, 0.0f);
10    // Load the inputs
11    wmma::load_matrix_sync(a_frag, a, 16);
12    wmma::load_matrix_sync(b_frag, b, 16);
13    // Perform the matrix multiplication
14    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
15    // Store the output
16    wmma::store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
17 }
```

Part 4 – Extended CUDA Features

Part 4 – Extended CUDA Features

Independent Thread Scheduling Compatibility

Programming Model Extensions

The Volta and Turing architectures feature Independent Thread Scheduling among threads in a warp. If the developer made assumptions about warp-synchronicity, 1 this feature can alter the set of threads participating in the executed code compared to previous architectures.

Independent Thread Scheduling Compatibility

Programming Model Extensions

- To avoid data corruption, applications using warp intrinsics (`__shfl*`, `__any`, `__all`, and `__ballot`) should transition to the new, safe, synchronizing counterparts, with the `*_sync` suffix. The new warp intrinsics take in a mask of threads that explicitly define which lanes (threads of a warp) must participate in the warp intrinsic.
- Applications that assume reads and writes are implicitly visible to other threads in the same warp need to insert the new `__syncwarp()` warp-wide barrier synchronization instruction between steps where data is exchanged between threads via global or shared memory. Assumptions that code is executed in lockstep or that reads/writes from separate threads are visible across a warp without synchronization are invalid.
- Applications using `__syncthreads()` or the PTX bar.sync (and their derivatives) in such a way that a barrier will not be reached by some non-exited thread in the thread block must be modified to ensure that all non-exited threads reach the barrier.

Part 4 – Extended CUDA Features

Cooperative Groups

Programming Model Extensions

Cooperative Groups is an extension to the CUDA programming model, introduced in CUDA 9, for organizing groups of communicating threads. Cooperative Groups allows developers to express the granularity at which threads are communicating, helping them to express richer, more efficient parallel decompositions.

Part 4 – Extended CUDA Features

Part 4 – Extended CUDA Features

Advanced Warp-level Functions

Programming Model Extensions
- Independent Thread Scheduling Compatibility
- Cooperative Groups

CUDA 11 and Ampere Architecture
- Compute Sanitizer

A new tool to check memory accesses

CUDA 11 and Ampere Architecture

pre CUDA 11

`cuda-memcheck` tool

from CUDA 11

Compute Sanitizer, a next-generation, functional correctness checking tool that provides runtime checking for out-of-bounds memory accesses and race condition

Compute Sanitizer I

Out-of-bounds array access

```
1  __global__ void oobAccess(int* in, int* out)
2  {
3      int bid = blockIdx.x;
4      int tid = threadIdx.x;
5      if (bid == 4)
6          out[tid] = in[dMem[tid]];
7  }
8
9  int main()
10 {
11     ...
12     // Array of 8 elements, where element 4 causes the OOB
13     std::array<int, Size> hMem = {0, 1, 2, 10, 4, 5, 6, 7};
14     cudaMemcpy(d_mem, hMem.data(), size, cudaMemcpyHostToDevice);
15
16     oobAccess<<<10, Size>>>(d_in, d_out);
17     cudaDeviceSynchronize();
18     ...
19
20 $ /usr/local/cuda-11.0/Sanitizer/compute-sanitizer --destroy-on-device-error kernel --show-backtrace
       no basic
21 ========= COMPUTE-SANITIZER
22 Device: Tesla T4
23 ========= Invalid __global__ read of size 4 bytes
24 ========= at 0x480 in /tmp/CUDA11.0/ComputeSanitizer/Tests/Memcheck/basic/basic.cu:40:oobAccess(int*,
       int*)
25 ========= by thread (3,0,0) in block (4,0,0)
26 ========= Address 0x7f551f200028 is out of bounds
```

Compute Sanitizer II

Race condition

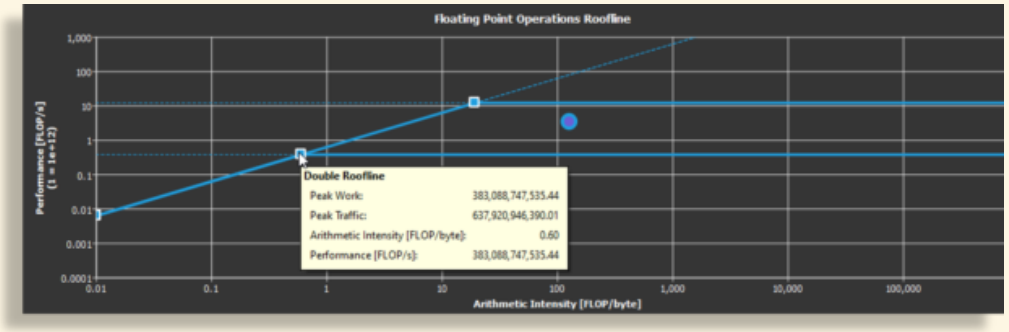
```
1   __global__ void Basic()
2   {
3       __shared__ volatile int i;
4       i = threadIdx.x;
5   }
6
7   int main()
8   {
9       Basic<<<1,2>>>();
10      cudaDeviceSynchronize();
11      ...
12
13
14  $ /usr/local/cuda-11.0/Sanitizer/compute-sanitizer --destroy-on-device-error kernel --show-backtrace
        no --tool racecheck --racecheck-report hazard raceBasic
15  ========= COMPUTE-SANITIZER
16  ========= ERROR: Potential WAW hazard detected at __shared__ 0x0 in block (0,0,0) :
17  ========= Write Thread (0,0,0) at 0x100 in /tmp/CUDA11.0/ComputeSanitizer/Tests/Racecheck/raceBasic/
        raceBasic.cu:11:Basic(void)
18  ========= Write Thread (1,0,0) at 0x100 in /tmp/CUDA11.0/ComputeSanitizer/Tests/Racecheck/raceBasic/
        raceBasic.cu:11:Basic(void)
19  ========= Current Value : 0, Incoming Value : 1
20  =========
21  ========= RACECHECK SUMMARY: 1 hazard displayed (1 error, 0 warnings)
```

Roofline model visualization

CUDA 11 and Ampere Architecture

Arithmetic Intensity is the most important concept in Roofline.

- Ratio of Total FLOPs performed to Total Bytes moved
- Total Bytes to/from DRAM and includes all cache and prefetcher effects
- Can be very different from total loads/stores (bytes requested) due to cache reuse



Pramod Ramarao. Cuda 11 features revealed. https://developer.nvidia.com/blog/cuda-11-features-revealed/, May 2020

L2 persistence cache access

CUDA 11 and Ampere Architecture

When a CUDA kernel accesses a data region in the global memory repeatedly, such data accesses can be considered to be persisting. On the other hand, if the data is only accessed once, such data accesses can be considered to be streaming.

Starting with CUDA 11.0, devices of compute capability 8.0 and above have the capability to influence persistence of data in the L2 cache, potentially providing higher bandwidth and lower latency accesses to global memory.

For details please consult:

NVIDIA. Cuda c++ programming guide. 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

Bibliography


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


Pramod Ramarao. Cuda 11 features revealed.
https://developer.nvidia.com/blog/cuda-11-features-revealed/,
May 2020.


NVIDIA CUDA Toolkit. Cuda c++ best practices guide.
https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html,
2020.

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