center for
excellence in parallel
programming

CUDA

Georges-Emmanuel Moulard
Paul Karlshöfer

Atos

CUDA ADVANCED

05/08/2020

Debugging in CUDA

05/08/2020

Recap

- Each API call returns a **`cudaError_t`** code.
- So we wrap each synchronous CUDA call with our CUDA_SAFE_CALL ( ) marco.

- Kernels do not have a return value, if case of failure, the next synchronized call will return the error or any asynchronous call which was executed previously.

- But, many runtime errors are not detectable this way.

cuda-gdb

- CUDA command line debugger
- Compile application with "`-g -G`" option.
- Most **`gdb`** commands are available
- Features: Setup of breakpoints, Inspecting program state (value of variables), State of cuda threads, State of a particular SM, launch traces of kernel etc.
- Detailed information: https://docs.nvidia.com/cuda/cuda-gdb/index.html

cuda-memcheck

- Analyzes code for specific problems

memcheck	checks for memory access errors and leaks
racecheck	checks for shared memory data access hazards
initcheck	checks for uninitialized global memory
synccheck	checks for thread synchronization hazards

- Set via `cuda-memcheck --tool <memcheck|racecheck|synccheck|initcheck>`

- Detailed description: https://docs.nvidia.com/cuda/cuda-memcheck/index.html

TP: Explore the memcheck tool

- /TP_CUDA_C/ADVANCED/DEBUG_EXAMPLE/

- Consider the bad_sample.cu
- Task:
  1. Compile the program (with –g –G)
  2. Run the program (Notice that the program exists normally) No errors detected by CUDA_SAFE_CALL
  3. Use **cuda-memcheck** to find the errors

```
cuda-memcheck --tool <memcheck|racecheck|synccheck|initcheck> ./app
```

KERNEL OPTIMIZATION

05/08/2020

Thread ID & Warps

05/08/2020

Thread ID

- The ID of a thread within a block is always:

```
tid = threadIdx.x + threadIdx.y*(blockDim.x)
     +threadIdx.z *(blockDim.x * blockDim.y)
```

  - threadIdx.{x,y,z}: index of the thread in its x, y, z dimension
  - blockDim.{x,y,z}: size of the block in its x, y, z dimension

- Example:
  - Considering a block with: *blockDim.x = 8, blockDim.y = 6, blockDim.z = 4*
  - the thread with: *threadIdx.x = 1, threadIdx.y = 2, threadIdx.z = 3*
  - has the ID: 1 + 2*(8) + 3*(6*8) = 161

WARP

- The warp is the base unit of execution on the GPU multiprocessor:
  - warps are subdivision of a block of threads
  - streaming multiprocessors have warp schedulers

- All aspects of kernel performance are driven by warps:
  - memory access
    - global, constant and shared
  - branch issues
  - instruction level parallelism (ILP)
  - occupancy

- **A warp is composed of 32 threads of consecutive increasing ID**
  - the first warp starts with the thread of ID 0
  - a warp scheduler launches 32 instructions by cycle (one by thread)

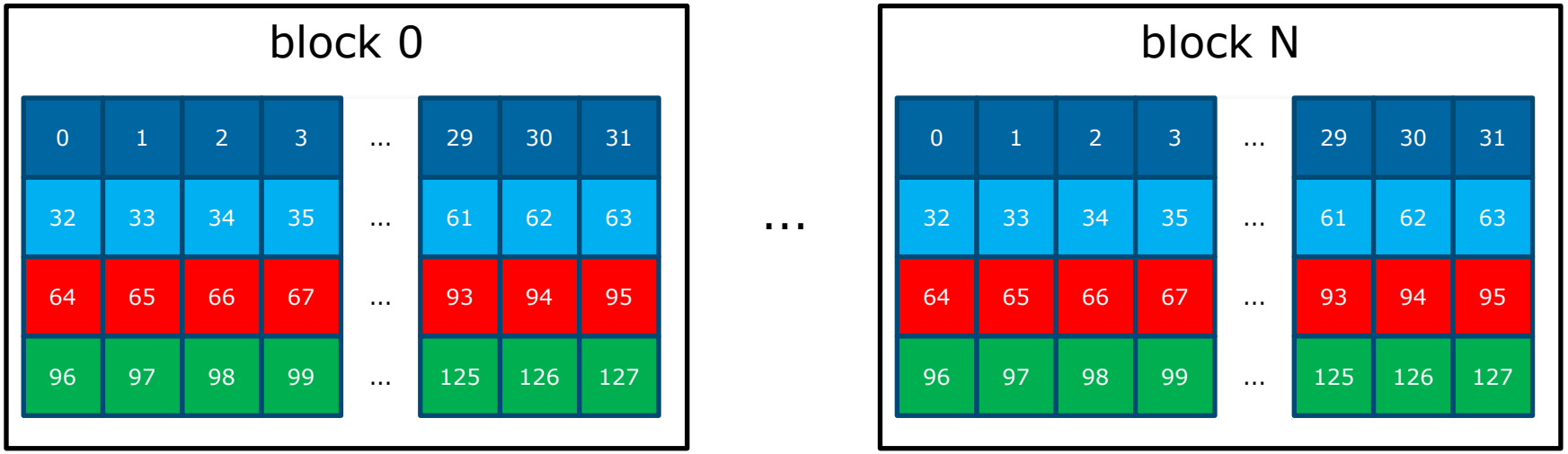
LAB: WARPnBLOCKSIZE

- Complete warp.cu
- (10 min)

WARP: Example 1

- Warps inside thread blocks of size 32x4:
  - 4 warps by block

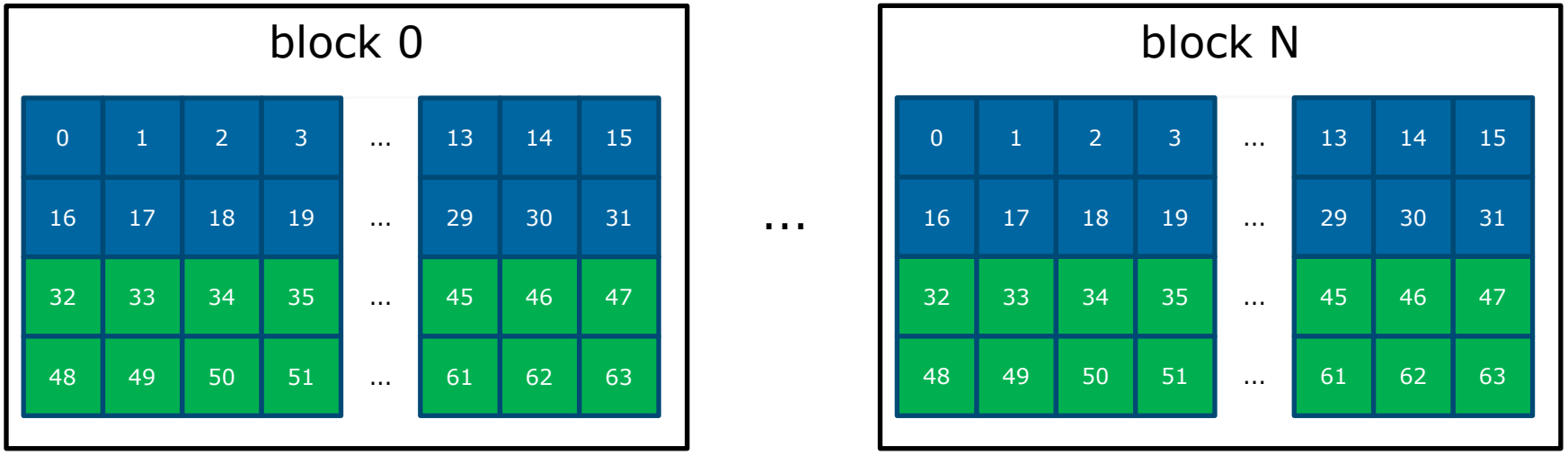
```
dim3 dimBlock;
dimBlock.x = 32;
dimBlock.y =  4;
```



WARP: Example 2

- Warps inside thread blocks of size 16x4:
  - 2 warps by block

```
dim3 dimBlock;
dimBlock.x = 16;
dimBlock.y =  4;
```

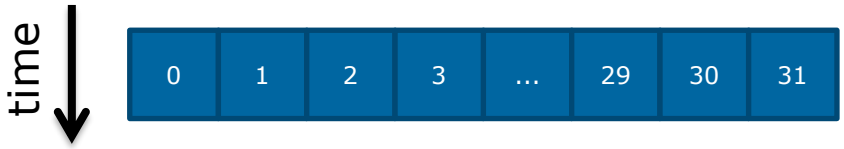


Impact of Branches

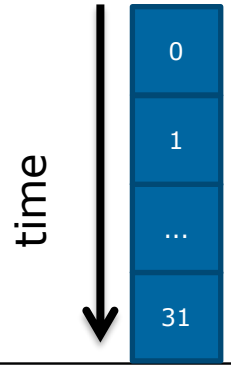
05/08/2020

BRANCHES

- Branches: if, while, ...
- Branches are not an issue if all threads within a same warp take the same branch:



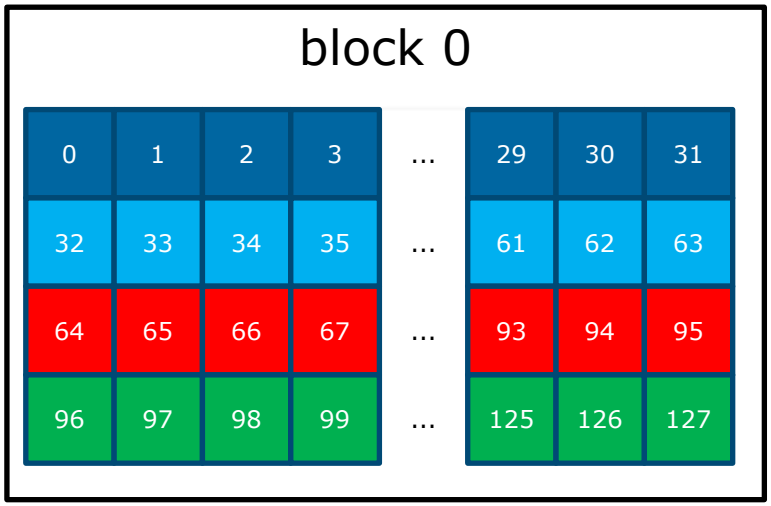
- Having threads within a same warp doing different things leads to serialization!
  - worth scenario: all threads in the warp take different branches



Branches: Example 1

- Thread blocks of size 32x4:
  - 4 warps by block

```
dim3 dimBlock;
dimBlock.x = 32;
dimBlock.y =  4;
```



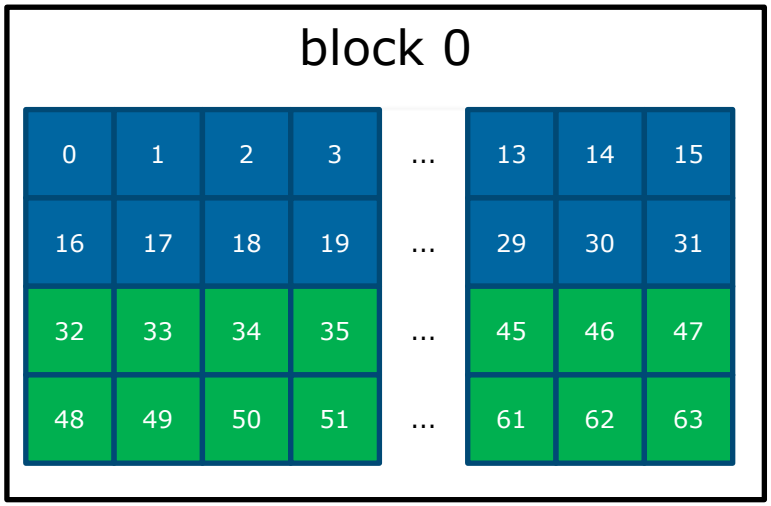

```
__global__ void myKernel(...){ //no serialization

  int id = threadIdx.x + threadIdx.y * blockDim.x
         + threadIdx.z * blockDim.x * blockDim.y;

  if(threadIdx.y % 2 == 0){
    do 1
  }else{
    do 2
  }
}
```

```
__global__ void myKernel(...){ //serialization

  int id = threadIdx.x + threadIdx.y * blockDim.x
         + threadIdx.z * blockDim.x * blockDim.y;

  if(threadIdx.x <16){
    do 1
  }else{
    do 2
  }
}
```

Branches: Example 2

- Thread blocks of size 16x4:
  - 4 warps by block

```
dim3 dimBlock;
dimBlock.x = 16;
dimBlock.y =  4;
```




```
__global__ void myKernel(...){ //serialization

  int id = threadIdx.x + threadIdx.y * blockDim.x
         + threadIdx.z * blockDim.x * blockDim.y;

  if(threadIdx.y % 2 == 0){
    do 1
  }else{
    do 2
  }
}
```

```
__global__ void myKernel(...){ //no serialization

  int id = threadIdx.x + threadIdx.y * blockDim.x
         + threadIdx.z * blockDim.x * blockDim.y;

  if(threadIdx.y < 2 ){
    do 1
  }else{
    do 2
  }
}
```

Global Memory Accesses

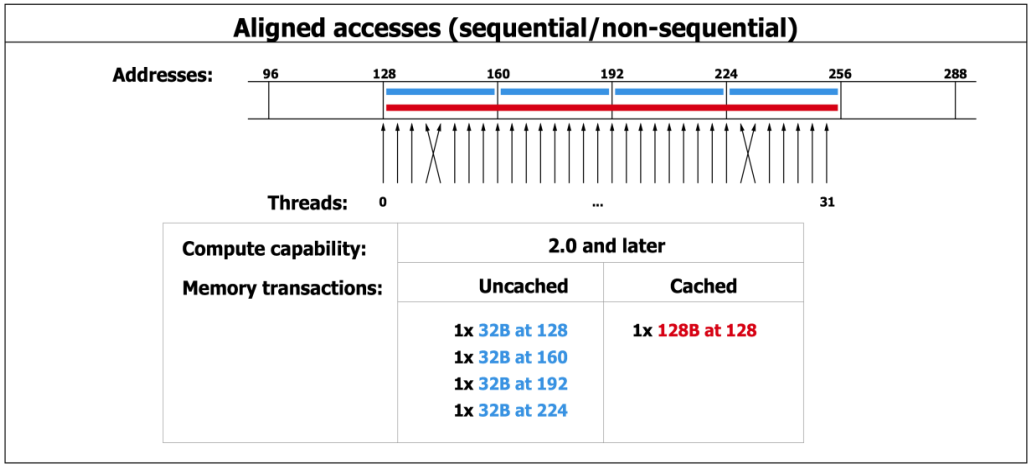
05/08/2020

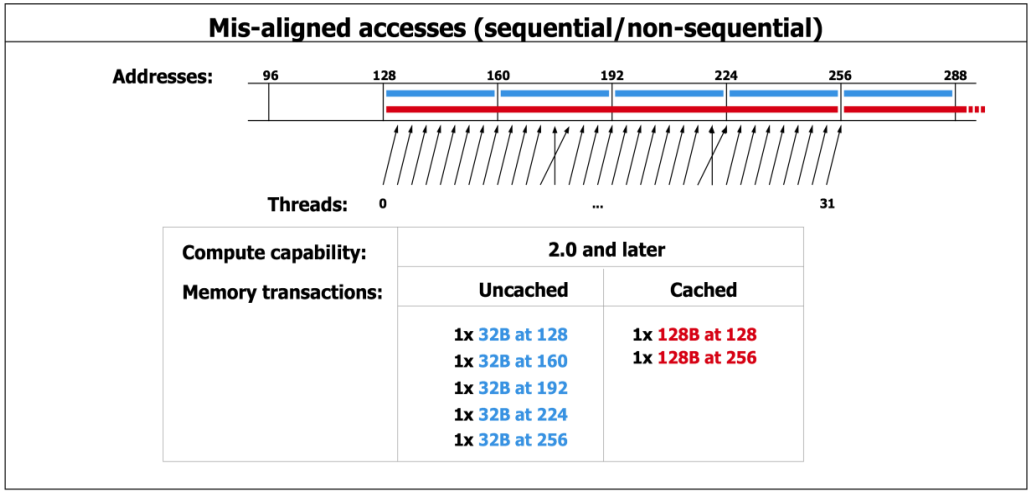
Global Memory Accesses

- Global memory is accessed via 32, 64 or 128 bytes memory transactions
- These memory transactions must be naturally aligned:
  - Only the 32, 64 or 128 bytes segments of device memory that are aligned to their size (i.e., whose first address is a multiple of their size) can be read or written by memory transactions
- When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of its threads into one or more of these memory transactions:
  - depending on:
    - the size of the word accessed by each thread
    - the distribution of the memory addresses across the threads

Global Memory Accesses

- Global memory is cache in L2
- Each multiprocessor has a L1
- Cache:
  - data will be cached in L1. This policy depends on:
    - architecture (3.x,6.x,...)
    - compilation options
    - (see documentation)





Good Memory Accesses

- Use aligned type (size 1, 2, 4, 8 or 16 Bytes)
  - struct __align__(16) {float x; float y; float z;};
    - float3: 3*4 = 12
    - float4: 4*4 = 16
- Even better **use** of **structure of arrays** instead of array of structures!
- Use 1D array. In 2D, use 1D access:
  - a[x + y * nx]
- Even better in 2D:
  - use cuMemAllocPitch()
    - add padding to have each line aligned

Lab: GLOBAL_MEMORY_ACCESSES

- Complete global_access_offset.cu
  - Small recap on events -> check cheat sheet
  - Compile with –O0
  - Run with: `for i in {0..32}; do ./offset.exe $i ; done`

Copy Kernel: Mis-Aligned Accesses Effect

```
__global__ void copy_global(int offset, const int*  __restrict__ in, int *out){
  int idx =  threadIdx.x + blockIdx.x * blockDim.x;
  out[idx + offset] = in[idx + offset];
}
```

offset (1= 4 Bytes)	time in µs
0	**1078.656**
1	1174.720
2	1171.520
3	1175.456
4	1174.336
5	1174.560
6	1172.992
7	1176.320
8	**1164.416**
9	1185.824
10	1187.744
11	1185.696
12	1186.144
13	1185.440
14	1185.696
15	1187.360
16	**1165.792**

offset (1= 4 Bytes)	time in µs
17	1184.992
18	1185.280
19	1184.416
20	1183.264
21	1184.704
22	1185.024
23	1186.400
24	**1167.808**
25	1186.688
26	1187.200
27	1184.768
28	1186.976
29	1187.840
30	1189.472
31	1186.080
32	**1068.224**

Lab: GLOBAL_MEMORY_ACCESSES

- Complete global_access_stride.cu

Copy Kernel: Stride Effect

```
__global__ void copy_global(int stride, const int*  __restrict__ in, int *out){
  int idx =  threadIdx.x + blockIdx.x * blockDim.x;
  out[idx * stride] = in[idx * stride];
}
```

stride	time in µs
1	1091.648
2	2694.624
3	3974.112
4	5305.536
5	6723.072
6	8128.608
7	9554.144
8	10944.320
9	11256.000
10	11547.104
11	11819.072
12	12184.064
13	12530.112
14	12919.040
15	13300.896
16	13354.432
17	14070.464
18	14800.096
19	16281.440
20	17069.568

stride	time in µs
21	17948.545
22	18972.928
23	19875.199
24	20824.256
25	21818.207
26	22839.297
27	23967.936
28	24991.039
29	26094.176
30	27074.623
31	28111.424
32	28460.607
33	29365.248
34	29742.432
35	29933.119
36	30344.928
37	30212.352
38	30526.912
39	30723.584

Restrict Keyword

- nvcc supports restricted pointers via the __restrict__ keyword.
- Restricted pointers were introduced in C99 to alleviate the aliasing problem that exists in C-type languages, and which inhibits all kind of optimization from code re-ordering to common sub-expression elimination.

```
void foo(const float* a,
         const float* b,
         float* c)
{
   c[0] = a[0] * b[0];
   c[1] = a[0] * b[0];
   c[2] = a[0] * b[0] * a[1];
   c[3] = a[0] * a[1];
   c[4] = a[0] * b[0];
   c[5] = b[0];
   ...
}
```

Restrict Keyword

- nvcc supports restricted pointers via the __restrict__ keyword.
- Restricted pointers were introduced in C99 to alleviate the aliasing problem that exists in C-type languages, and which inhibits all kind of optimization from code re-ordering to common sub-expression elimination.

```
void foo(const float* __restrict__ a,
         const float* __restrict__ b,
         float* __restrict__ c)
{
   c[0] = a[0] * b[0];
   c[1] = a[0] * b[0];
   c[2] = a[0] * b[0] * a[1];
   c[3] = a[0] * a[1];
   c[4] = a[0] * b[0];
   c[5] = b[0];
   ...
}
```

compiler optimizations

```
float t0 = a[0];
float t1 = b[0];
float t2 = t0 * t1;
float t3 = a[1];
c[0] = t2;
c[1] = t2;
c[4] = t2;
c[2] = t2 * t3;
c[3] = t0 * t3;
c[5] = t1;
...
```

Read-Only Data Cache

- CUDA architectures have a read-only data cache
- The compiler try to detect data that are read-only to use the read-only data cache
  - but may not succeed
- Marking pointers with the **const** and **__restrict__** qualifiers increases the likelihood that the compiler will detect the read-only condition.
- The use of the read-only data cache an be forced using:
  - T __ldg(const T* address);

```
__global__ void myKernel_lgd(int size_x, int size_y, int size_z, int* in, int* out){
  ...
  int temp = __ldg(&in[idx + idy *size_x + (idz+0) * size_x * size_y])
```

Grid-Stride Loops

05/08/2020

Grid-Stride Loops

```
for(int i = 0; i < N; i++){
   op(A[i]);
}
```

Porting ↓ to GPU

```
template<class T>
__global__ void kernel(int N, T *A){
   //compute idx and range check
   op(A[idx]);
}
…
kernel<<<numBlocks, numThreads>>>(N, A);
```

What if N is bigger than the grid?

Grid-Stride Loops

- The code only works if the GPU can handle a grid equal in size to the amount of elements.

- On CC 7.5
  - Maximum number of threads per block: 1024
  - Maximum x-dimension of a grid of thread blocks: 2<<32 - 1

- Distribute more work to each thread without changing the global memory access pattern by **grid-stride** access

```
template<class T>
__global__ void kernel(int N, T *A){
   for(int idx = blockIdx.x * blockDim.x + threadIdx.x; idx < N;
           idx += blockDim.x * girdDim.x)
      op(A[idx]);
}
numBlocks = min((N + numThreads - 1)/numThreads, MAX_BLOCKS);
kernel<<<numBlocks, numThreads>>>(N, A);
```

Grid-Stride Loops

- The code only works if the GPU can handle a grid equal in size to the amount of elements.

- On CC 7.5
  - Maximum number of threads per block: 1024
  - Maximum x-dimension of a grid of thread blocks: 2<<32 - 1

- Distribute more work to each thread without changing the global memory access pattern by **grid-stride** access

```
template<class T>
__global__ void kernel(int N
    for(int idx = blockIdx.x
             idx += blockDim.
        op(A[idx]);
}
numBlocks = min((N + umThreads - 1)/numThreads, MAX_BLOCKS);
kernel<<<numBlocks, numThreads>>>(N, A);
```

Independent to problem size!
MAX_BLOCKS adaptable to the hardware

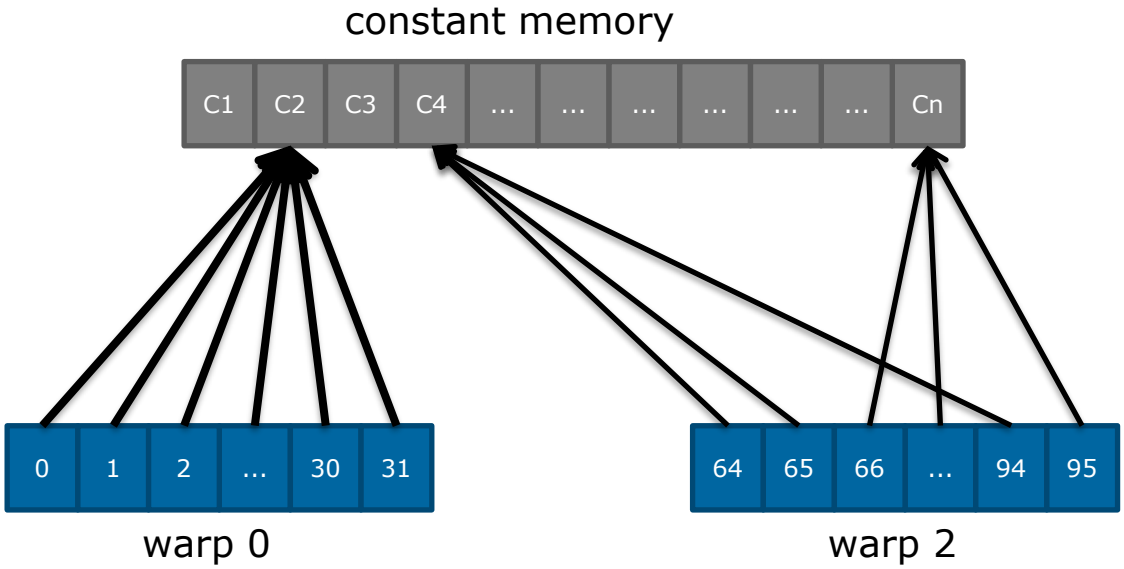
Constant Memory

05/08/2020

Constant Memory

- The constant memory space resides in device memory
  - size 64kB
- Each multiprocessor has a **read-only constant cache** that is shared by all functional units and speeds up reads from the constant memory space
- When threads within a warp perform a request to the constant memory:
  - the request is split into as many separate requests as there are different memory addresses
  - the resulting requests are then serviced:
    - at the throughput of the constant cache in case of a cache hit
    - at the throughput of device memory otherwise

Example



- Warp 0:
  - threads access the same memory address: served in 1 request
- Warp 2:
  - threads access to 2 different memory addresses: served in 2 requests

Constant Memory

- Declared with the **__constant__** memory space specifier
  - lifetime of the CUDA context in which it is created
  - distinct object per device

- Initialized with **cudaMemcpytoSymbol**
  - cudaError_t cudaMemcpytoSymbol(const char * symbol, const void * src, size_t count , size_t offset=0, enum cudaMemcpyKind)

- Retrieved in kernel via the symbol

Lab: Constant Memory

- Change coef to use the constant memory

```
__global__ void mykernel(float *a, float *coef){
  size_t index = threadIdx.x + blockIdx.x * blockDim.x;

  for(int i=0; i<4; i++)
    a[index] = a[index] + coef[i];

}

int main(){

  ...
  float *d_coef;
  cudaMalloc( (void **) &d_coef, 4 *sizeof(float) );
  cudaMemcpy(d_coef, &cpucoeff, 4 *sizeof(float), cudaMemcpyHostToDevice);
  ...

}
```

Example

```
__constant__ float coeff[4];


__global__ void mykernel(float *a){
  size_t index = threadIdx.x + blockIdx.x * blockDim.x;

  for(int i=0; i<4; i++)
     a[index] = a[index] + coeff[i];

}


int main(int argc, char ** argv){
  float cpucoeff[4] = {11.32f, 65.36f,  8.32f, 47.28f};

  ...
  cudaMemcpyToSymbol(coeff, cpucoeff, 4 *sizeof(float),0, cudaMemcpyHostToDevice);
  ...
  mykernel<<<...>>>(a);
  ...
}
```

Shared Memory

05/08/2020

Shared Memory

- The shared memory is a **memory on a multiprocessor (on-chip)**
  - small: usually 48 KB per block (dynamic: up to 96 on Volta)

- This **memory** is **shared between threads** of a **same threads block**
  - if *thread 1* from *block 3* writes in shared memory:
    - all threads from block 3 can access the new data
    - threads from other blocks can not

- The shared memory:
  - **is faster than the global memory** (shared memory closer to cores)
    - registers : ~ *few cycles*
    - shared memory : ~ ***dozens*** *of cycles*
    - global memory : ~ ***hundreds*** *cycles*

- Shared memory can be seen as a cache driven by the user
  - Physically on same memory block as L1

Shared Memory – Preferences

- L1 cache and shared memory have to split on chip memory resources.
- Ratio can be set via **`cudaDeviceSetCacheConfig(arg)`**.
  - Can be set globally or per kernel
- ARG: (for cards which have 64 KB of shared memory per SM)
  - **`cudaFuncCachePreferShared`**
    - 48 KB shared memory and 16 KB L1 cache
  - **`cudaFuncCachePreferL1`**
    - 16 KB shared memory and 48 KB L1 cache
  - **`cudaFuncCachePreferEqual`**
    - 32 KB shared memory and 32 KB L1 cache
- Supported modes depend on architecture ! **Only a preference !**

How to Allocate Shared Memory

- Two ways to allocate shared memory : **statically** and/or **dynamically**
- Shared Memory is declared using the **__shared__** memory space specifier
- Static allocation:

```
__global__ void  myKernel(int offset, const int*  __restrict__ in, int *out){

 __shared__   float   shArray[32];

 int idx =  threadIdx.x + blockIdx.x * blockDim.x;
 ...
 out[idx + offset] = in[idx + offset] + shArray[val];
}
```

How to Allocate Shared Memory

- Dynamic allocation :
  - as an extern variable
  - size determined at runtime by the kernel's caller via a launch configuration argument
    - third argument, size in Bytes

```
__global__ void  myKernel(int offset, const int*  __restrict__ in, int *out){
 extern __shared__  float shArray[];
  ...
  out[idx + offset] = in[idx + offset] + shArray[val];
}

void mycall(...){
  ...
  myKernel<<< DimGrid, DimBlock, 32* sizeof(float), Stream >>>
}
```

How to Allocate Shared Memory

- Dynamic allocation, several arrays example

```
__global__ void  myKernel(int offset, const int*  __restrict__ in, int *out){
 extern __shared__  float shArray[];
  ...
  short* array0 = (short*)shArray;
  float*  array1 = (float*)&shArray0[128];
  int*     array2 = (int*)&shArray1[64];
  ...
}

void mycall(...){
  ...
  myKernel<<< DimGrid, DimBlock, 256* sizeof(float), Stream >>>
}
```

- !!!! Pointers need to be aligned to the type they point to

How to Use Shared Memory

- Shared memory is shared between threads of a same threads block
  - example: t0 writes in shared memory, t56 can read the data written by t0
- Threads in a block are executed by warps
- The execution order of warps is unknown
- When t56 can be sure that t0 has finished its write?
  - we need a way to synchronize threads within a block
- **__syncthreads()**

```
myshared[threadIdx.x] = threadIdx.x +10;

__syncthreads();

a[threadIdx.x] = myshared[threadIdx.x] ;
```

__syncthreads()

- Acts as **a barrier** at which **all threads in the block must wait** before any is allowed to proceed

- Before Volta (V100):
  - __syncthreads() could succeed without being executed by every thread
    - as long as at least some thread in every warp reached the barrier

- Since Volta:
  - __syncthreads() is enforced per thread and thus will not succeed until reached by all non-exited threads in the block

- !!! A code with no issues on Kepler or Pascal can causes deadlock on Volta !!!
  - be very careful with __syncthreads() in branches !!!

TP: N_body revisited

/TP_CUDA_C/SHARED_MEMORY/

Remember the n_body problem. So far, we simply transformed the `update_position(…)` and `update_velocity(…)` into kernels and managed the data movement.

Task: Adapt the kernels to the GPU architecture. Identify the data movement from global memory to the registers. **Speed up the computation by using shared memory.**

Try to keep the shared memory consumption below 32KB per Block.

Make sure that results remain correct!

Try with run with `./nBody_basic  32768 1000`

Number of particles Timesteps simulated

TP: N_body revisited Hints

- **Problem**: In `update_velocity(…)`, each thread has to load the position of each particle in the j-loop.

- **Idea**:
  - Load a chunk of data to shared memory (each thread performs one load)
  - Wait until shared memory is populated
  - Perform distance calculations on shared memory
  - Wait until shared memory is not used anymore
  - Load next chunk and start over

- **Speedup:**

TP: N_body revisited Hints

- **Problem**: In `update_velocity(…)`, each thread has to load the position of each particle in the j-loop.

- **Idea**:
  - Load a chunk of data to shared memory (each thread performs one load)
  - Wait until shared memory is populated
  - Perform distance calculations on shared memory
  - Wait until shared memory is not used anymore
  - Load next chunk and start over

- **Speedup:**
  - On a V100 we obtain a speedup around 15%.
- **Runtime:**

TP: N_body revisited Hints

- **Problem**: In `update_velocity(…)`, each thread has to load the position of each particle in the j-loop.

- **Idea**:
  - Load a chunk of data to shared memory (each thread performs one load)
  - Wait until shared memory is populated
  - Perform distance calculations on shared memory
  - Wait until shared memory is not used anymore
  - Load next chunk and start over

- **Speedup:**
  - On a V100 we obtain a speedup around 15%.
- **Runtime:**

	No shared mem	Shared mem	Speedup
default	6s	5s	~15%
-g –G	112s	105s	~15%

Matrix Transpose Example

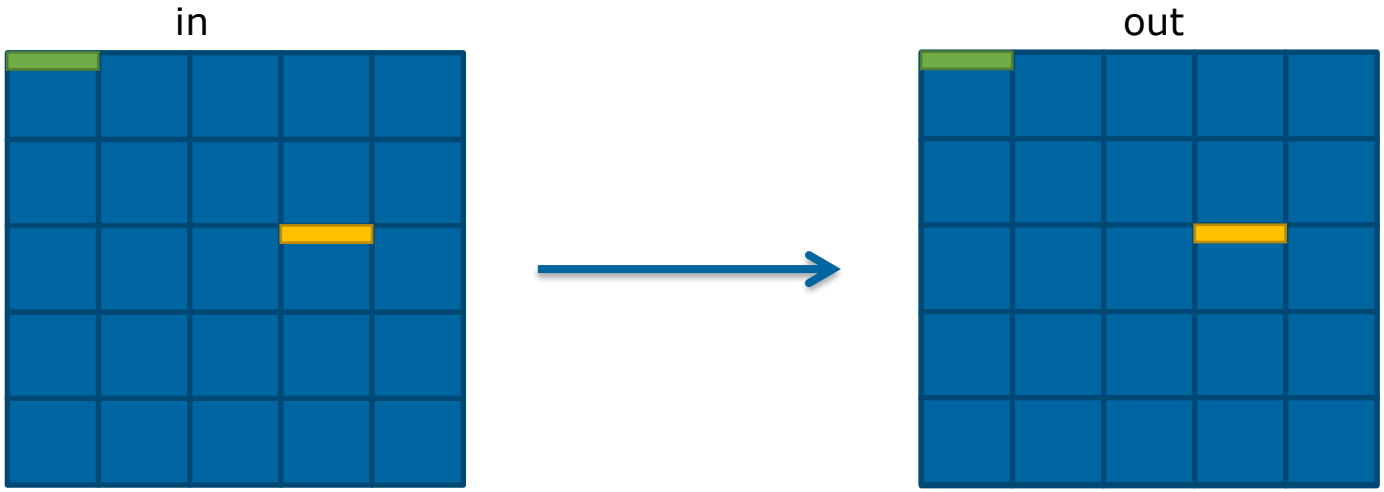
05/08/2020

LAB: Matrice Transpose

- 1) complete copy2d.cu
- 2) copy copy2d.cu into transpose_global.cu
  - modify transpose_global.cu to perform a matrice transposition
- 3) complete transpose_block.cu to perform a matrice transposition by block
- 4) in transpose_block.cu, use shared memory as indermediate buffer for the transposition by block
- 5) complete transpose_shared.cu to perform a full transposition using shared memory
- 6) complete bank.cu
- 7) optimize bank.cu
- 8) optimize transpose_shared.cu

1st Step: Simple Copy

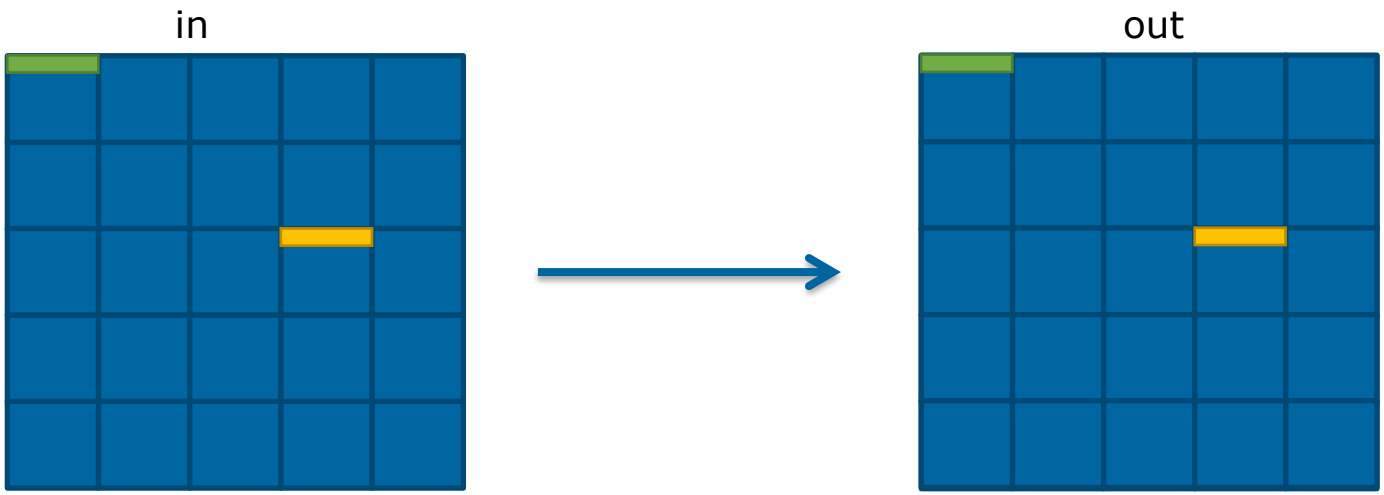
```
for(int j=0; j<size; j++)
  for(int i=0; i<size; i++)
    out[i + j *size] = in[i + j * size];
```



1st Step: Simple Copy

```
__global__ void copy2d(int size, int *in, int *out){

    ...

    out[idx + idy *size] = in[idx + idy * size];
}
```

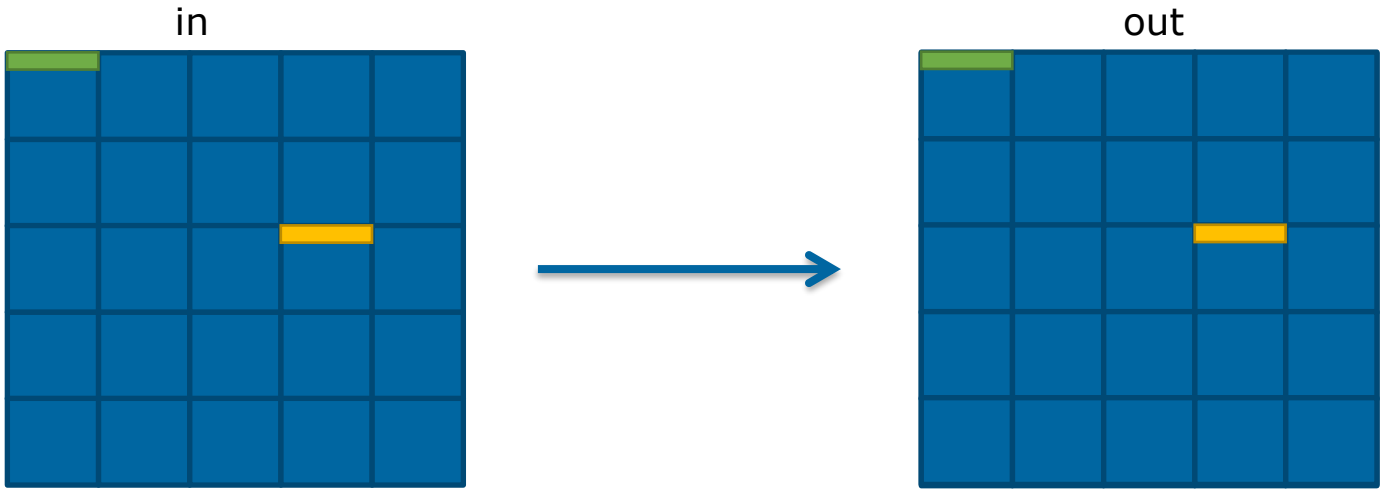
- Block size 32x32



1st Step: Simple Copy

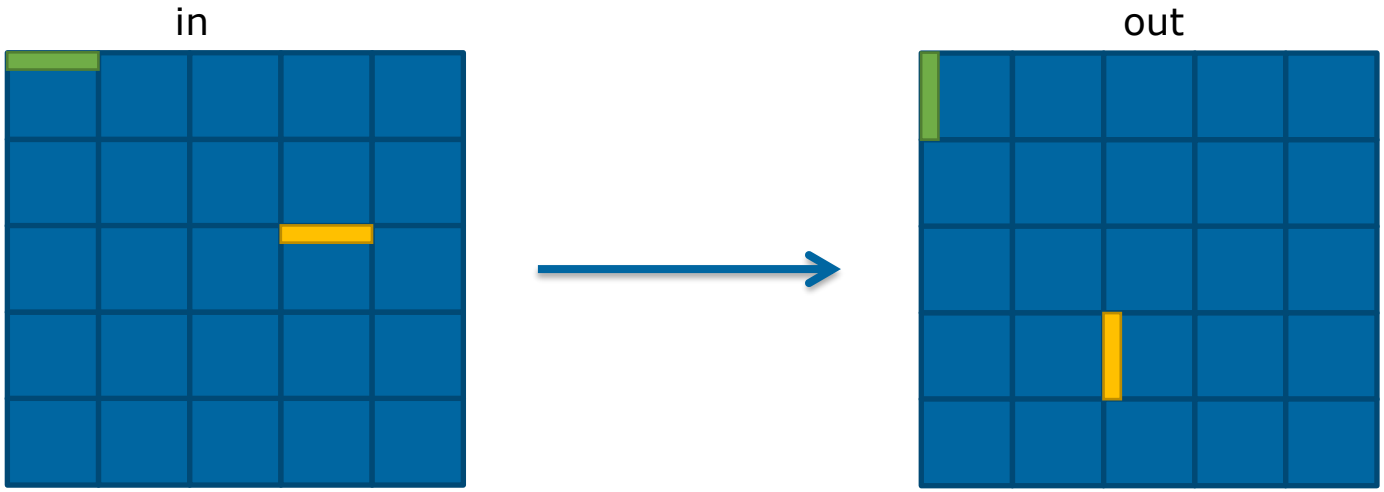
```
__global__ void copy2d(int size, int *in, int *out){

    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int idx =  tx + blockIdx.x * blockDim.x;
    int idy =  ty + blockIdx.y * blockDim.y;

    out[idx + idy *size] = in[idx + idy * size];
}
```

- Block size 32x32
- **1108.928 µs** for a 4096 *4096 matrix on a K80



Matrix Transpose: Via Global Memory Only

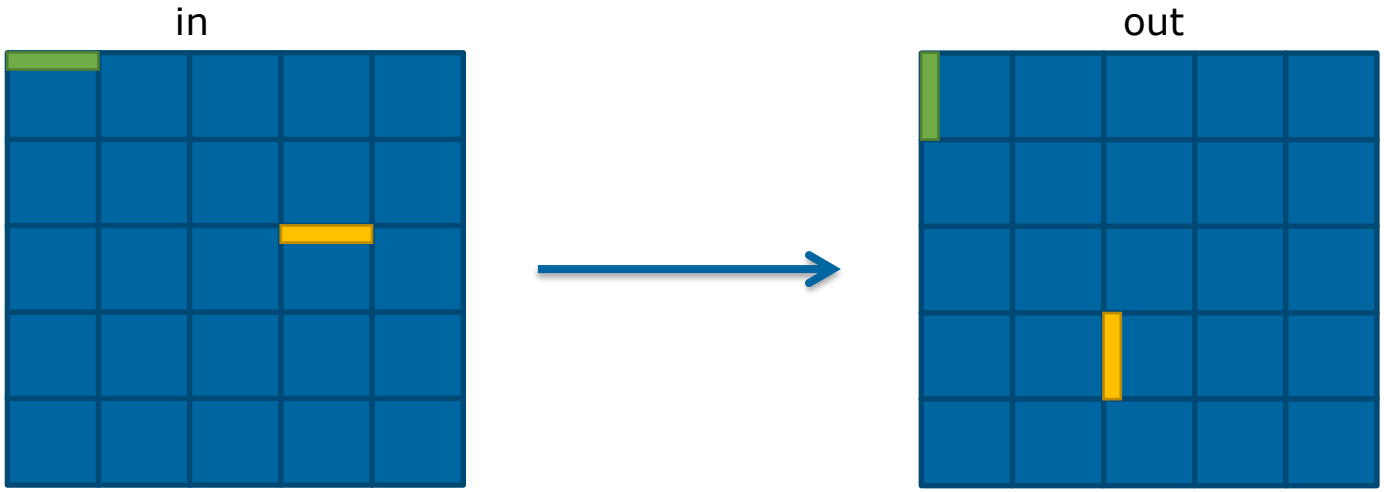
```
for(int j=0; j<size; j++)
  for(int i=0; i<size; i++)
    out[j + i *size] = in[i + j * size];
```



Matrix Transpose: Via Global Memory Only

```
__global__ void transpose_global(int size, int *in, int *out){

    ...

    out[ ... ] = in[ ... ];
}
```

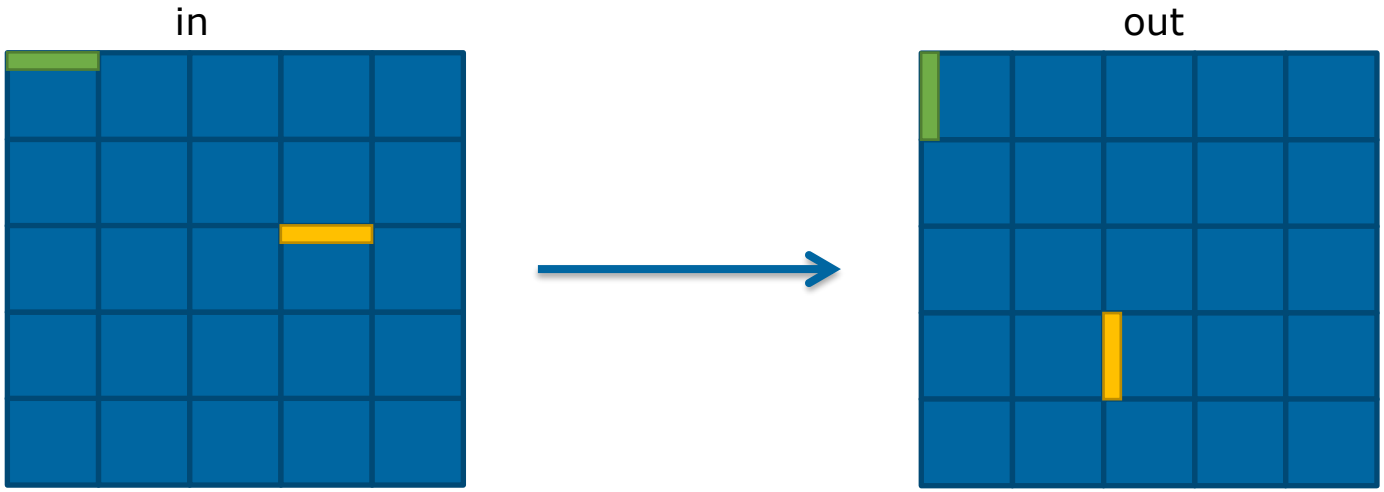
- Block size 32x32
- Writes in global memory are ???



Matrix Transpose: Via Global Memory Only

```
__global__ void transpose_global(int size, int *in, int *out){

    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int idx =  tx + blockIdx.x * blockDim.x;
    int idy =  ty + blockIdx.y * blockDim.y;

    out[idy + idx *size] = in[idx + idy * size];
}
```

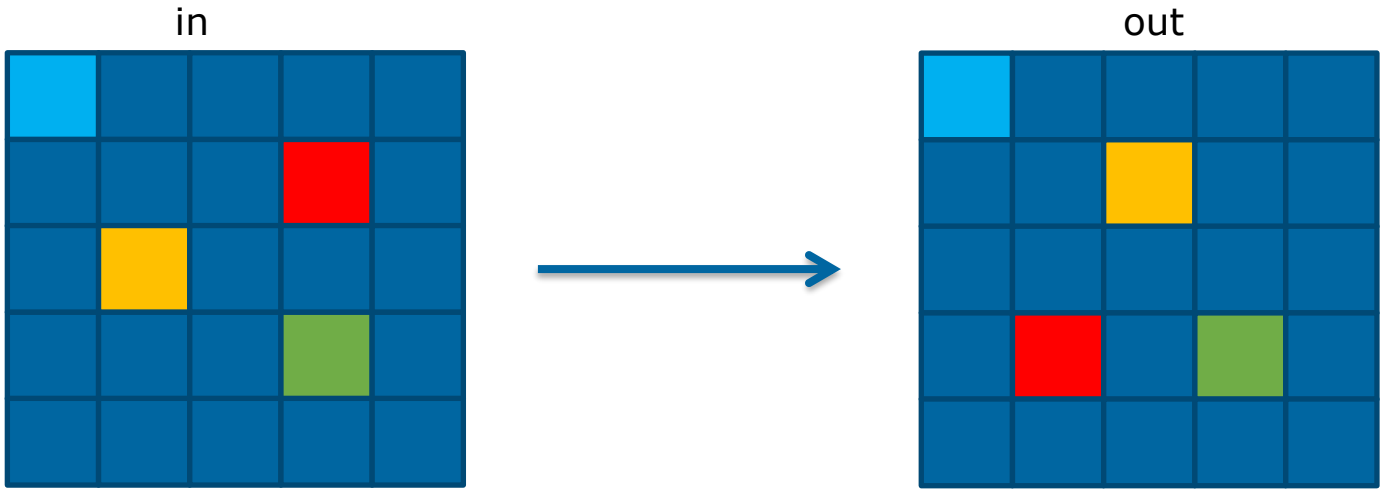
- Block size 32x32
- **3009.024 µs** for a 4096 *4096 matrix on a K80
- Writes in global memory are **non-coalesced**



1st Step Transposition Per Block

```
__global__ void transpose_block(int size, int *in, int *out){

  ...

  out[idx_out + idy_out *size] = in[idx_in + idy_in * size];
}
```

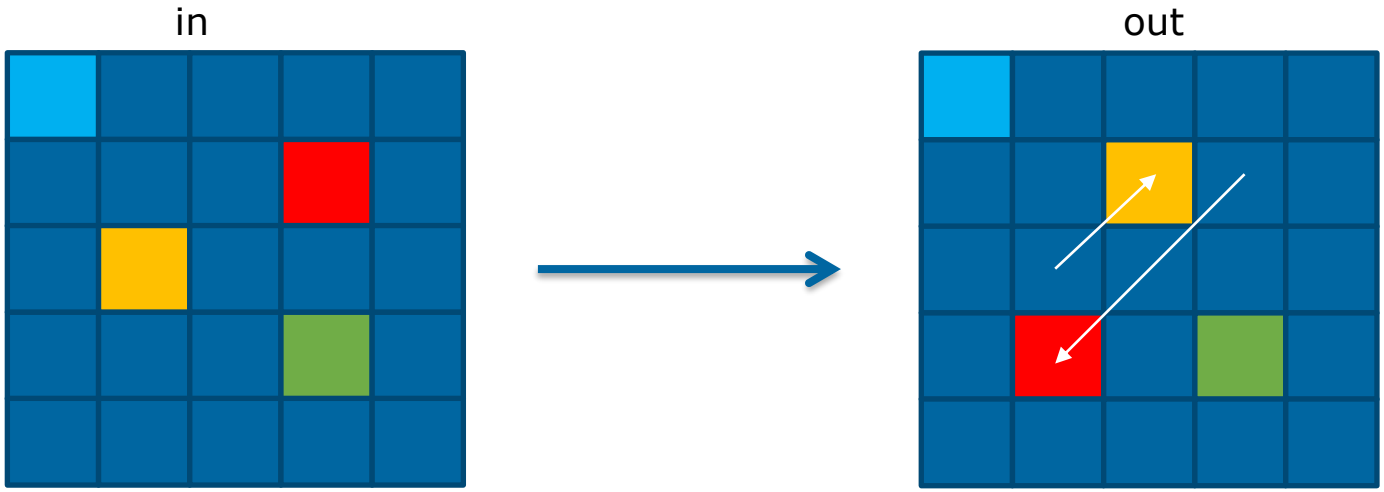
- Block size 32x32



1st Step Transposition Per Block

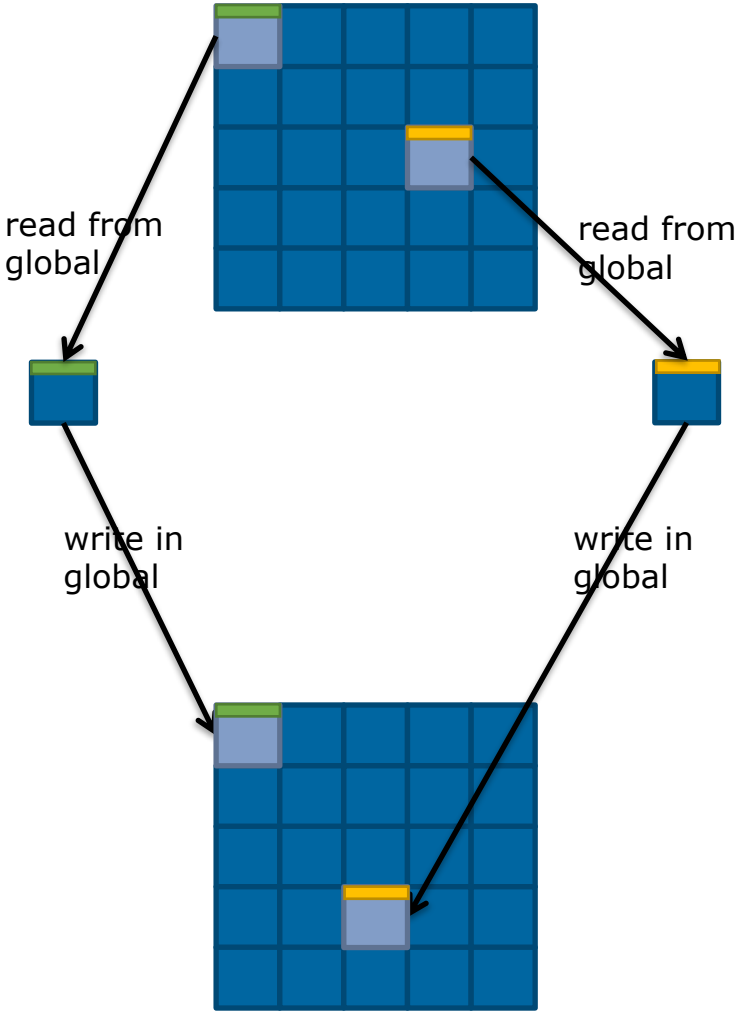
```
__global__ void transpose_block(int size, int *in, int *out){

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  out[idx_out + idy_out *size] = in[idx_in + idy_in * size];
}
```

- Block size 32x32



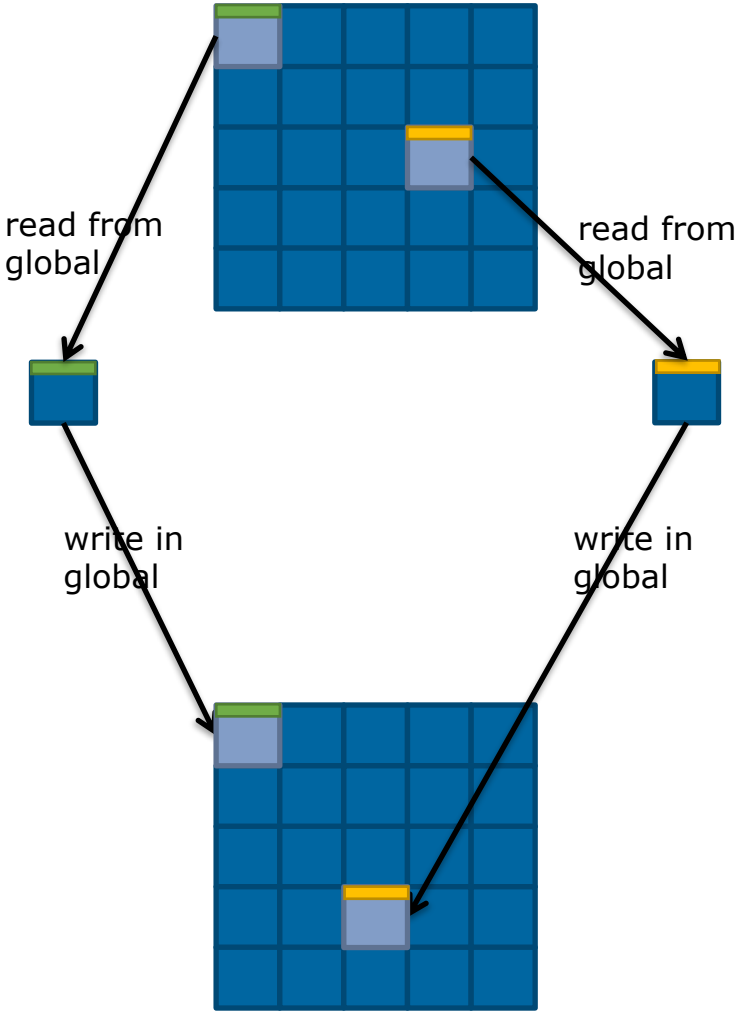
2nd Step Transposition Per Block via Shared Memory

```
__global__ void transpose_shared(int size, int *in, int *out){

 __shared__ ...

 ...

 out[idx_out + idy_out *size] = ...

}
```



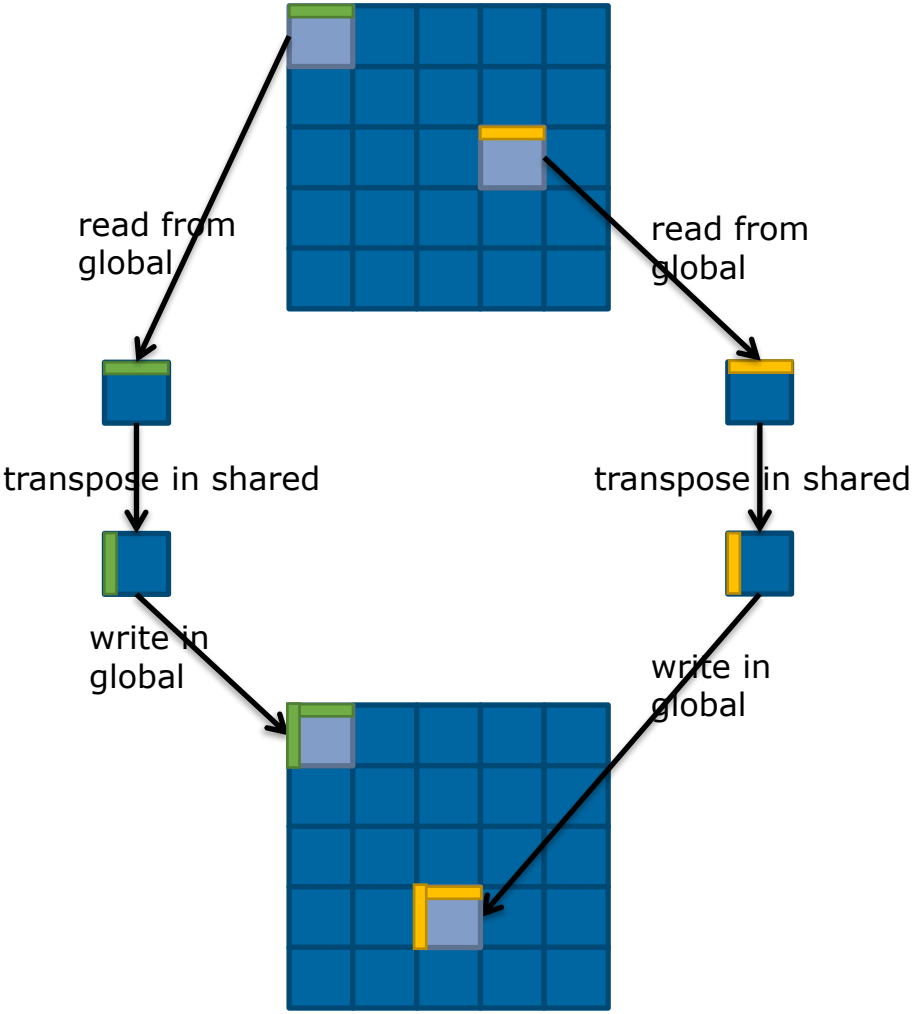
2nd Step Transposition Per Block via Shared Memory

```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][32];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[ty][tx];

}
```



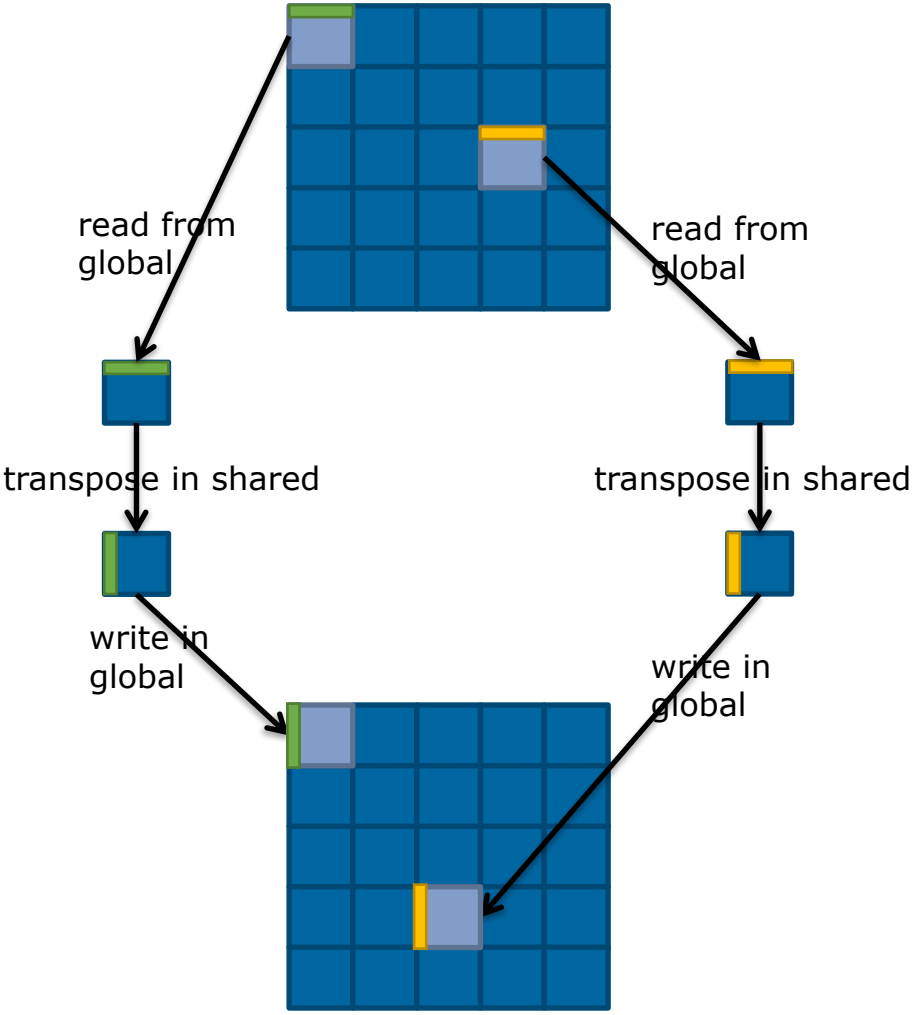
3rd Step Full Transposition Via Shared Memory

```
__global__ void transpose_shared(int size, int *in, int *out){

__shared__ ...

...

out[idx_out + idy_out *size] = ...

}
```



3rd Step Full Transposition Via Shared Memory

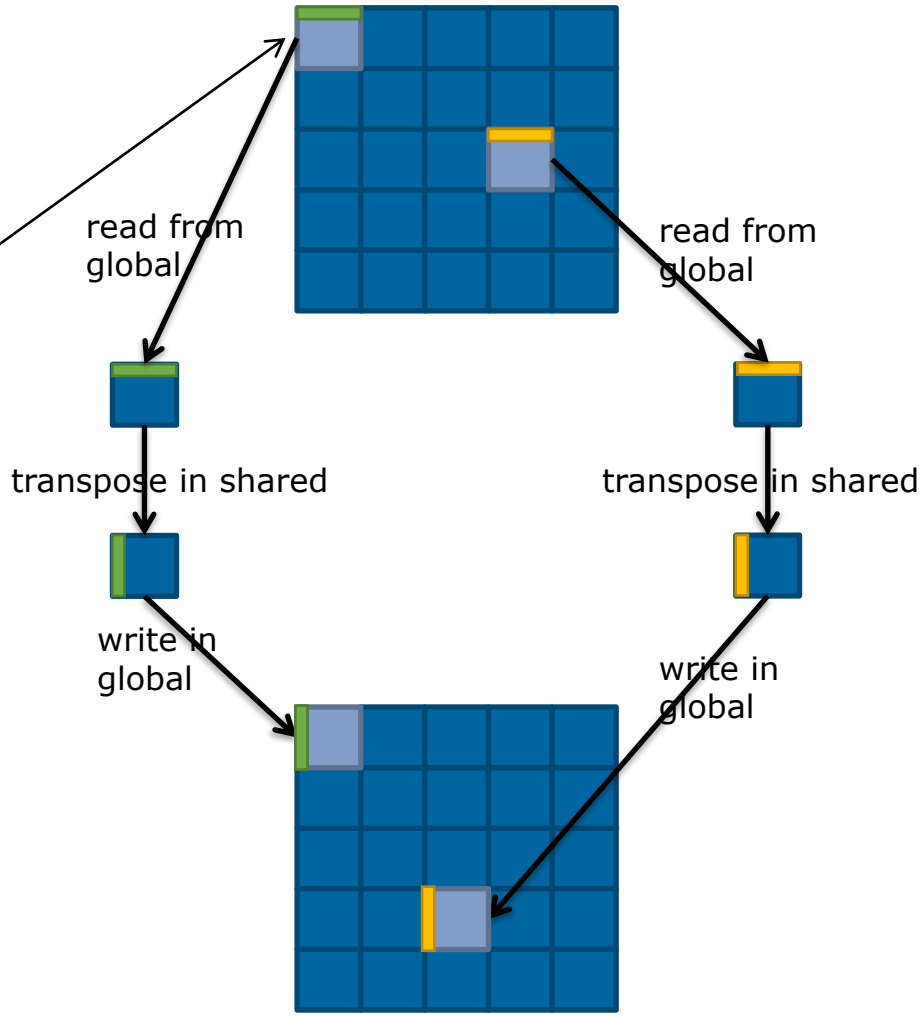
```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][32];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



- **2393.984 µs** for a 4096 *4096 matrix on a K80

3rd Step Full Transposition Via Shared Memory

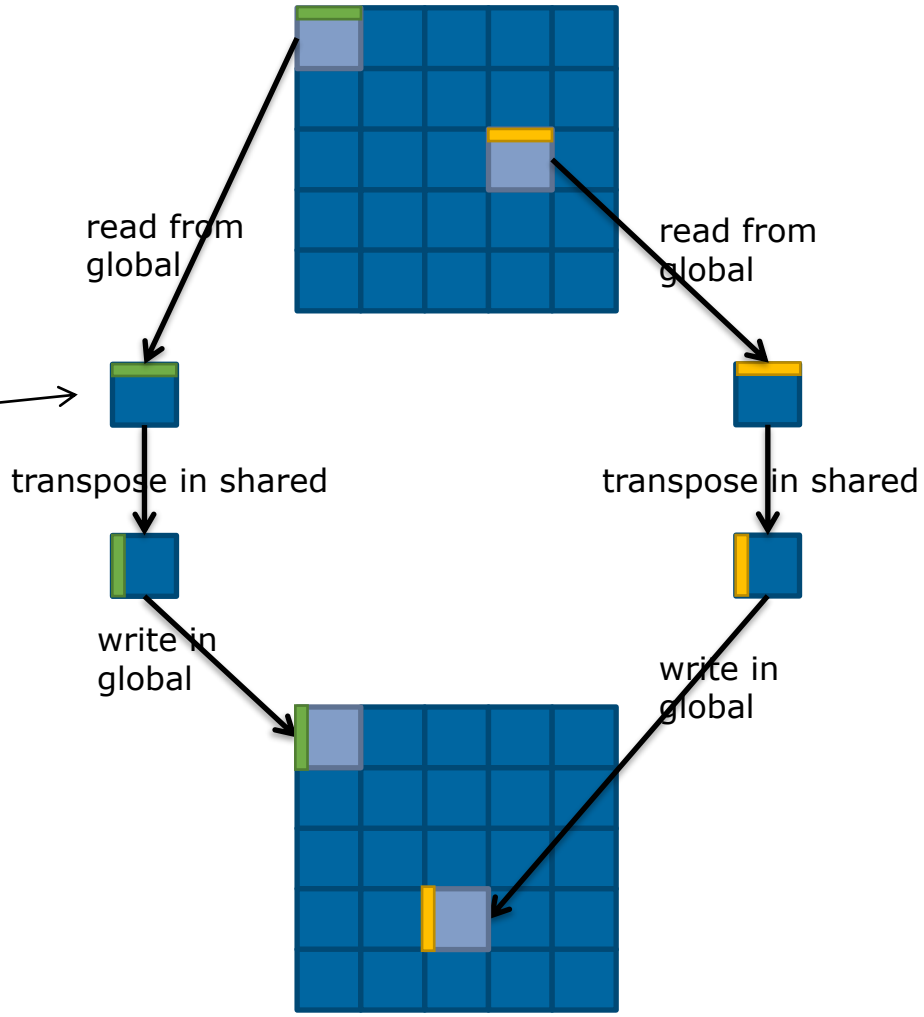
```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][32];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



- **2393.984 µs** for a 4096 *4096 matrix on a K80

3rd Step Full Transposition Via Shared Memory

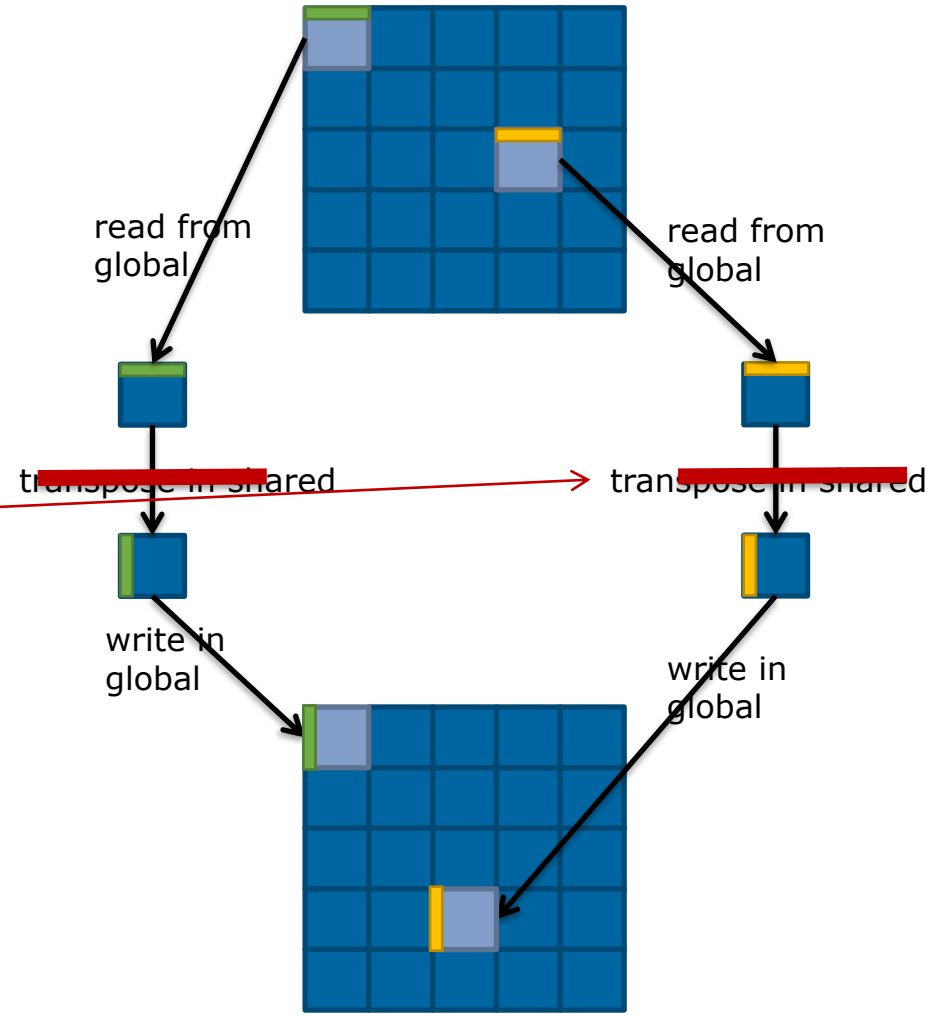
```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][32];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



- **2393.984 µs** for a 4096 *4096 matrix on a K80

3rd Step Full Transposition Via Shared Memory

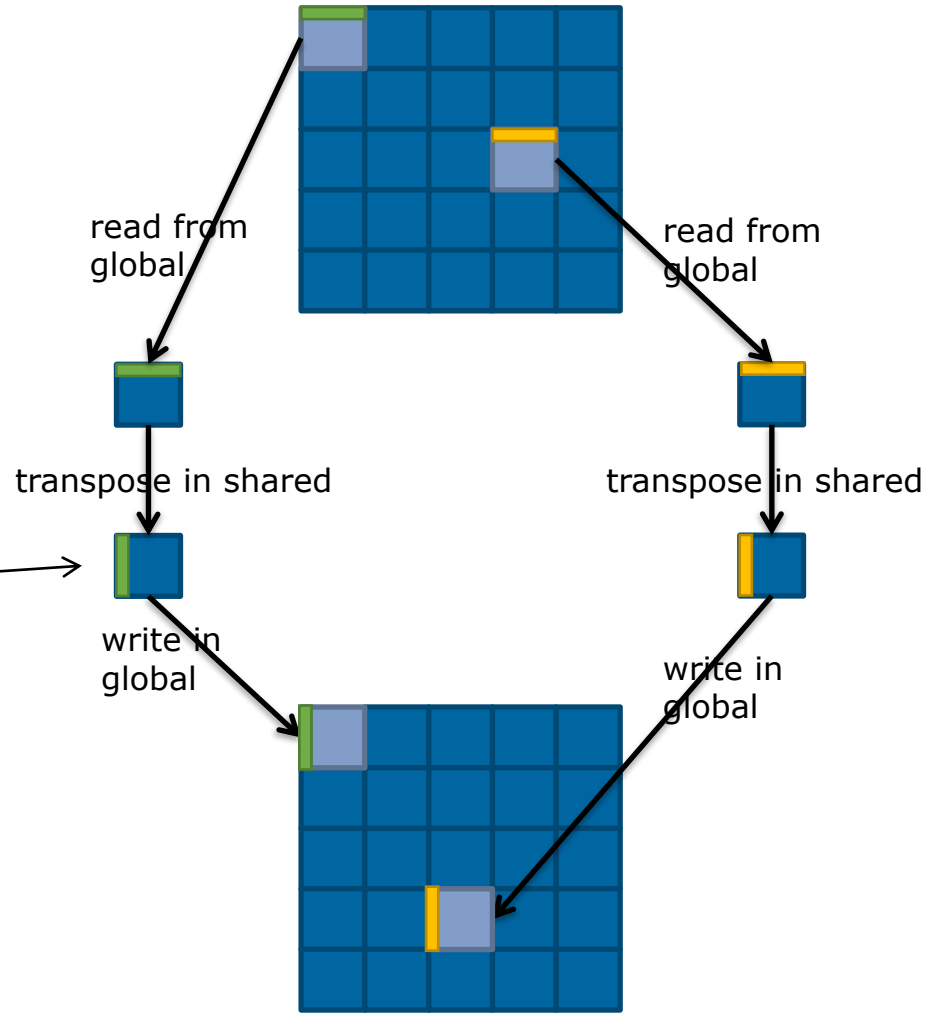
```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][32];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



- **2393.984 µs** for a 4096 *4096 matrix on a K80

3rd Step Full Transposition Via Shared Memory

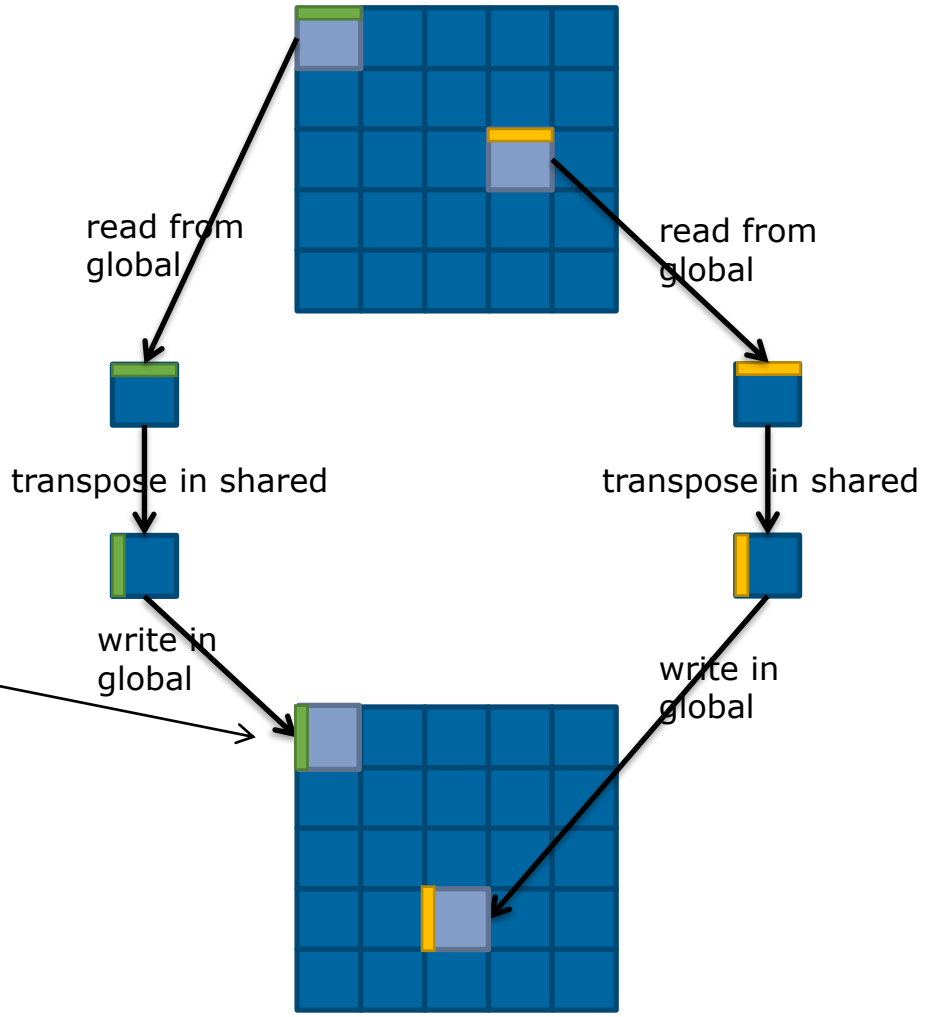
```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][32];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



- **2393.984 µs** for a 4096 *4096 matrix on a K80

3rd Step Full Transposition Via Shared Memory

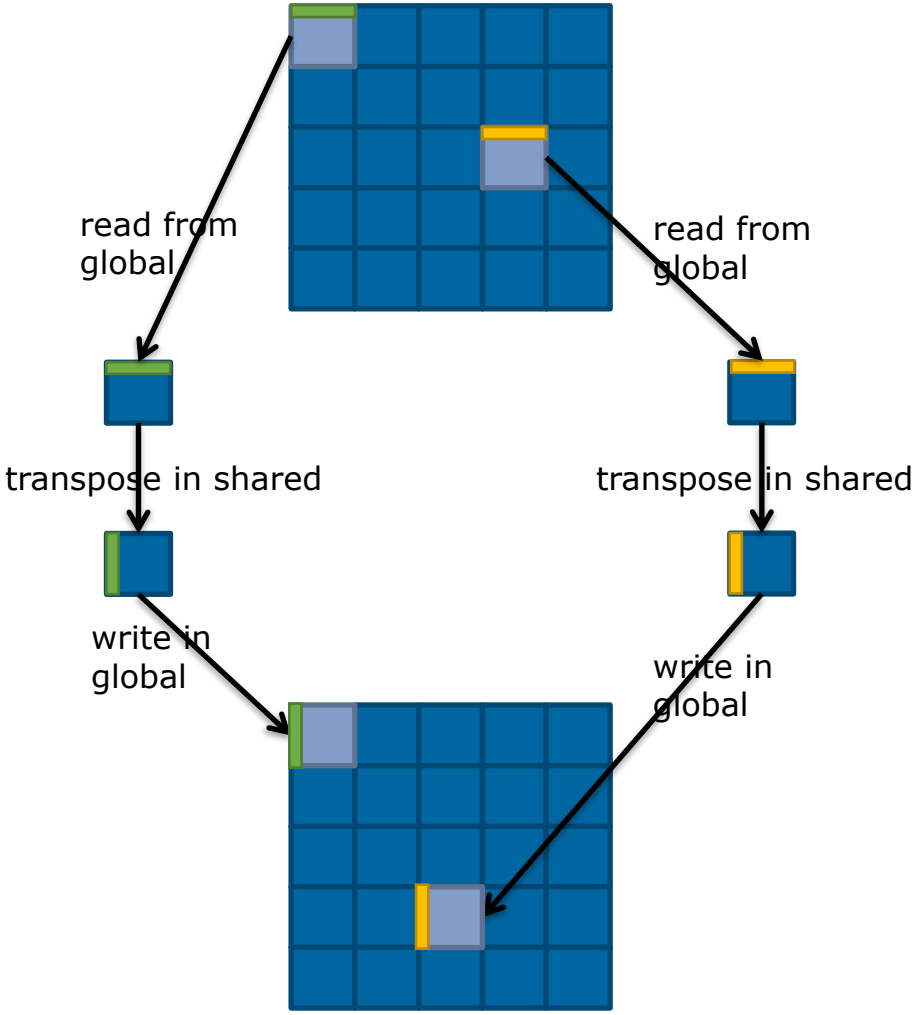
```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][32];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];
}
```



- **2393.984 µs** for a 4096 *4096 matrix on a K80

3rd Step Full Transposition Via Shared Memory

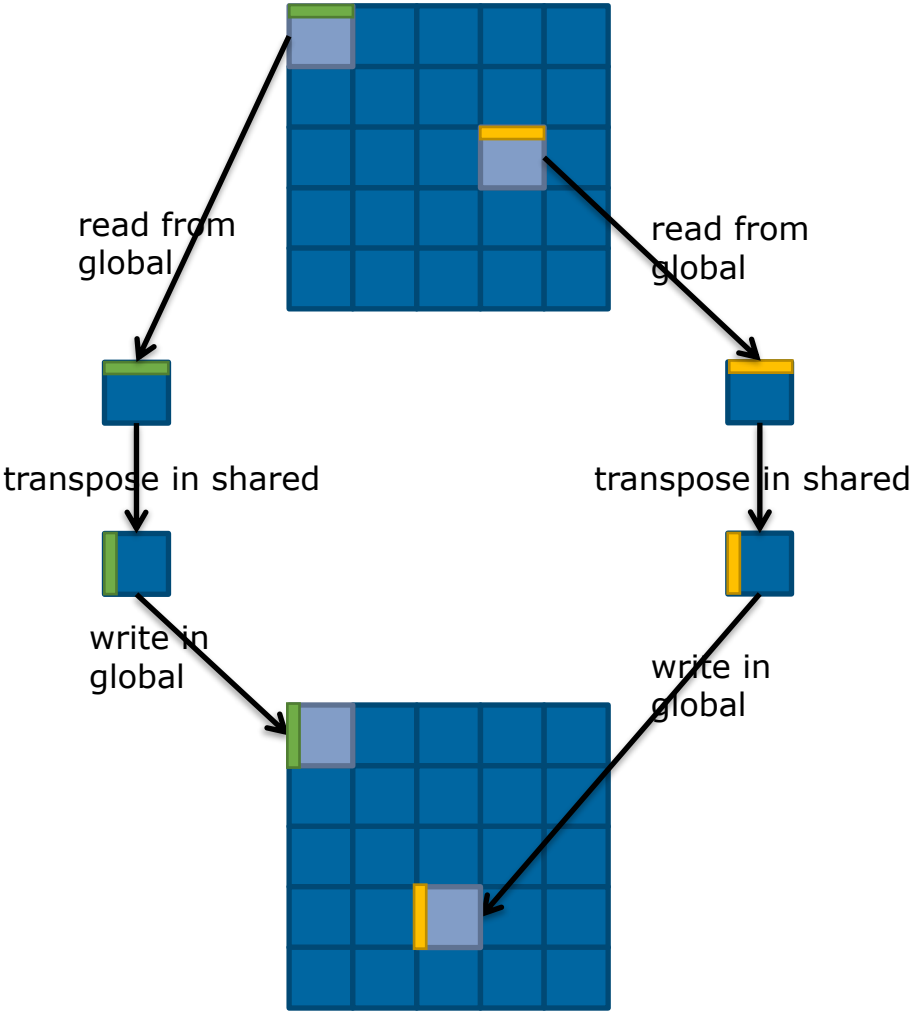
```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][32];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



- **2393.984 µs** for a 4096 *4096 matrix on a K80
- **but ?**

3rd Step Full Transposition Via Shared Memory

```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][32];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



- **2393.984 µs** for a 4096 *4096 matrix on a K80
- but bank conflicts when reading shared memory

Shared Memory Banks

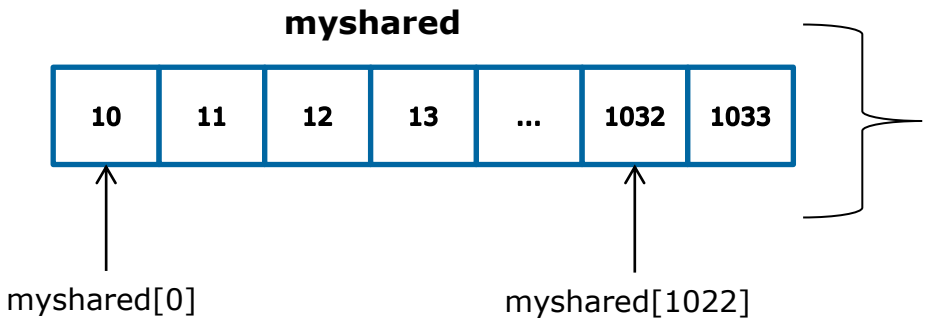
05/08/2020

Shared Memory: Banks

- To achieve high bandwidth, **shared memory is divided into** equally-sized memory modules, called **banks**
- Any memory read or write request made of *n* addresses that fall in *n* distinct memory banks can be serviced simultaneously
  - 1 transaction
- Otherwise:
  - serialized accesses
  - except if threads access the same address in a bank => broadcast
- Memory **requests** are **made by warp**, so bank conflicts can appears only inside a warp.
- For all CUDA architecture >= 2.x , the number of shared memory banks is 32

Banks

```
__global__ void myKernel_1D(...){

  __shared__ int myshared[1024];

  myshared[threadIdx.x] = threadIdx.x +10;

}
```

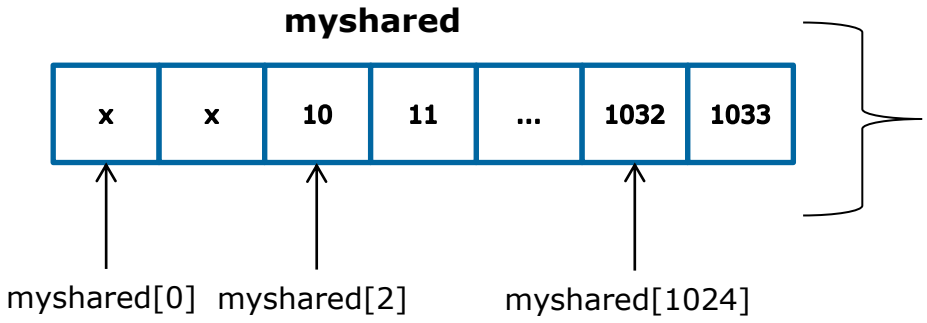


banks

0	1	2	3	...	29	30	31
10	11	12	13	...	39	40	41
42	43	44	45	...	71	72	73
74	75	76	77	...	103	104	105
...	...	...	...	...	...	...	...
1002	1003	1004	1005	...	1031	1032	1033

Banks

```
__global__ void myKernel_1D(...){

  __shared__ int myshared[1024+2];

  myshared [threadIdx.x + 2] = threadIdx.x +10;

}
```

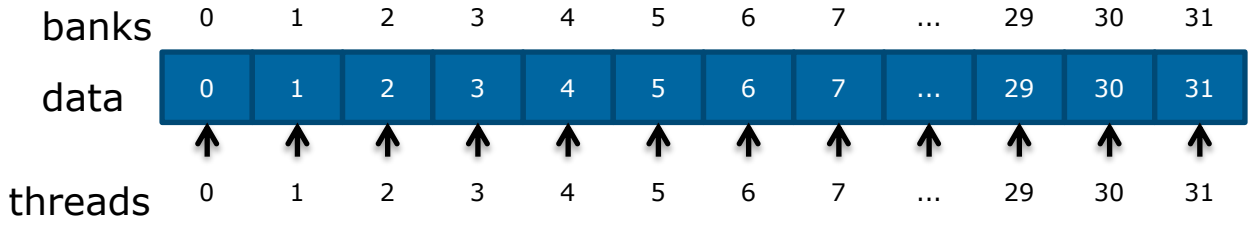


banks

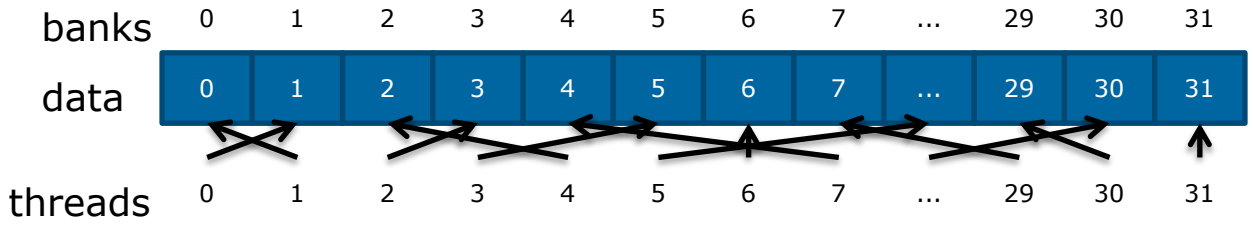
0	1	2	3	...	29	30	31
X	X	10	11	...	37	38	39
40	41	42	43	...	69	70	71
72	73	74	75	...	101	102	103
...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...
1032	1033	X	X	X	X	X	X

Conflict & Conflict-Free Accesses

- All threads within the warp access different banks:
  - regular accesses: conflict free



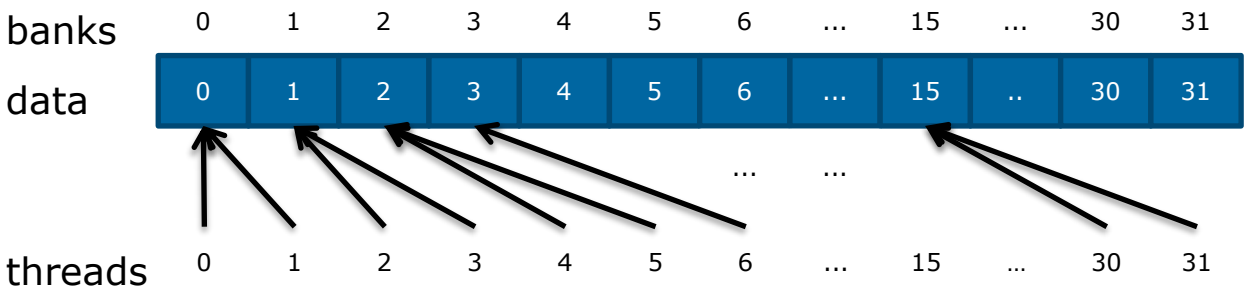
  - permutation within a warp: conflict free



Conflict & Conflict-Free Accesses

- Threads within the warp access several time a same bank:
  - same addresses accessed by bank: conflict free (broadcast)

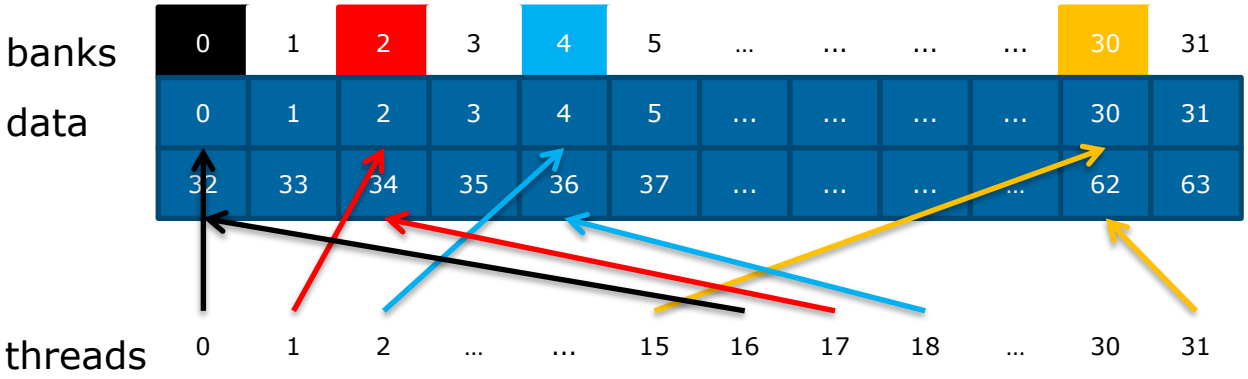
```
__global__ void myKernel_1D(...){

  __shared__ int myshared[1024];
  ...
  int temp = myshared[threadIdx.x /2];

}
```



Conflict & Conflict-Free Accesses

- Threads within the warp access several time a same bank:
  - different addresses accessed by bank: **conflicts**

```
__global__ void myKernel_1D(...){

  __shared__ int myshared[32*2];
  ...
  int temp = myshared[threadIdx.x * 2];   //2-way conflict

}
```



Shared Memory and Architecture

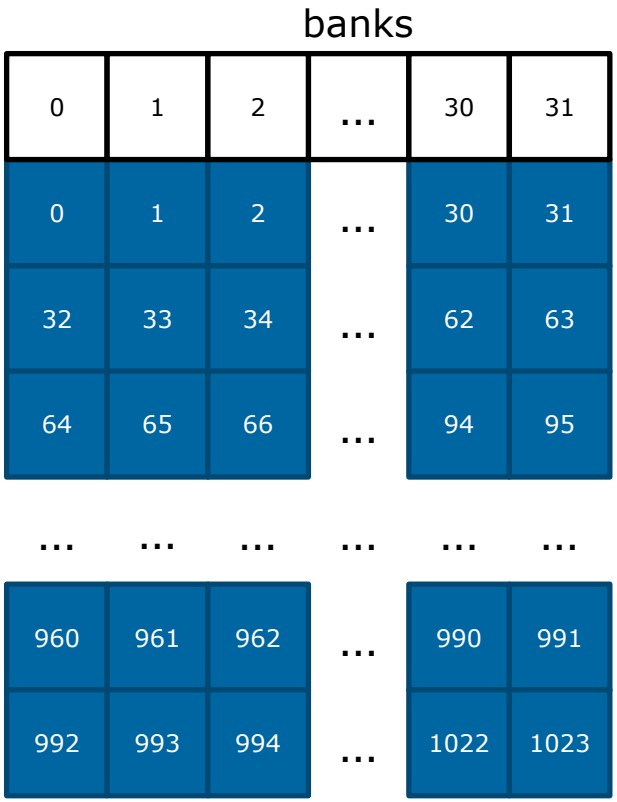
- 2.x (Fermi):
  - Shared memory has **32 banks** that are organized such that successive **32-bit words** map to successive banks
  - Each bank has a bandwidth of **32 bits per two clock cycles**
- 3.x (Kepler):
  - Shared memory has **32 banks** with two user-selectable addressing modes : 4 or 8 bytes bank width (32-bits or 64-bits modes)
  - Each bank has a bandwidth of **64 bits per clock cycle**
- 5.x, 6.x, 7.x (Maxwell, Pascal, Volta):
  - Shared memory has **32 banks** that are organized such that successive **32-bit words** map to successive banks
  - Each bank has a bandwidth of **32 bits per clock cycle**

LAB: Bank Access Analysis

- Complete bank.cu

Bank Access Analysis

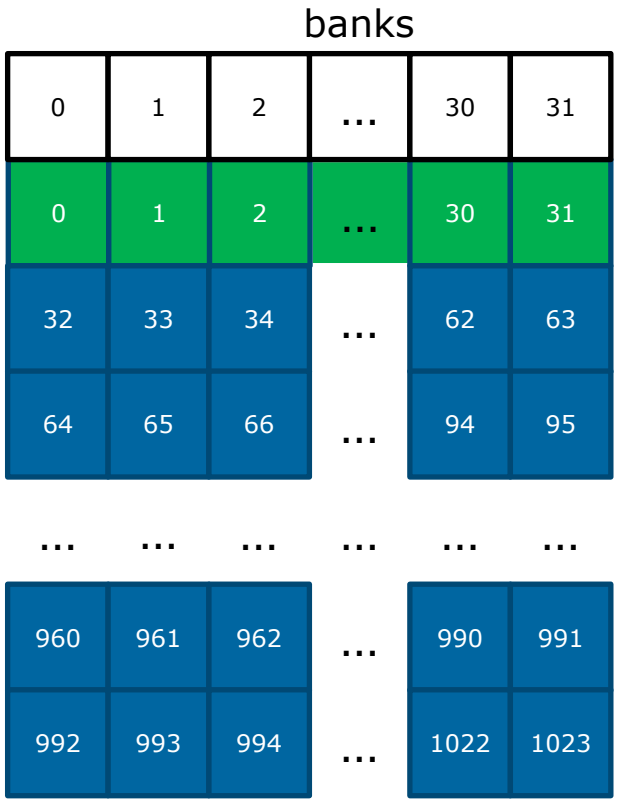
```
__global__ void transpose_shared(int size, int *in, int *out){

 __shared__ int shmem[32][32];

 int tx = threadIdx.x;
 int ty = threadIdx.y;

 int idx_in =  tx + blockIdx.x * blockDim.x;
 int idy_in =  ty + blockIdx.y * blockDim.y;

 int idx_out =  tx + blockIdx.y * blockDim.y;
 int idy_out =  ty + blockIdx.x * blockDim.x;

 shmem[ty][tx] = in[idx_in + idy_in * size];

 __syncthreads();

 out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



Bank Access Analysis

```
__global__ void transpose_shared(int size, int *in, int *out){

 __shared__ int shmem[32][32];

 int tx = threadIdx.x;
 int ty = threadIdx.y;

 int idx_in =  tx + blockIdx.x * blockDim.x;
 int idy_in =  ty + blockIdx.y * blockDim.y;

 int idx_out =  tx + blockIdx.y * blockDim.y;
 int idy_out =  ty + blockIdx.x * blockDim.x;

 shmem[ty][tx] = in[idx_in + idy_in * size];

 __syncthreads();

 out[idx_out + idy_out *size] = shmem[tx][ty];

}
```

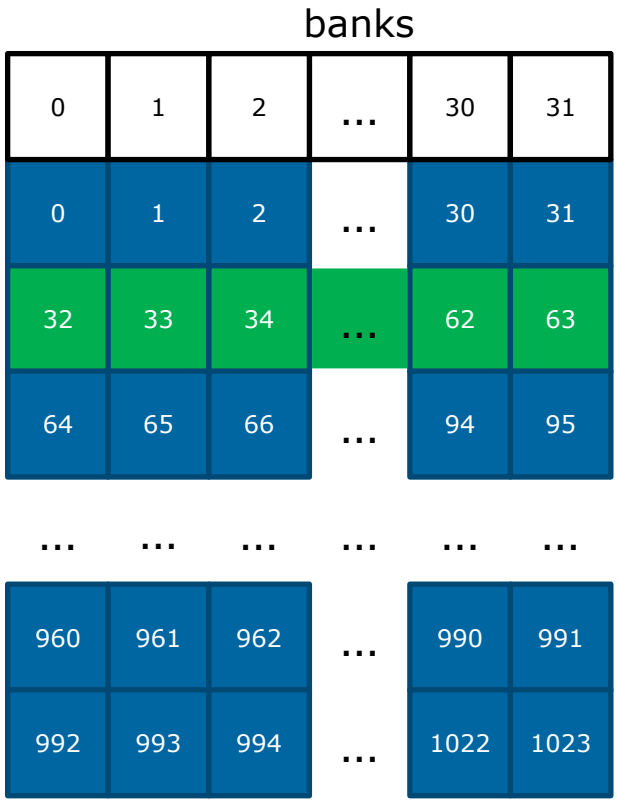
warp 1



Bank Access Analysis

```
__global__ void transpose_shared(int size, int *in, int *out){

 __shared__ int shmem[32][32];

 int tx = threadIdx.x;
 int ty = threadIdx.y;

 int idx_in =  tx + blockIdx.x * blockDim.x;
 int idy_in =  ty + blockIdx.y * blockDim.y;

 int idx_out =  tx + blockIdx.y * blockDim.y;
 int idy_out =  ty + blockIdx.x * blockDim.x;

 shmem[ty][tx] = in[idx_in + idy_in * size];

 __syncthreads();

 out[idx_out + idy_out *size] = shmem[tx][ty];

}
```

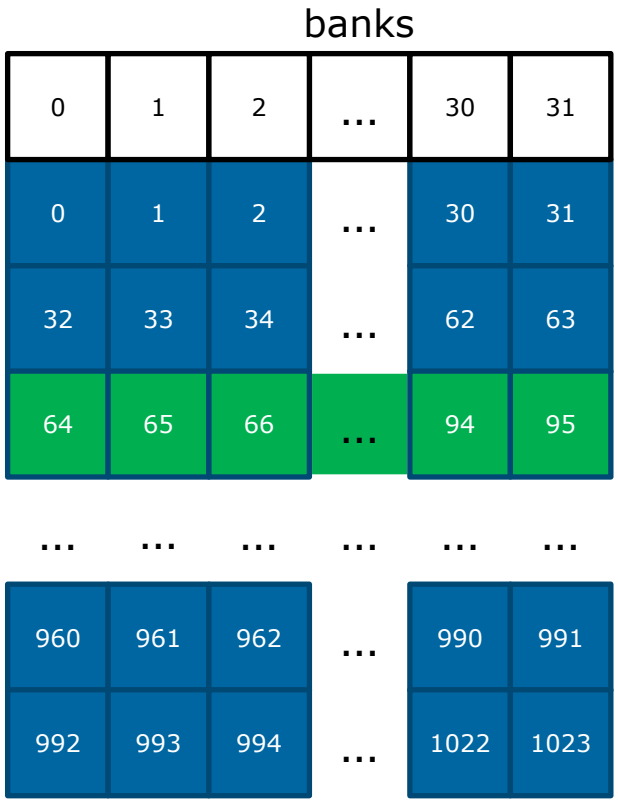
warp 2



Bank Access Analysis

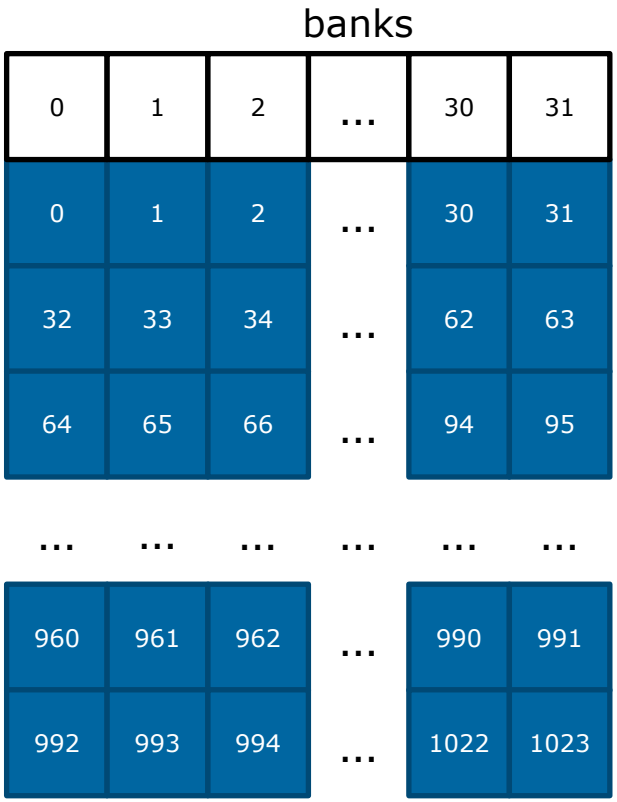
```
__global__ void transpose_shared(int size, int *in, int *out){

 __shared__ int shmem[32][32];

 int tx = threadIdx.x;
 int ty = threadIdx.y;

 int idx_in =  tx + blockIdx.x * blockDim.x;
 int idy_in =  ty + blockIdx.y * blockDim.y;

 int idx_out =  tx + blockIdx.y * blockDim.y;
 int idy_out =  ty + blockIdx.x * blockDim.x;

 shmem[ty][tx] = in[idx_in + idy_in * size];

 __syncthreads();

 out[idx_out + idy_out *size] = shmem[tx][ty];

}
```

warp 3



Bank Access Analysis

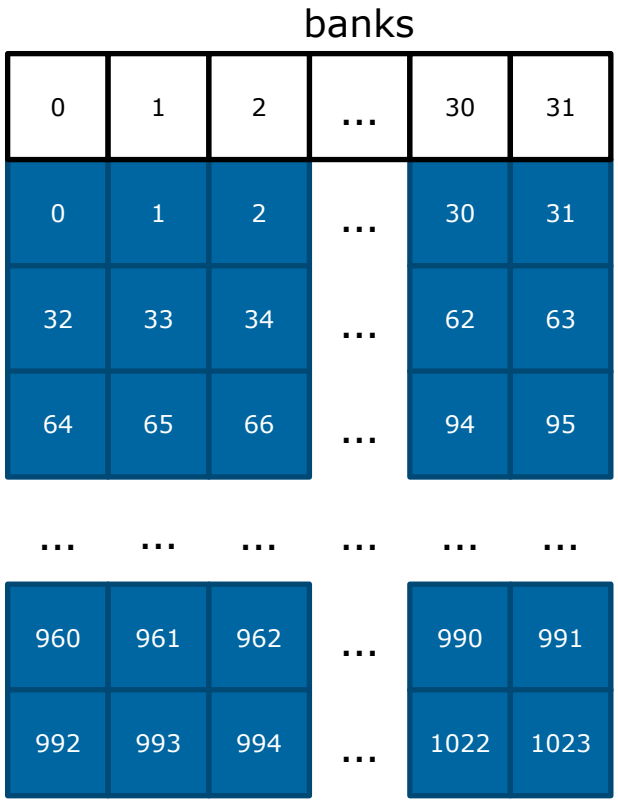
```
__global__ void transpose_shared(int size, int *in, int *out){

 __shared__ int shmem[32][32];

 int tx = threadIdx.x;
 int ty = threadIdx.y;

 int idx_in =  tx + blockIdx.x * blockDim.x;
 int idy_in =  ty + blockIdx.y * blockDim.y;

 int idx_out =  tx + blockIdx.y * blockDim.y;
 int idy_out =  ty + blockIdx.x * blockDim.x;

 shmem[ty][tx] = in[idx_in + idy_in * size];

 __syncthreads();

 out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



Bank Access Analysis

```
__global__ void transpose_shared(int size, int *in, int *out){

 __shared__ int shmem[32][32];

 int tx = threadIdx.x;
 int ty = threadIdx.y;

 int idx_in =  tx + blockIdx.x * blockDim.x;
 int idy_in =  ty + blockIdx.y * blockDim.y;

 int idx_out =  tx + blockIdx.y * blockDim.y;
 int idy_out =  ty + blockIdx.x * blockDim.x;

 shmem[ty][tx] = in[idx_in + idy_in * size];

 __syncthreads();

 out[idx_out + idy_out *size] = shmem[tx][ty];

}
```

warp 1



Bank Access Analysis

```
__global__ void transpose_shared(int size, int *in, int *out){

 __shared__ int shmem[32][32];

 int tx = threadIdx.x;
 int ty = threadIdx.y;

 int idx_in =  tx + blockIdx.x * blockDim.x;
 int idy_in =  ty + blockIdx.y * blockDim.y;

 int idx_out =  tx + blockIdx.y * blockDim.y;
 int idy_out =  ty + blockIdx.x * blockDim.x;

 shmem[ty][tx] = in[idx_in + idy_in * size];

 __syncthreads();

 out[idx_out + idy_out *size] = shmem[tx][ty];

}
```

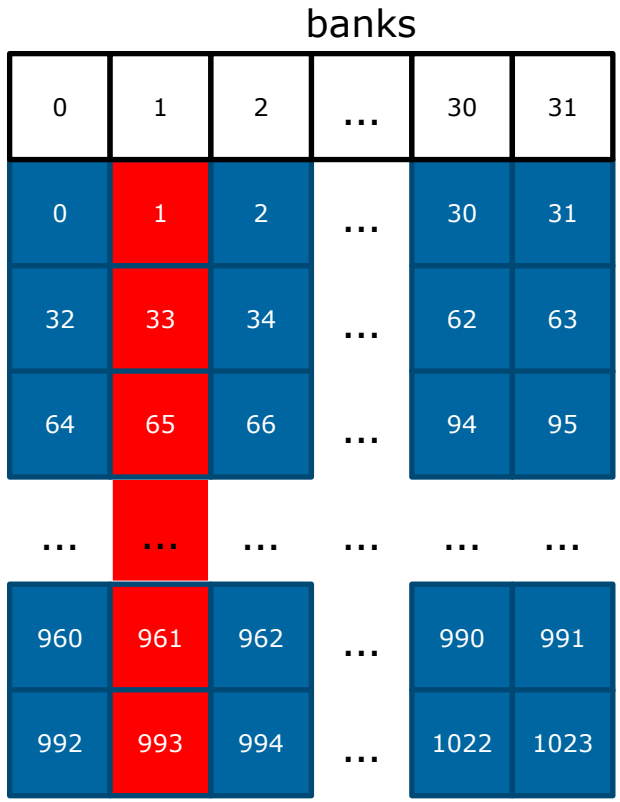
warp 1

banks

	0	1	2	...	30	31
0	0	1	2	...	30	31
32	32	33	34	...	62	63
64	64	65	66	...	94	95
...	...	...	...	...	...	...
960	960	961	962	...	990	991
992	992	993	994	...	1022	1023

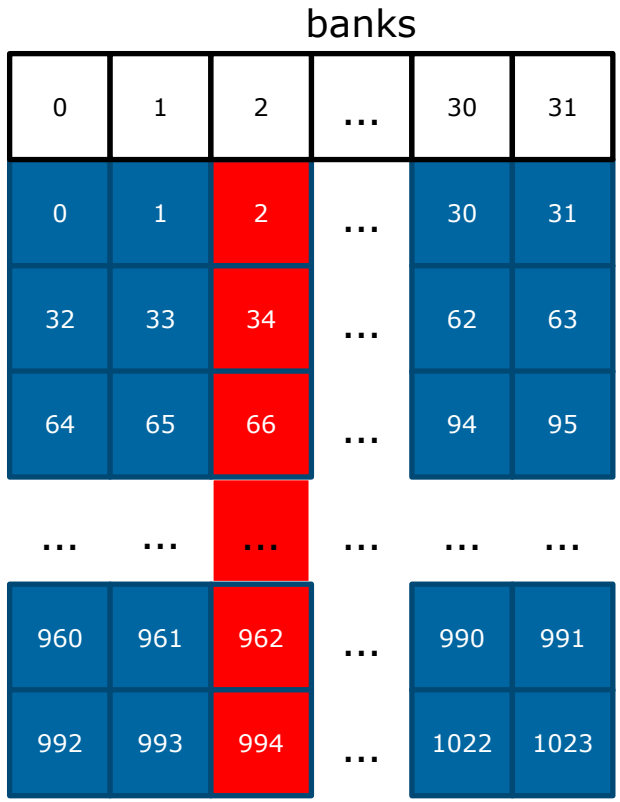
Bank Access Analysis

```
__global__ void transpose_shared(int size, int *in, int *out){

 __shared__ int shmem[32][32];

 int tx = threadIdx.x;
 int ty = threadIdx.y;

 int idx_in =  tx + blockIdx.x * blockDim.x;
 int idy_in =  ty + blockIdx.y * blockDim.y;

 int idx_out =  tx + blockIdx.y * blockDim.y;
 int idy_out =  ty + blockIdx.x * blockDim.x;

 shmem[ty][tx] = in[idx_in + idy_in * size];

 __syncthreads();

 out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



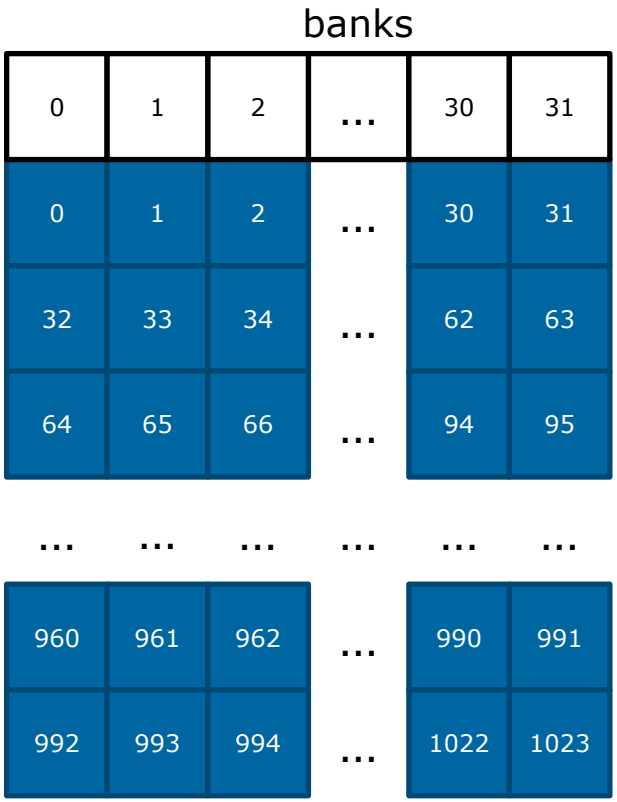
Bank Access Analysis

```
__global__ void transpose_shared(int size, int *in, int *out){

 __shared__ int shmem[32][32];

 int tx = threadIdx.x;
 int ty = threadIdx.y;

 int idx_in =  tx + blockIdx.x * blockDim.x;
 int idy_in =  ty + blockIdx.y * blockDim.y;

 int idx_out =  tx + blockIdx.y * blockDim.y;
 int idy_out =  ty + blockIdx.x * blockDim.x;

 shmem[ty][tx] = in[idx_in + idy_in * size];

 __syncthreads();

 out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



Bank Access Analysis

```
__global__ void transpose_shared(int size, int *in, int *out){

 __shared__ int shmem[32][32];

 int tx = threadIdx.x;
 int ty = threadIdx.y;

 int idx_in =  tx + blockIdx.x * blockDim.x;
 int idy_in =  ty + blockIdx.y * blockDim.y;

 int idx_out =  tx + blockIdx.y * blockDim.y;
 int idy_out =  ty + blockIdx.x * blockDim.x;

 shmem[ty][tx] = in[idx_in + idy_in * size];

 __syncthreads();

 out[idx_out + idy_out *size] = shmem[tx][ty];

}
```

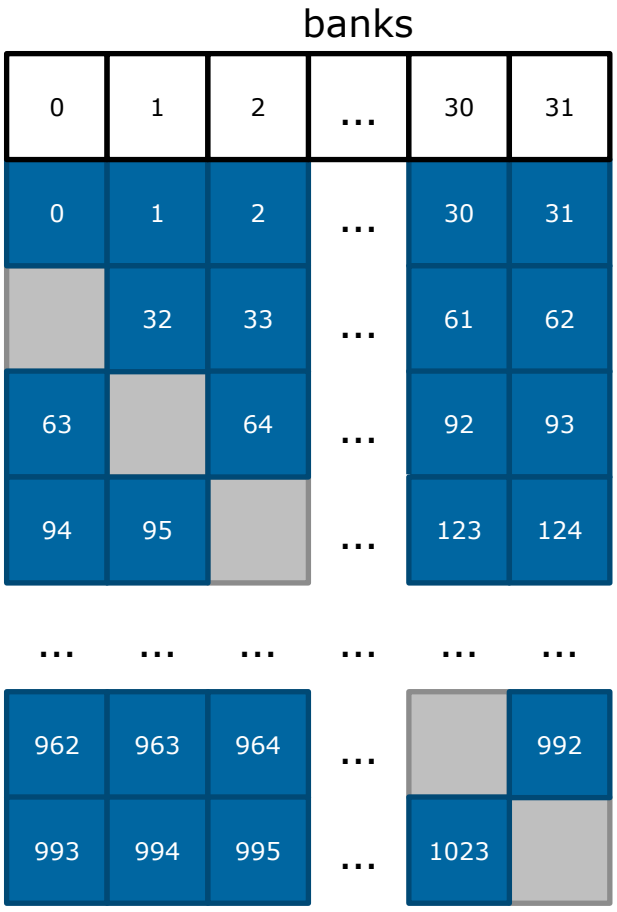


4th Step Full Transposition Via Shared Memory

- Avoid the bank conflicts when reading shared memory in **bank.cu**
- Only 1 numeral to modify
- Optimize the transposition via Shared Memory

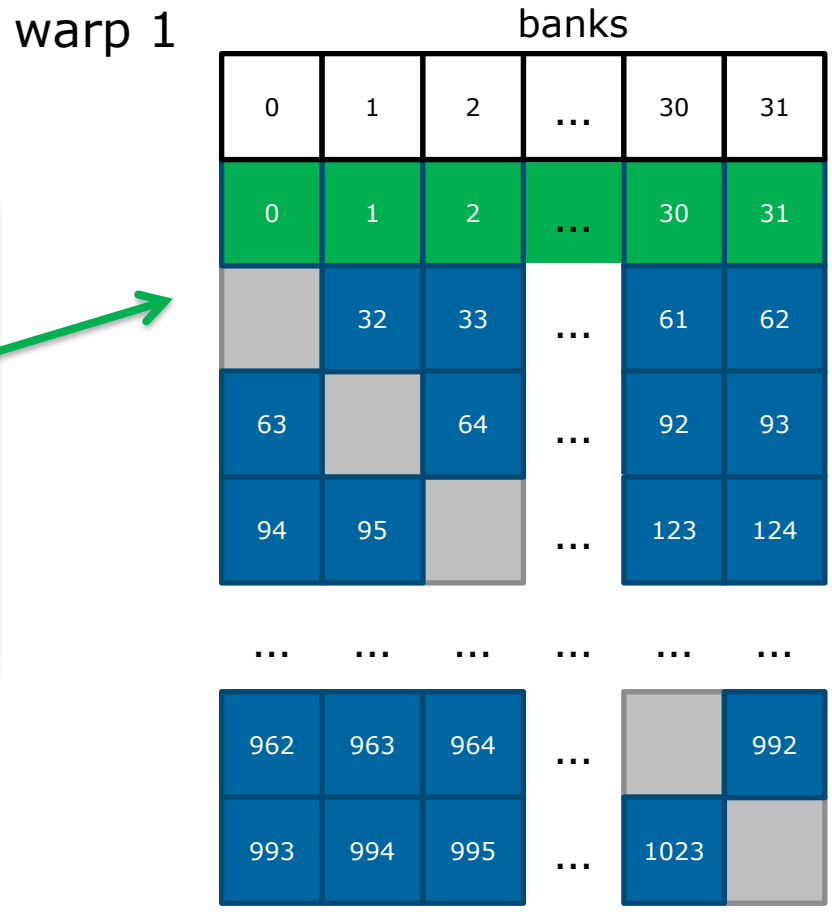
Solving Bank Conflicts

```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][33];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



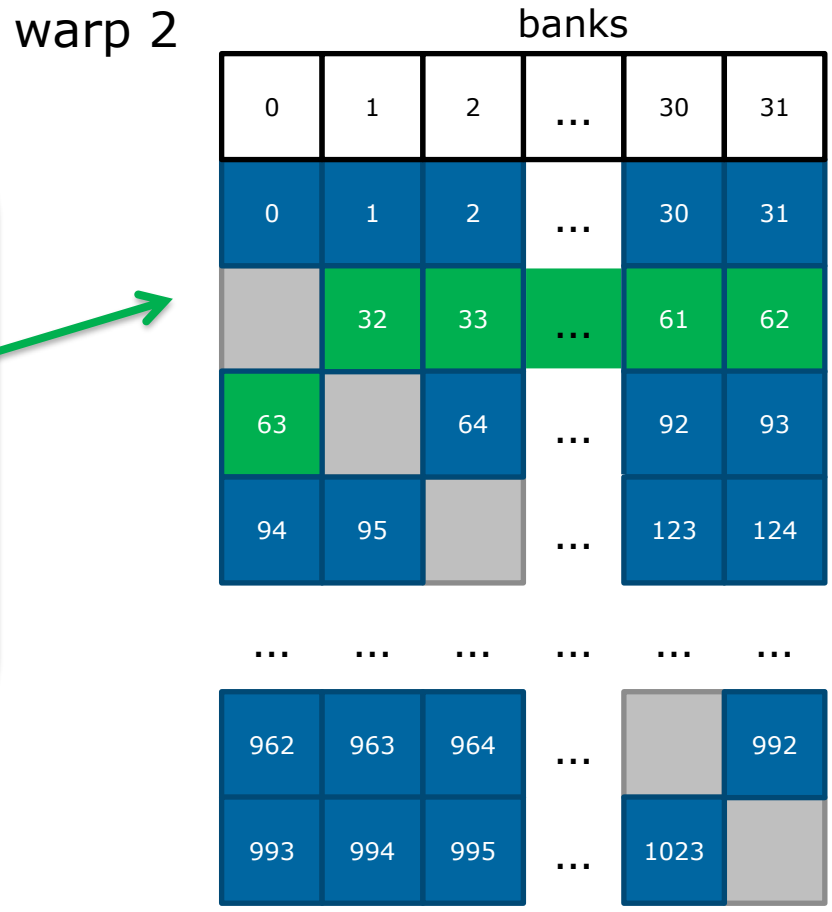
Solving Bank Conflicts

```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][33];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



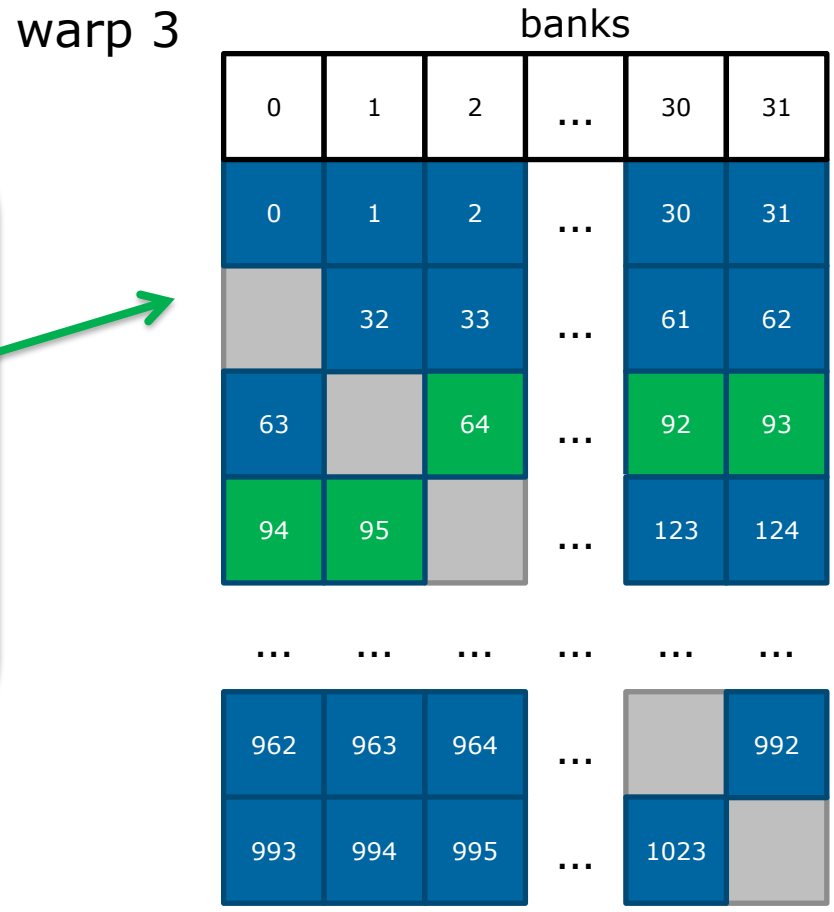
Solving Bank Conflicts

```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][33];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



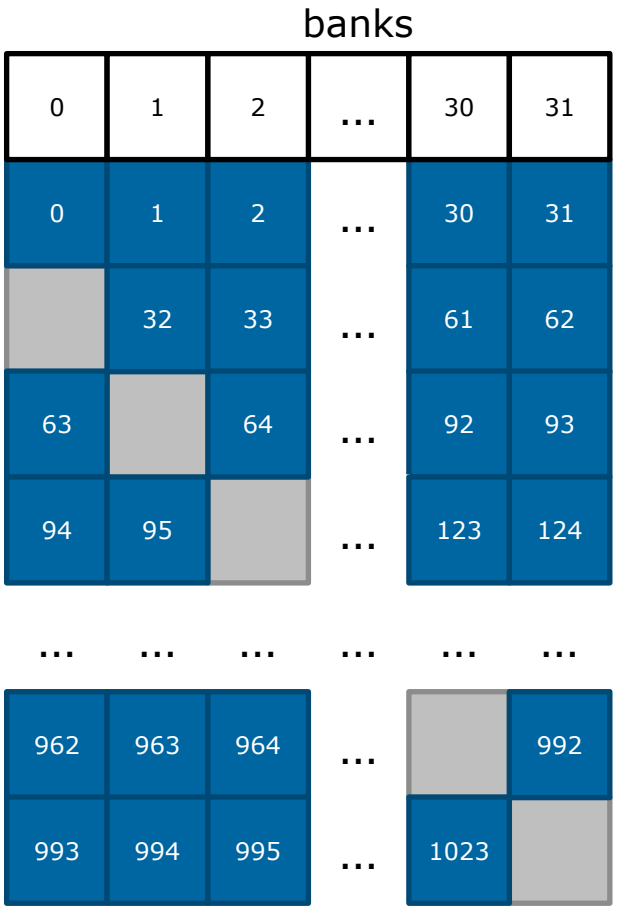
Solving Bank Conflicts

```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][33];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



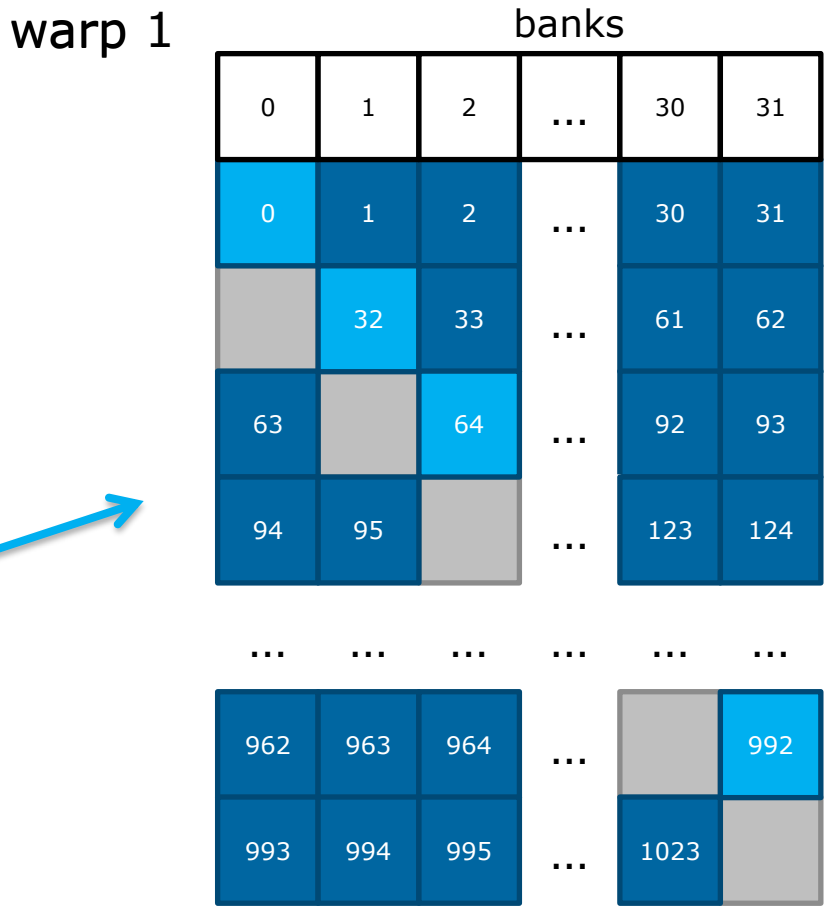
Solving Bank Conflicts

```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][33];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



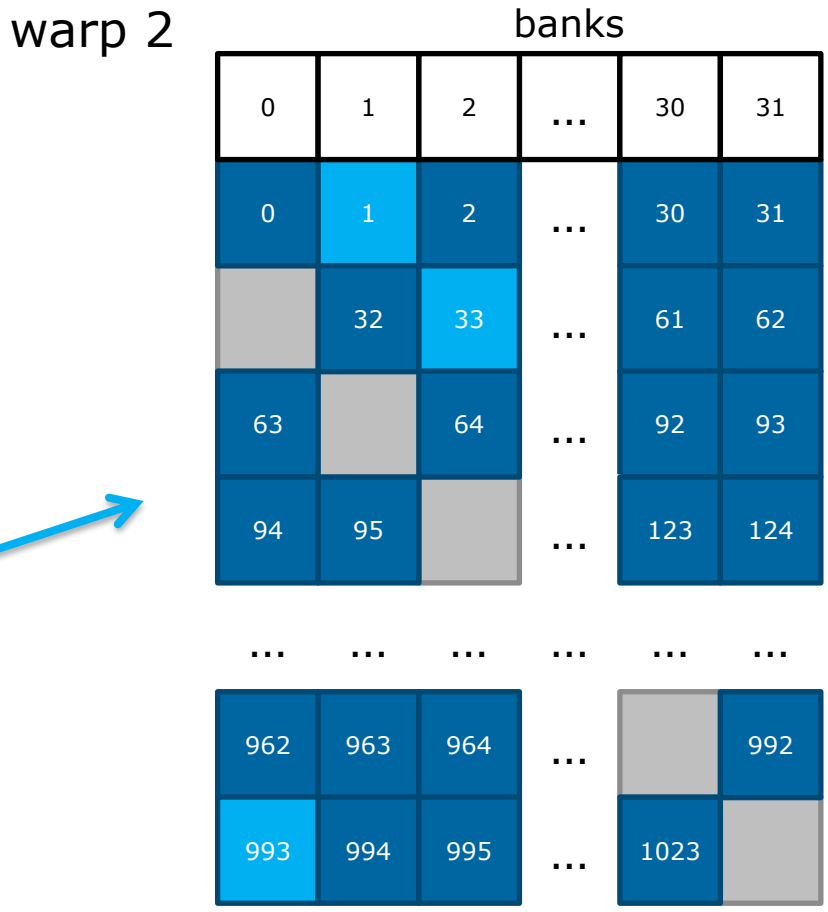
Solving Bank Conflicts

```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][33];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



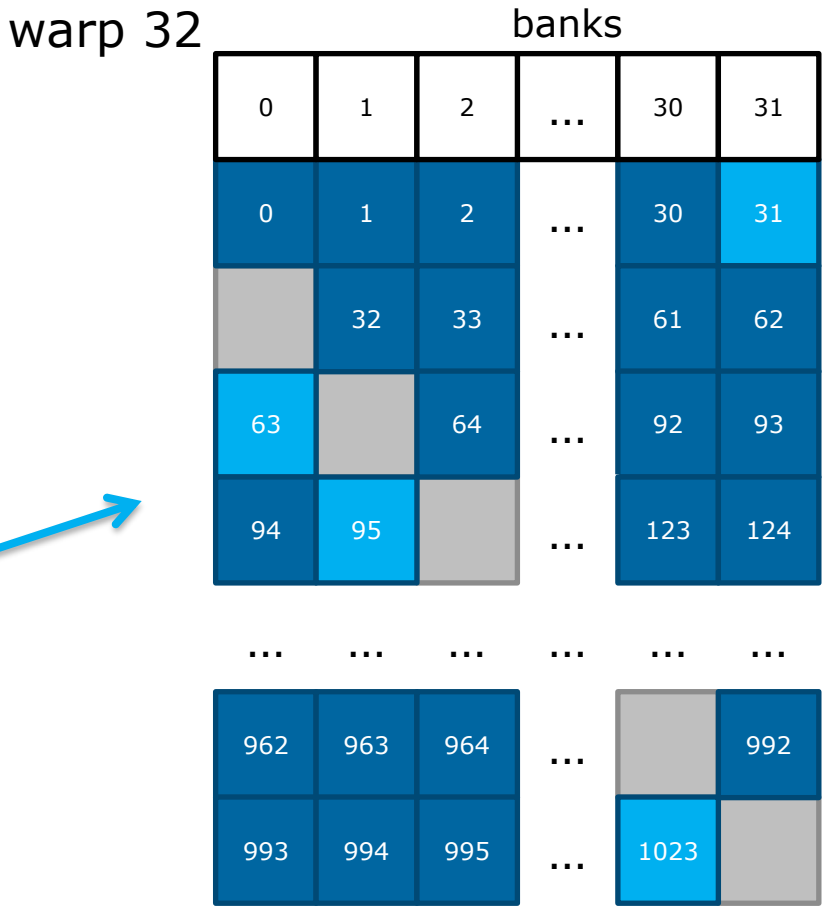
Solving Bank Conflicts

```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][33];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



Solving Bank Conflicts

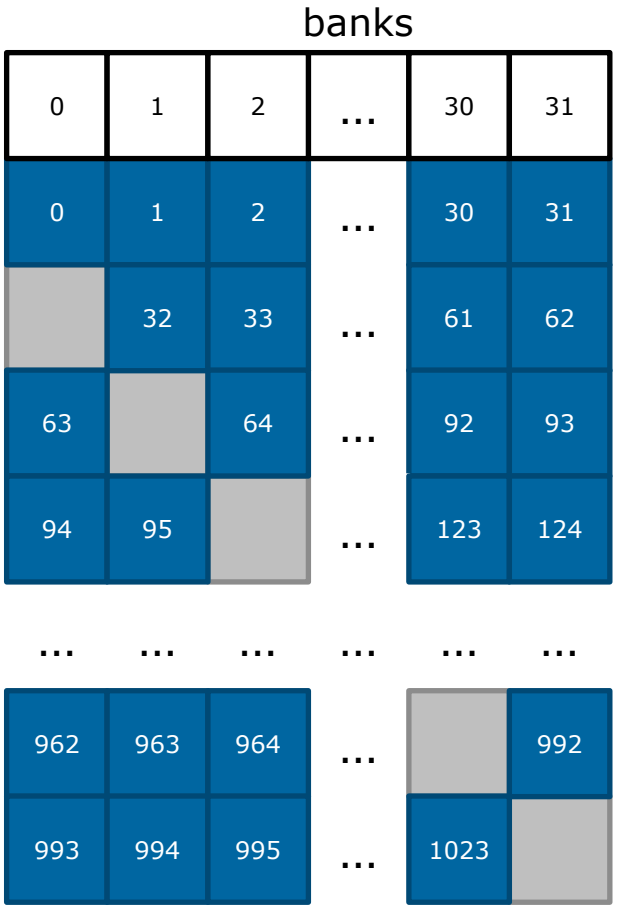
```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][33];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];

}
```



Solving Bank Conflicts

```
__global__ void transpose_shared(int size, int *in, int *out){

  __shared__ int shmem[32][33];

  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int idx_in =  tx + blockIdx.x * blockDim.x;
  int idy_in =  ty + blockIdx.y * blockDim.y;

  int idx_out =  tx + blockIdx.y * blockDim.y;
  int idy_out =  ty + blockIdx.x * blockDim.x;

  shmem[ty][tx] = in[idx_in + idy_in * size];

  __syncthreads();

  out[idx_out + idy_out *size] = shmem[tx][ty];

}
```

- **1317.024 µs** for a 4096 *4096 matrix on a K80



Performance Summary

Kernel	Time in µs
Copy (no transposition)	1108.928
Transposition in global	3009.024
Transposition in shared	2393.984
Transposition in shared without bank conflicts	1317.024

Minimize Global Memory Accesses

05/08/2020

Minimize Global Memory Accesses

- Device memory is the slowest memory on the GPU
- Reducing global memory accesses inside a kernel implies better performances
- Be careful with data reuse:
  - on Kepler global memory accesses are by default cached only in L2 (not L1)
- Keep data in:
  - register: if the current thread will reuse the data
  - shared memory: if another thread from the same block will also use the data
    - as in the transpose example

Software Pipelining Example

- We will port and optimize the following loopnest on GPU:
  - using a 2D grid (for iterations on i and j)
  - each thread will execute the k loop

```
#define   in(i,j,k)   in[(i)+(j)*size+ (k) *size*size)
#define out(i,j,k) out[(i)+(j)*size+ (k) *size*size)

for(int k=4; k<(size-4); k++){
 for(int j=0; j<size; j++){
   for(int i=0; i<size; i++){

       out(i,j,k) = in(i,j,k) + in(i,j,k+1) - in(i,j,k-1)
                              + in(i,j,k+2) - in(i,j,k-2)
                              + in(i,j,k+3) - in(i,j,k-3)
                              + in(i,j,k+4) - in(i,j,k-4);
  }
 }
}
```

Software Pipelining Example

- Each thread (i,j) performs out(i,j,k) = in(i,j,k) + in(i,j,k+1) - in(i,j,k-1)
  + in(i,j,k+2) - in(i,j,k-2)
  + in(i,j,k+3) - in(i,j,k-3)
  + in(i,j,k+4) - in(i,j,k-4)

```
__global__ void myKernel_ref(int size_x, int size_y, int size_z, int* in, int* out){

  int idx = threadIdx.x + blockIdx.x * blockDim.x;  //i
  int idy = threadIdx.y + blockIdx.y * blockDim.y;  //j
  int idz = 0;  //k

  int temp;

  for(idz = 4; idz < (size_z-4); idz++){

    temp =  in[idx + idy *size_x + (idz+0) * size_x * size_y]
         +    in[idx + idy *size_x + (idz+1) * size_x * size_y] - in[idx + idy *size_x + (idz-1) * size_x * size_y]
         +    in[idx + idy *size_x + (idz+2) * size_x * size_y] - in[idx + idy *size_x + (idz-2) * size_x * size_y]
         +    in[idx + idy *size_x + (idz+3) * size_x * size_y] - in[idx + idy *size_x + (idz-3) * size_x * size_y]
         +    in[idx + idy *size_x + (idz+4) * size_x * size_y] - in[idx + idy *size_x + (idz-4) * size_x * size_y];

    if( (idx<size_x) && (idy<size_y) ){
      out[idx + idy *size_x + idz * size_x * size_y] = temp;
    }

  }
}
```

9 reads in the *in* array by iteration for a (i,j) thread

=> (size_z – 8) * 9 read by threads

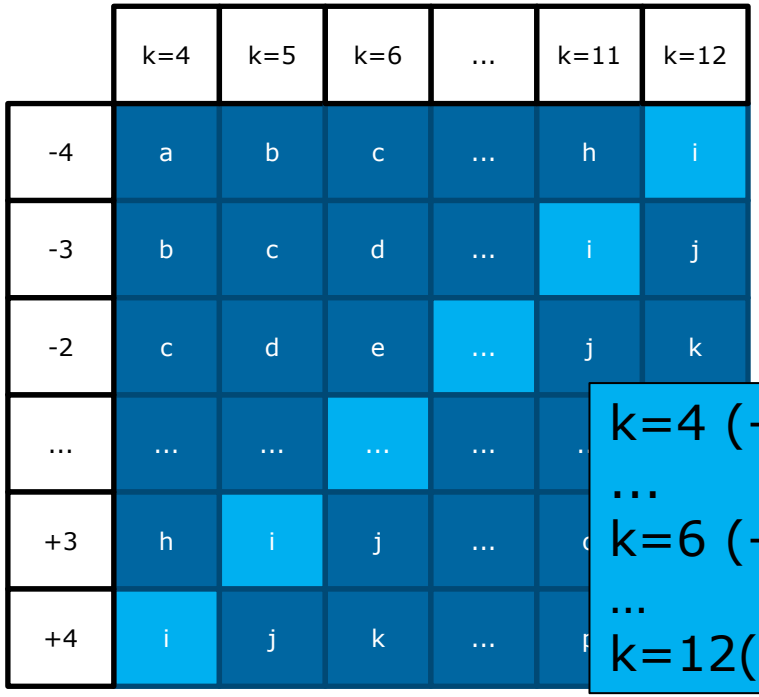
Software Pipelining Example

- For a same (i,j) thread, successive computations of k accesses common data
- *in(i,j,k+4)* for the *kth* iteration is the same memory access as:
  - *in(i,j,k+3)* for the *(k+1)* iteration
  - *in(i,j,k+2)* for the *(k+2)* iteration
  - ...
  - *in(i,j,k-4)* for the *(k+8)* iteration
- Two consecutive k iterations share 7 data
- Keep data in register to avoid to re-read a data from global memory

	k=4	k=5	k=6	...	k=11	k=12
-4	a	b	c	...	h	i
-3	b	c	d	...	i	j
-2	c	d	e	...	j	k
...	...	...	...	...	...	...
+3	h	i	j	...	o	p
+4	i	j	k	...	p	q

Software Pipelining Example

- For a same (i,j) thread, successive computations of k accesses common data
- *in(i,j,k+4)* for the *kth* iteration is the same memory access as:
  - *in(i,j,k+3)* for the *(k+1)* iteration
  - *in(i,j,k+2)* for the *(k+2)* iteration
  - ...
  - *in(i,j,k-4)* for the *(k+8)* iteration
- Two consecutive k iterations share 7 data
- Keep data in register to avoid to re-read a data from global memory



Software Pipelining Example

- For a same (i,j) thread, successive computations of k accesses common data
- *in(i,j,k+4)* for the *kth* iteration is the same memory access as:
  - *in(i,j,k+3)* for the *(k+1)* iteration
  - *in(i,j,k+2)* for the *(k+2)* iteration
  - ...
  - *in(i,j,k-4)* for the *(k+8)* iteration
- Two consecutive k iterations share 7 data
- Keep data in register to avoid to re-read a data from global memory

	k=4	k=5	k=6	...	k=11	k=12
-4	a	b	c	...	h	i
-3	b	c	d	...	i	j
-2	c	d	e	...	j	k
...	...	...	...	...	...	...
+3	h	i	j	...	o	p
+4	i	j	k	...	p	q

Software Pipelining Example

- For a same (i,j) thread, successive computations of k accesses common data
- *in(i,j,k+4)* for the *kth* iteration is the same memory access as:
  - *in(i,j,k+3)* for the *(k+1)* iteration
  - *in(i,j,k+2)* for the *(k+2)* iteration
  - ...
  - *in(i,j,k-4)* for the *(k+8)* iteration
- Two consecutive k iterations share 7 data
- Keep data in register to avoid to re-read a data from global memory

	k=4	k=5	k=6	...	k=11	k=12
-4	a	b	c	...	h	i
-3	b	c	d	...	i	j
-2	c	d	e	...	j	k
...	...	...	...	...	...	...
+3	h	i	j	...	o	p
+4	i	j	k	...	p	q

LAB: Software Pipelining

```
__global__ void myKernel_pipeline(int size_x, int size_y, int size_z, const int* __restrict in, int* __restrict__ out){

  int idx = threadIdx.x + blockIdx.x * blockDim.x;    int idy = threadIdx.y + blockIdx.y * blockDim.y;   int idz = 0;

  int in_m4 =
  int in_m3 =
  int in_m2 =
  ...
  int in_p3 =
  int in_p4 =

  for(idz = 4; idz < (size_z-4); idz++){

    in_m4 =
    in_m3 =
    in_m2 =
    ...
    in_p3 =
    in_p4 =

    int temp = in_cu      +
                          +
                          +
                          +

    if( (idx<size_x) && (idy<size_y) ){
      out[idx + idy *size_x + idz * size_x * size_y] = temp;
    }

  }
}
```

preload data:
8 reads

shift data

read missing data:
1 read

Software Pipelining Example

```
__global__ void myKernel_pipeline(int size_x, int size_y, int size_z, const int* __restrict in, int* __restrict__ out){

 int idx = threadIdx.x + blockIdx.x * blockDim.x;    int idy = threadIdx.y + blockIdx.y * blockDim.y;   int idz = 0;

 int in_m4 = 0;
 int in_m3 = in[idx + idy *size_x + 0 * size_x * size_y];     //in[idx + idy *size_x + (idz-4) * size_x * size_y]; with idz = 4
 int in_m2 = in[idx + idy *size_x + 1 * size_x * size_y];     //in[idx + idy *size_x + (idz-3) * size_x * size_y]; with idz = 4
 ...
 int in_p3 = in[idx + idy *size_x + 6 * size_x * size_y];     //in[idx + idy *size_x + (idz+2) * size_x * size_y]; with idz = 4
 int in_p4 = in[idx + idy *size_x + 7 * size_x * size_y];     //in[idx + idy *size_x + (idz+3) * size_x * size_y]; with idz = 4

 for(idz = 4; idz < (size_z-4); idz++){

  in_m4 = in_m3;
  in_m3 = in_m2;
  in_m2 = in_m1;
  ...
  in_p3 = in_p4;
  in_p4 = in[idx + idy *size_x + (idz+4) * size_x * size_y];

  int temp = in_cu      +    in_p1 - in_m1
                        +    in_p2 - in_m2
                        +    in_p3 - in_m3
                        +    in_p4 - in_m4;

  if( (idx<size_x) && (idy<size_y) ){
    out[idx + idy *size_x + idz * size_x * size_y] = temp;
  }

 }
}
```

Software Pipelining Example

```
__global__ void myKernel_pipeline(int size_x, int size_y, int size_z, const int* __restrict in, int* __restrict__ out){

 int idx = threadIdx.x + blockIdx.x * blockDim.x;    int idy = threadIdx.y + blockIdx.y * blockDim.y;   int idz = 0;

 int in_m4 = 0;
 int in_m3 = in[idx + idy *size_x + 0 * size_x * size_y];     //in[idx + idy *size_x + (idz-4) * size_x * size_y]; with idz = 4
 int in_m2 = in[idx + idy *size_x + 1 * size_x * size_y];     //in[idx + idy *size_x + (idz-3) * size_x * size_y]; with idz = 4
 ...
 int in_p3 = in[idx + idy *size_x + 6 * size_x * size_y];     //in[idx + idy *size_x + (idz+2) * size_x * size_y]; with idz = 4
 int in_p4 = in[idx + idy *size_x + 7 * size_x * size_y];     //in[idx + idy *size_x + (idz+3) * size_x * size_y]; with idz = 4

 for(idz = 4; idz < (size_z-4); idz++){

  in_m4 = in_m3;
  in_m3 = in_m2;
  in_m2 = in_m1;
  ...
  in_p3 = in_p4;
  in_p4 = in[idx + idy *size_x + (idz+4) * size_x * size_y];

  int temp = in_cu      +    in_p1 - in_m1
                        +    in_p2 - in_m2
                        +    in_p3 - in_m3
                        +    in_p4 - in_m4;

  if( (idx<size_x) && (idy<size_y) ){
    out[idx + idy *size_x + idz * size_x * size_y] = temp;
  }

 }
}
```

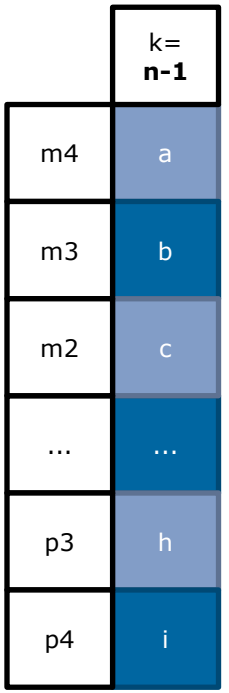
preload data:
8 reads

Software Pipelining Example

```
__global__ void myKernel_pipeline(int size_x, int size_y, int size_z, const int* __restrict in, int* __restrict__ out){

 int idx = threadIdx.x + blockIdx.x * blockDim.x;    int idy = threadIdx.y + blockIdx.y * blockDim.y;   int idz = 0;

 int in_m4 = 0;
 int in_m3 = in[idx + idy *size_x + 0 * size_x * size_y];     //in[idx + idy *size_x + (idz-4) * size_x * size_y]; with idz = 4
 int in_m2 = in[idx + idy *size_x + 1 * size_x * size_y];     //in[idx + idy *size_x + (idz-3) * size_x * size_y]; with idz = 4
 ...
 int in_p3 = in[idx + idy *size_x + 6 * size_x * size_y];     //in[idx + idy *size_x + (idz+2) * size_x * size_y]; with idz = 4
 int in_p4 = in[idx + idy *size_x + 7 * size_x * size_y];     //in[idx + idy *size_x + (idz+3) * size_x * size_y]; with idz = 4

 for(idz = 4; idz < (size_z-4); idz++){

  in_m4 = in_m3;
  in_m3 = in_m2;
  in_m2 = in_m1;
  ...
  in_p3 = in_p4;
  in_p4 = in[idx + idy *size_x + (idz+4) * size_x * size_y];

  int temp = in_cu      +    in_p1 - in_m1
                        +    in_p2 - in_m2
                        +    in_p3 - in_m3
                        +    in_p4 - in_m4;

  if( (idx<size_x) && (idy<size_y) ){
    out[idx + idy *size_x + idz * size_x * size_y] = temp;
  }

 }
}
```

shift data

previous state

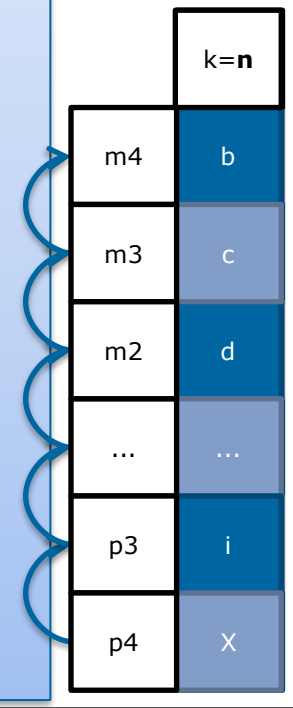


Software Pipelining Example

```
__global__ void myKernel_pipeline(int size_x, int size_y, int size_z, const int* __restrict in, int* __restrict__ out){

 int idx = threadIdx.x + blockIdx.x * blockDim.x;    int idy = threadIdx.y + blockIdx.y * blockDim.y;   int idz = 0;

 int in_m4 = 0;
 int in_m3 = in[idx + idy *size_x + 0 * size_x * size_y];     //in[idx + idy *size_x + (idz-4) * size_x * size_y]; with idz = 4
 int in_m2 = in[idx + idy *size_x + 1 * size_x * size_y];     //in[idx + idy *size_x + (idz-3) * size_x * size_y]; with idz = 4
 ...
 int in_p3 = in[idx + idy *size_x + 6 * size_x * size_y];     //in[idx + idy *size_x + (idz+2) * size_x * size_y]; with idz = 4
 int in_p4 = in[idx + idy *size_x + 7 * size_x * size_y];     //in[idx + idy *size_x + (idz+3) * size_x * size_y]; with idz = 4

 for(idz = 4; idz < (size_z-4); idz++){

  in_m4 = in_m3;
  in_m3 = in_m2;
  in_m2 = in_m1;
  ...
  in_p3 = in_p4;
  in_p4 = in[idx + idy *size_x + (idz+4) * size_x * size_y];

  int temp = in_cu     +   in_p1 - in_m1
                       +   in_p2 - in_m2
                       +   in_p3 - in_m3
                       +   in_p4 - in_m4;

  if( (idx<size_x) && (idy<size_y) ){
    out[idx + idy *size_x + idz * size_x * size_y] = temp;
  }

 }
}
```

shift data

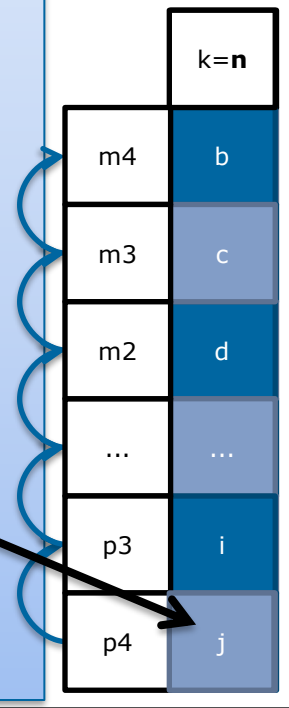
current state



Software Pipelining Example

```
__global__ void myKernel_pipeline(int size_x, int size_y, int size_z, const int* __restrict in, int* __restrict__ out){

 int idx = threadIdx.x + blockIdx.x * blockDim.x;    int idy = threadIdx.y + blockIdx.y * blockDim.y;   int idz = 0;

 int in_m4 = 0;
 int in_m3 = in[idx + idy *size_x + 0 * size_x * size_y];     //in[idx + idy *size_x + (idz-4) * size_x * size_y]; with idz = 4
 int in_m2 = in[idx + idy *size_x + 1 * size_x * size_y];     //in[idx + idy *size_x + (idz-3) * size_x * size_y]; with idz = 4
 ...
 int in_p3 = in[idx + idy *size_x + 6 * size_x * size_y];     //in[idx + idy *size_x + (idz+2) * size_x * size_y]; with idz = 4
 int in_p4 = in[idx + idy *size_x + 7 * size_x * size_y];     //in[idx + idy *size_x + (idz+3) * size_x * size_y]; with idz = 4

 for(idz = 4; idz < (size_z-4); idz++){

  in_m4 = in_m3;
  in_m3 = in_m2;
  in_m2 = in_m1;
  ...
  in_p3 = in_p4;
  in_p4 = in[idx + idy *size_x + (idz+4) * size_x * size_y];

  int temp = in_cu      +    in_p1 - in_m1
                        +    in_p2 - in_m2
                        +    in_p3 - in_m3
                        +    in_p4 - in_m4;

  if( (idx<size_x) && (idy<size_y) ){
    out[idx + idy *size_x + idz * size_x * size_y] = temp;
  }

 }
}
```

read missing data:
1 read



Software Pipelining Example

```
__global__ void myKernel_pipeline(int size_x, int size_y, int size_z, const int* __restrict in, int* __restrict__ out){

 int idx = threadIdx.x + blockIdx.x * blockDim.x;    int idy = threadIdx.y + blockIdx.y * blockDim.y;   int idz = 0;

 int in_m4 = 0;
 int in_m3 = in[idx + idy *size_x + 0 * size_x * size_y];     //in[idx + idy *size_x + (idz-4) * size_x * size_y]; with idz = 4
 int in_m2 = in[idx + idy *size_x + 1 * size_x * size_y];     //in[idx + idy *size_x + (idz-3) * size_x * size_y]; with idz = 4
 ...
 int in_p3 = in[idx + idy *size_x + 6 * size_x * size_y];     //in[idx + idy *size_x + (idz+2) * size_x * size_y]; with idz = 4
 int in_p4 = in[idx + idy *size_x + 7 * size_x * size_y];     //in[idx + idy *size_x + (idz+3) * size_x * size_y]; with idz = 4

 for(idz = 4; idz < (size_z-4); idz++){

  in_m4 = in_m3;
  in_m3 = in_m2;
  in_m2 = in_m1;
  ...
  in_p3 = in_p4;
  in_p4 = in[idx + idy *size_x + (idz+4) * size_x * size_y];

  int temp = in_cu     +   in_p1 - in_m1
                       +   in_p2 - in_m2
                       +   in_p3 - in_m3
                       +   in_p4 - in_m4;

  if( (idx<size_x) && (idy<size_y) ){
   out[idx + idy *size_x + idz * size_x * size_y] = temp;
  }

 }
}
```

preload data:
8 reads

shift data

read missing data:
1 read

(8 + (size_z – 8) * 1)
= size_z reads by thread

Software Pipelining Example

	Number of reads in global memory	
size_z	1st version 9*(size_z-8)	2nd version size_z
9	9	9
10	18	10
20	108	20
100	828	100

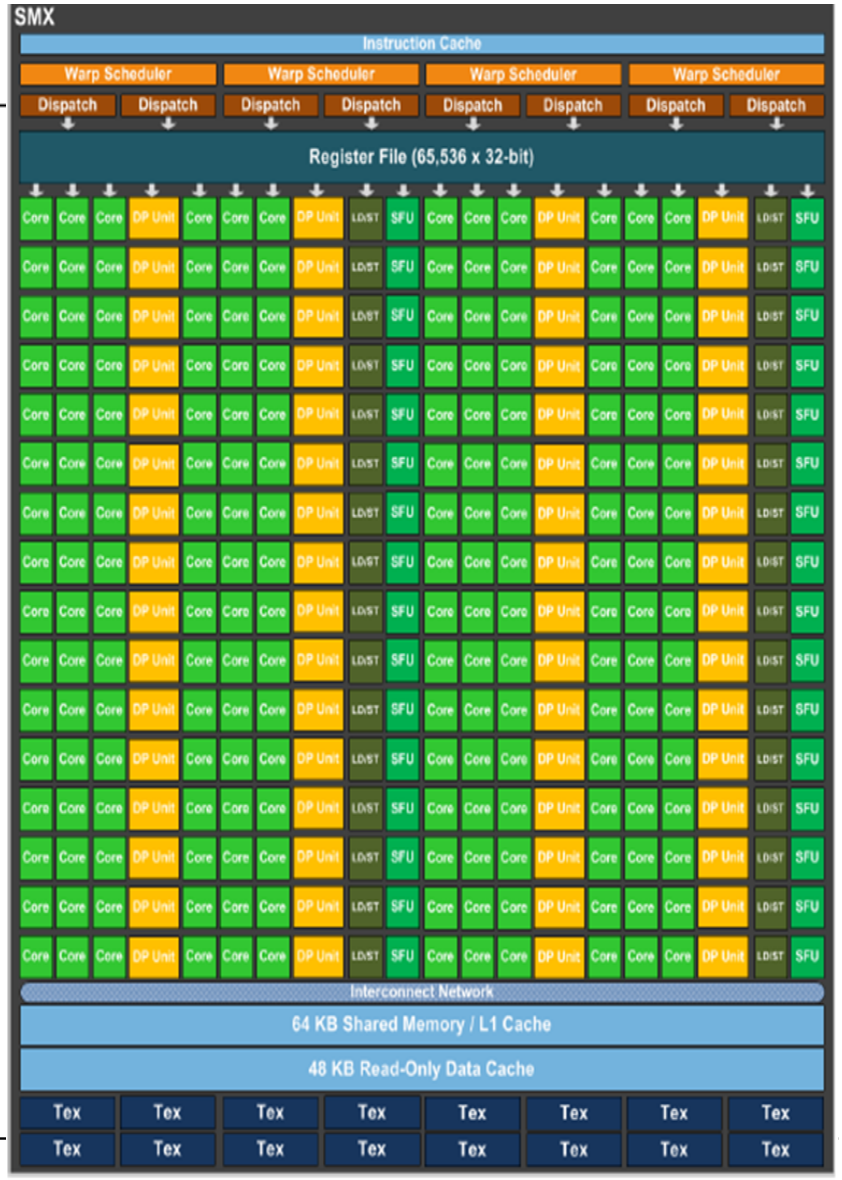
- On a 512*512*512 cube (on Tesla K80):
  - 1st version: **21471.871** µs
  - 2nd version: **8530.880** µs
- This optimization increases pressure on registers and may degrade performance in peculiar cases (more register spilling, less occupancy)

Increasing GPU Efficiency

05/08/2020

Application bottlenecks

- Compute bound : all computation units used
  - INT/SP computations
  - DP computations
  - SFU computations
- Memory bound : computation units are waiting for operand data from load/store units
  - Latency bound
  - Bandwidth bound



Definitions

- Latency: time required to perform an operation
- Throughput: how many operations complete per cycle
- Pizza delivery analogy:
  - Latency: Pizza delivery takes 30 minutes to go from pizza shop to your house
  - Throughput: Pizza delivery can’t deliver more than 5 pizzas at a time

Latencies: Kepler Example

- Global / local memory : ~200-400 clock cycles of memory latency (3.x)
- Registers : zero extra clock cycles /instruction
  - Issue:
    - Delays (11 cycles) may occur due to register read-after-write dependency. No dependent operation can start for this time.

```
x = a + b; // takes ~11 cycles to execute
y = a + c; // independent, can start anytime
(stall)
z = x + d; //dependent, must wait for completion
```

Latencies: Kepler Example

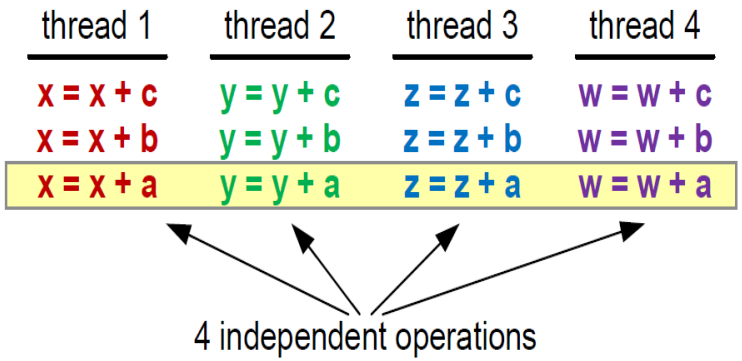
- Global / local memory : ~200-400 clock cycles of memory latency (3.x)
- Registers : zero extra clock cycles /instruction
  - Issue:
    - Delays (11 cycles) may occur due to register read-after-write dependency. No dependent operation can start for this time.
  - Solution:
    - Hide it by overlapping with other operations.

```
x = a + b; // takes ~11 cycles to execute
y = a + c; // independent, can start anytime
// add other independent operations
z = x + d; //dependent, must wait for completion
```

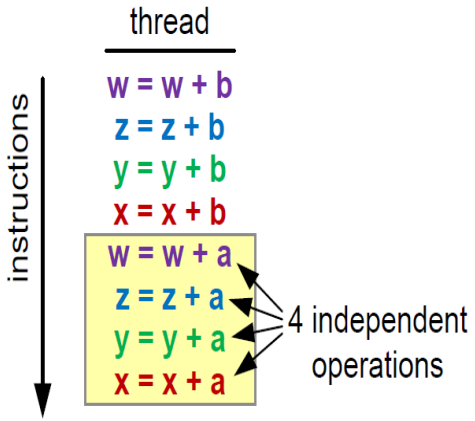
Adding Other Operations

- Adding other operations can be done with:

Thread level parallelism (TLP)



Instruction level parallelism (ILP)

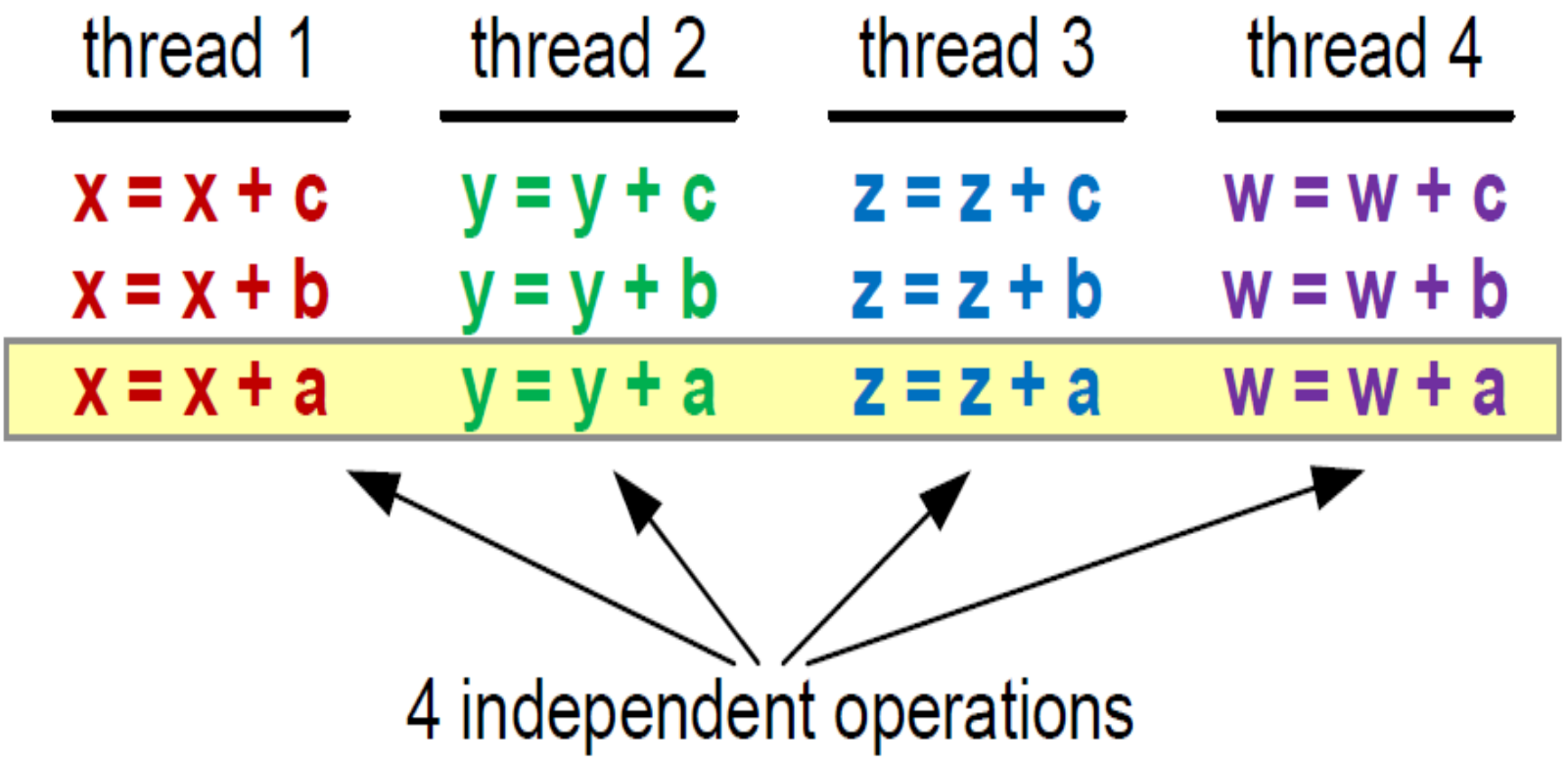


- Hiding Latency = Maintaining Throughput

Thread Level Parallelism: Occupancy

05/08/2020

Thread Level Parallelism



Occupancy

- What is occupancy?
  - Scope:
    - Software: kernel (not the whole program)
    - Hardware: streaming multiprocessor (independent of the amount of SM available)
  - Measure:

$$occupancy = \frac{\#active\ threads}{\#maximum\ supported\ threads}$$

- Why should I maximize it ?
  - If enough transactions in flight, latency is covered through warp context switching

Occupancy

- Potential occupancy limiters:
  - Hardware resources are allocated for an entire block
  - Utilizing too many resources per thread may limit the occupancy

- Because of this, programmers need to choose with care :
  - **The size of thread blocks**
  - The right **amount** of mixed resources per thread: **registers** & **shared memory**

- Occupancy of a same kernel may vary on different architecture:
  - due to differences between streaming multiprocessors:
    - 32 bit-registers: 64K on 3.5, 128K on 3.7
    - shared memory size : 112KB on 3.7, 96 KB maximum on 7.0

Potential Occupancy Limiters : Block Size

Example on 3.5 architecture:

- Each SM can have up to :
  - **maximum 2048 active threads**

$$occupancy = \frac{\#active\ threads}{\#maximum\ supported\ threads}$$

Block Size	Active Threads (without active block limit)	Number of Block	Occupancy
32	2048	64	1
64	2048	32	1
128	2048	16	1
192	1920 192 * 11 (blocks) > 2048	10	0,9375
256	2048	8	1

Potential Occupancy Limiters : Block Size

Example on 7.0 architecture:

- Each SM can have up to :
  - **maximum 1024 active threads per SM**

$$occupancy = \frac{\#active\ threads}{\#maximum\ supported\ threads}$$

Block Size	Active Threads (without active block limit)	Number of Block	Occupancy
32	1024	64	1
64	1024	32	1
128	1024	16	1
192	960 192 * 5 (blocks) > 1024	10	0,94
256	1024	8	1

Potential Occupancy Limiters : Block Size

Example on 3.5 / 7.5 architecture:

- Each SM can have up to :
  - **maximum 16 active blocks**

$$occupancy = \frac{\#active\ threads}{\#maximum\ supported\ threads}$$

Block Size	Active Threads (if 16 active blocks)	Occupancy on 3.5	Occupancy on 7.5
32	32 *16 = 512	512/2048 = 0,25	0.5
64	1024	0,5	1
128	2048	1	1
192	3072	>1	1
256	4096	>1	1

Potential Occupancy Limiters : Block Size

Example on 3.5 architecture:

- maximum 1024 threads in a block

block size	occupancy	block size	occupancy	block size	occupancy	block size	occupancy
32	0,25	288	0,98	544	0,80	800	0,78
64	0,5	320	0,94	576	0,84	832	0,81
96	0,75	352	0,86	608	0,89	864	0,84
128	1	384	0,94	640	0,94	896	0,88
160	0,94	416	0,81	672	0,98	928	0,91
192	0,94	448	0,88	704	0,69	960	0,94
224	0,98	480	0,94	736	0,72	992	0,97
256	1	512	1	768	0,75	1024	1

Potential Occupancy Limiters : Registers & Shared Memory

- Registers & shared memory usage given with :
  - nvcc --ptxas-options=-v / -Xptxas=-v

```
$> nvcc -Xptxas -v cuda_code.cu
ptxas info    : Compiling entry function 'cuda_kernel'
ptxas info    : Used 4 registers, 60+56 bytes lmem, 44+40 bytes smem,
                20 bytes cmem[1], 12 bytes cmem[14]
```

- Registers and shared memory are limited by streaming multiprocessor:
  - using too much of these resources reduces the occupancy

Potential occupancy limiters : Registers

Example on 3.5 architecture:

- Each SM can have up to:
  - 64K registers / SM (arch >3.5 128K registers, K80 is 3.7)

- Example 1
  - Kernel uses 32 registers per thread
  - 64k/32 = 2048 threads (>= 2048 threads thus occupancy = **1**)

- Example 2
  - Kernel uses 64 registers per thread
  - 64k/64 = 1024 threads (occupancy = 1024/2048 = **0,5**)

- Control register usage using nvcc flag: --maxregcount=N
  - registers are spilled in local memory

Potential occupancy limiters : Registers

Example on 7.5 architecture:

- Each SM can have up to:
  - 64K registers / SM (arch >3.5 128K registers, K80 is 3.7)
  - Max Threads per Multiprocessor 1024

- Example 1
  - Kernel uses 32 registers per thread
  - 64k/32 = 2048 threads (>= 1024 threads thus occupancy = **1**)

- Example 2
  - Kernel uses 64 registers per thread
  - 64k/64 = 1024 threads (occupancy = 1024/1024 = **1**)

- Control register usage using nvcc flag: --maxregcount=N
  - registers are spilled in local memory

Potential occupancy limiters : Shared Memory

Example on 3.5 architecture:

- Each SM can have up to:
  - 48k, 32k or 16k shared memory/L1 per SM
- Example 1, 48k shared memory
  - Kernel uses 24 bytes of shared memory per thread
  - 48k/24 = 2048 threads (occupancy = **1**)
- Example 2, 16k shared memory
  - Kernel uses 24 bytes of shared memory per thread
  - 16k/24 = 682 threads (occupancy = **0,3333**)
- Don’t use too much shared memory
- Choose L1/shared configuration appropriately

Potential occupancy limiters : Shared Memory

Example on 7.5 architecture:

- Per default: 32 KB of shared memory (can be switched to 64K)

- Example 1, 32k shared memory
  - Kernel uses 24 bytes of shared memory per thread
  - 32K/24 = ~1333 threads > 1024 threads (occupancy = **1**)

- Example 2, 32k shared memory, 64 threads per block
  - Kernel uses 33 bytes of shared memory per thread
  - User Shared Memory Per Block (bytes): 2560
  - Limits us to 12 active thread blocks per SM, since 13 * 2560 > 32K
  - But 12 * 64 = 768 -> Occupancy 768/1024 = **0.75**

  - Don’t use too much shared memory

- Choose L1/shared configuration appropriately

Occupancy

- But ... a better occupancy does not imply better performance:
  - fewer active threads computing in lower latency memory are usually faster:
- Computing in register and/or shared memory instead of global memory
- Occupancy is good for final optimizations
  - try to save registers or shared memory to increase active threads on optimized kernels

Modifying Occupancy

- Thread per blocks
  - block size at kernel launch
  - use a multiple of the warp size

- Shared Memory Amount
  - algorithm
  - different block sizes can use different amount of shared memory
    - for example 2D kernel using a stencil of 4 points
    - 16*8 block => (16+2*4) * (8+2*4) = 384 values needed in shared memory => 384/128 = 3 values by threads
    - 32*4 block => (32+2*4) * (4+2*4) = 480 values needed in shared memory => 480/128 = 3,75 values by threads

Modifying Occupancy

- Register usage:
  - modifying kernel code
    - moving instruction can help the compiler optimizing
  - use **--maxrregcount** compilation option
    - may increase spilling

```
nvcc --maxrregcount=64  file.cu –o a.out
```

  - use the **__launch_bounds__** macro
    - optimize register usage for a giving occupancy

```
__global__ void
__launch_bounds__ ( maxThreadsPerBlock , minBlocksPerMultiprocessor )
MyKernel (...) { ...}
```

Occupancy Calculator

CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	3.5	(Help)
1.b) Select Shared Memory Size Config (bytes)	49152	

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	32	
Shared Memory Per Block (bytes)	4096	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:	(Help)
Active Threads per Multiprocessor	2048
Active Warps per Multiprocessor	64
Active Thread Blocks per Multiprocessor	8
Occupancy of each Multiprocessor	100%

Physical Limits for GPU Compute Capability:	3.5
Threads per Warp	32
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Thread Blocks per Multiprocessor	16
Total # of 32-bit registers per Multiprocessor	65536
Register allocation unit size	256
Register allocation granularity	warp
Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	49152
Shared Memory Allocation unit size	256
Warp allocation granularity	4
Maximum Thread Block Size	1024

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	8	64	8
Registers (Warp limit per SM due to per-warp reg count)	8	64	8
Shared Memory (Bytes)	4096	49152	12

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM * Warps/Block = Warps/SM		
Limited by Max Warps or Max Blocks per Multipro	8	8	64
Limited by Registers per Multiprocessor	8	8	64
Limited by Shared Memory per Multiprocessor	12		

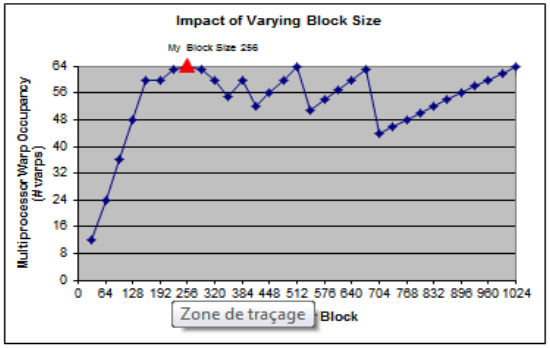
Note: Occupancy limiter is shown in orange

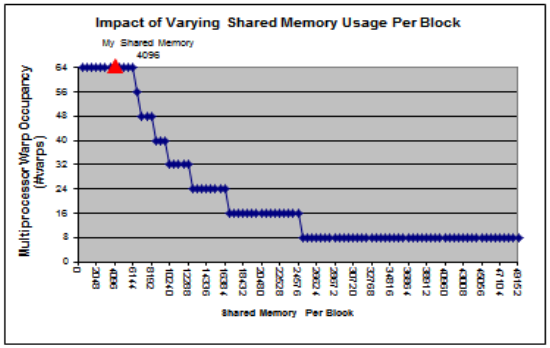
Physical Max Warps/SM = 64
Occupancy = 64 / 64 = 100%

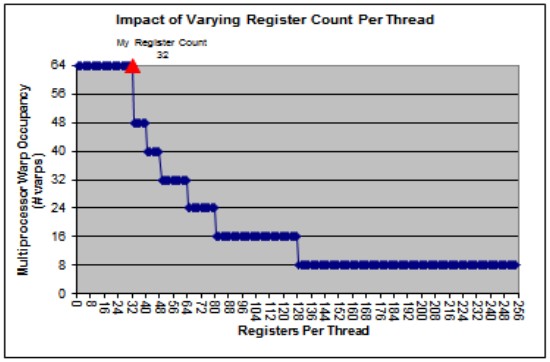
CUDA Occupancy Calculator	
Version:	5.1
Copyright and License	

k Here for detailed instructions on how to use this occupancy calcula
or more information on NVIDIA CUDA, visit http://developer.nvidia.com/cu

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.







Compute Capability

05/08/2020

Feature Support

Table 13. Feature Support per Compute Capability

Feature Support	Compute Capability					
(Unlisted features are supported for all compute capabilities)	3.0	3.2	3.5, 3.7, 5.0, 5.2	5.3	6.x	7.x
Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)	Yes					
atomicExch() operating on 32-bit floating point values in global memory (atomicExch())	Yes					
Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)	Yes					
atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())	Yes					
Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)	Yes					
Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)	Yes					
Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())	Yes					
Atomic addition operating on 64-bit floating point values in global memory and shared memory (atomicAdd())	No				Yes	
Warp vote and ballot functions (Warp Vote Functions)	Yes					
__threadfence_system() (Memory Fence Functions)						
__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Synchronization Functions)						
Surface functions (Surface Functions)						
3D grid of thread blocks						
Unified Memory Programming						
Funnel shift (see reference manual)	No	Yes				
Dynamic Parallelism	No		Yes			
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No			Yes		
Tensor Core	No					Yes

Technical Specifications

Table 14. Technical Specifications per Compute Capability

	Compute Capability											
Technical Specifications	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.5**
Maximum number of resident grids per device (Concurrent Kernel Execution)	16	4	32				16	128	32	16	128	
Maximum dimensionality of grid of thread blocks	3											
Maximum x-dimension of a grid of thread blocks	$2^{31}-1$											
Maximum y- or z-dimension of a grid of thread blocks	65535											
Maximum dimensionality of thread block	3											
Maximum x- or y-dimension of a block	1024											
Maximum z-dimension of a block	64											
Maximum number of threads per block	1024											
Warp size	32											
Maximum number of resident blocks per multiprocessor	16				32							16
Maximum number of resident warps per multiprocessor	64											32
Maximum number of resident threads per multiprocessor	2048											1024
Number of 32-bit registers per multiprocessor	64 K			128 K	64 K							
Maximum number of 32-bit registers per thread block	64 K	32 K	64 K				32 K	64 K		32 K	64 K	
Maximum number of 32-bit registers per thread	63	255										
Maximum amount of shared memory per multiprocessor	48 KB			112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB	64 KB
Maximum amount of shared memory per thread block [27]	48 KB										96 KB	64 KB
Number of shared memory banks	32											
Amount of local memory per thread	512 KB											
Constant memory size	64 KB											
Cache working set per multiprocessor for constant memory	8 KB							4 KB	8 KB			
Cache working set per multiprocessor for texture memory	Between 12 KB and 48 KB							Between 24 KB and 48 KB			32 - 128 KB	32 or 64 KB
Maximum width for a 1D texture reference bound to a CUDA array	65536											

...

Hardware Specificities

05/08/2020

Kepler

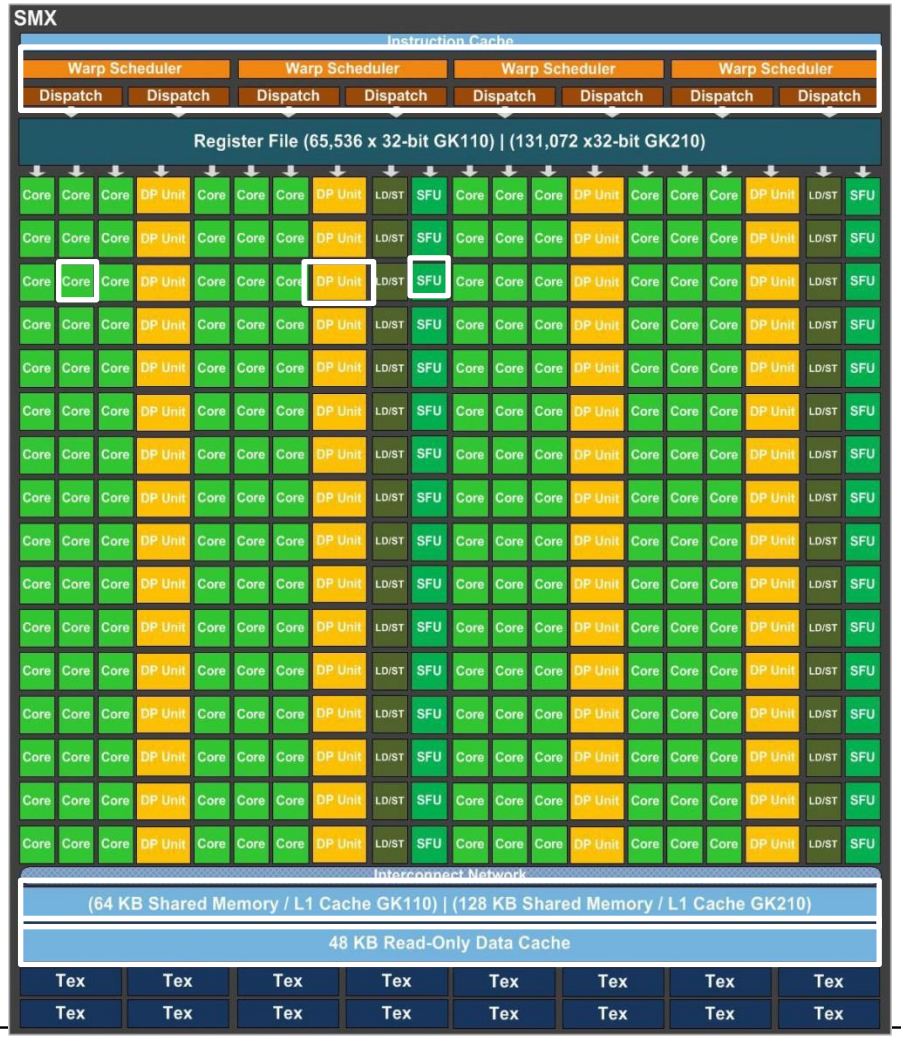
05/08/2020

Kepler GK110/GK210 Full chip block diagram

- 15 SM on die
  - GK110B (K40): 15
  - GK210 (K80): 13

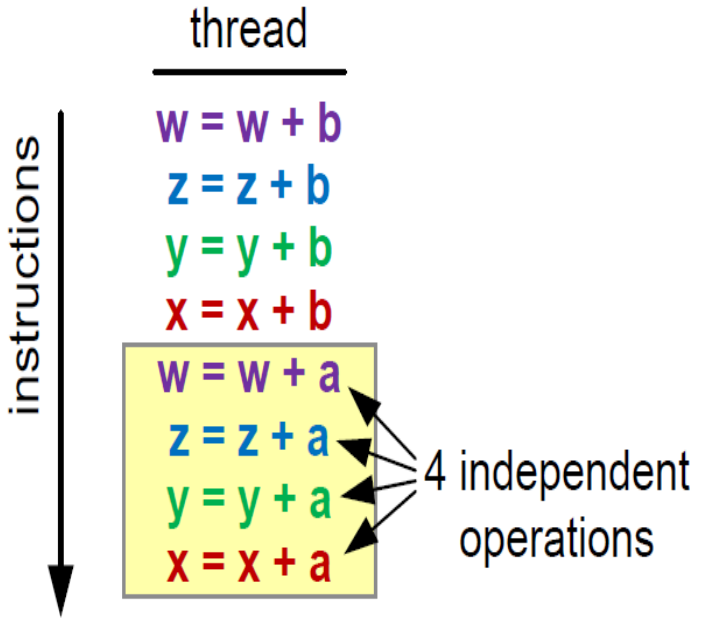
GK110/GK210 Streaming Multiprocessor

- 192 CUDA cores for 32-bit floating-point arithmetic operations (add,mul,muladd)
- 64 double-precision units
- 32 special function units for single-precision floating-point transcendental functions
- 4 warp schedulers
  - 8 dispatch units
- 1 Read-Only Data Cache
- Shared Memory/L1 Cache



Kepler Specificities

- Kepler benefits from instruction level parallelism (ILP) in single precision
  - Help to hide latencies and/or to saturate the computation units
- 192 units for 32-bit floating-point add, multiply, multiply-add
- 4 warp schedulers / multiprocessor
  - but 8 dispatch units:
    - **2 independent instructions / cycle**
- Without ilp:
  - 4 * 32 * 1 op/cycle = **128 < 192**
- With ilp:
  - 4 * 32 * 2 ops/cycle = **256 > 192**
- Unrolling could help !!!



Kepler Specificities

- By default on Kepler, the L1 cache is used to **cache** accesses to **local memory,** including **temporary register spills**

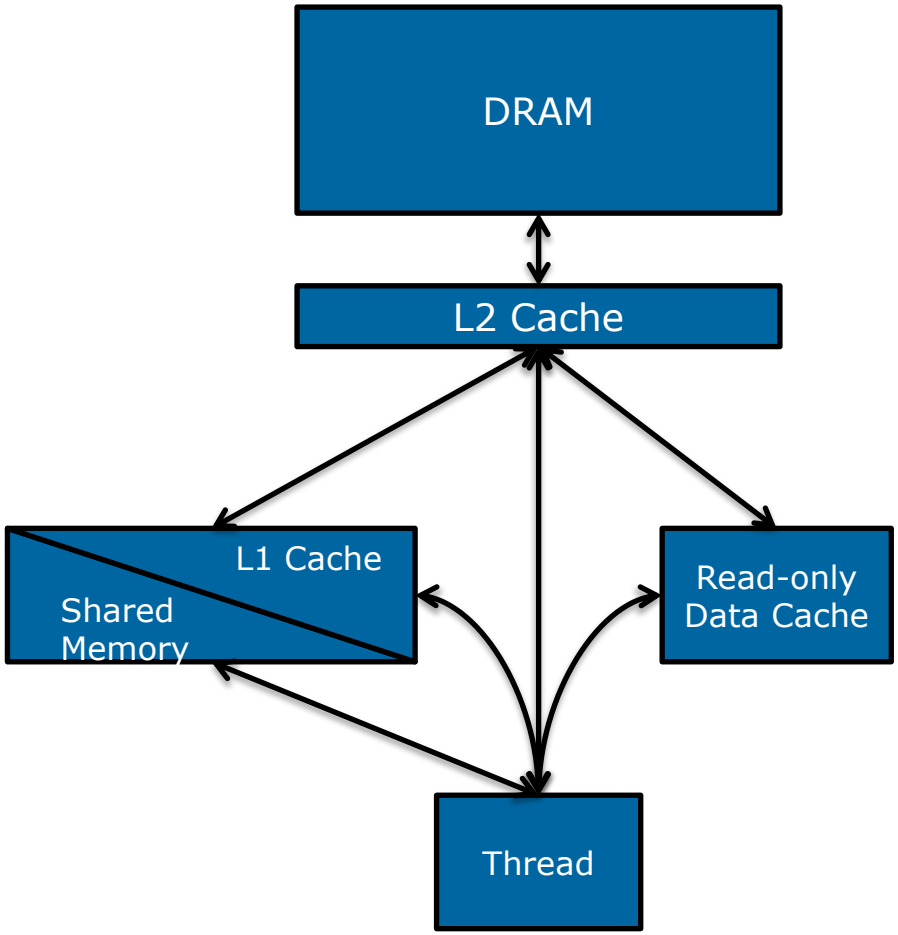
- Shared memory and L1 cache shared the same memory space
  - initial configuration
    - 48 KB of shared memory (**maximum per thread blocks**)
    - 16 KB of L1 cache

- Configurable using cudaDeviceSetCacheConfig():
  - 48 KB shmem / 16 KB L1
  - 32 KB shmem / 32 KB L1
  - 16 KB shmem / 48 KB L1

- The L1 can be used to cache global memory accesses using "-Xptxas –dlcm=ca" option to nvcc:
  - available on Tesla 3.5 and 3.7 devices

Memory Paths on Kepler

- GDDR5:
  - K40 (GK110B):
    - 12 Go - 288 GB/s
  - K80 (2*GK210):
    - 2*12 Go - 2*240 GB/s
- L2: 1536KB
- Read-Only Data Cache / Texture Cache:
  - 48 KB
- Shared Memory / L1 Cache:
  - GK110B: 64 KB
    - 48/16, 32/32 or 16/48
  - GK210: 64 KB (+ 64KB of shmem)
    - 112/16, 96/32 or 80/48
    - **48 KB shmem max per thread block (occupancy)**

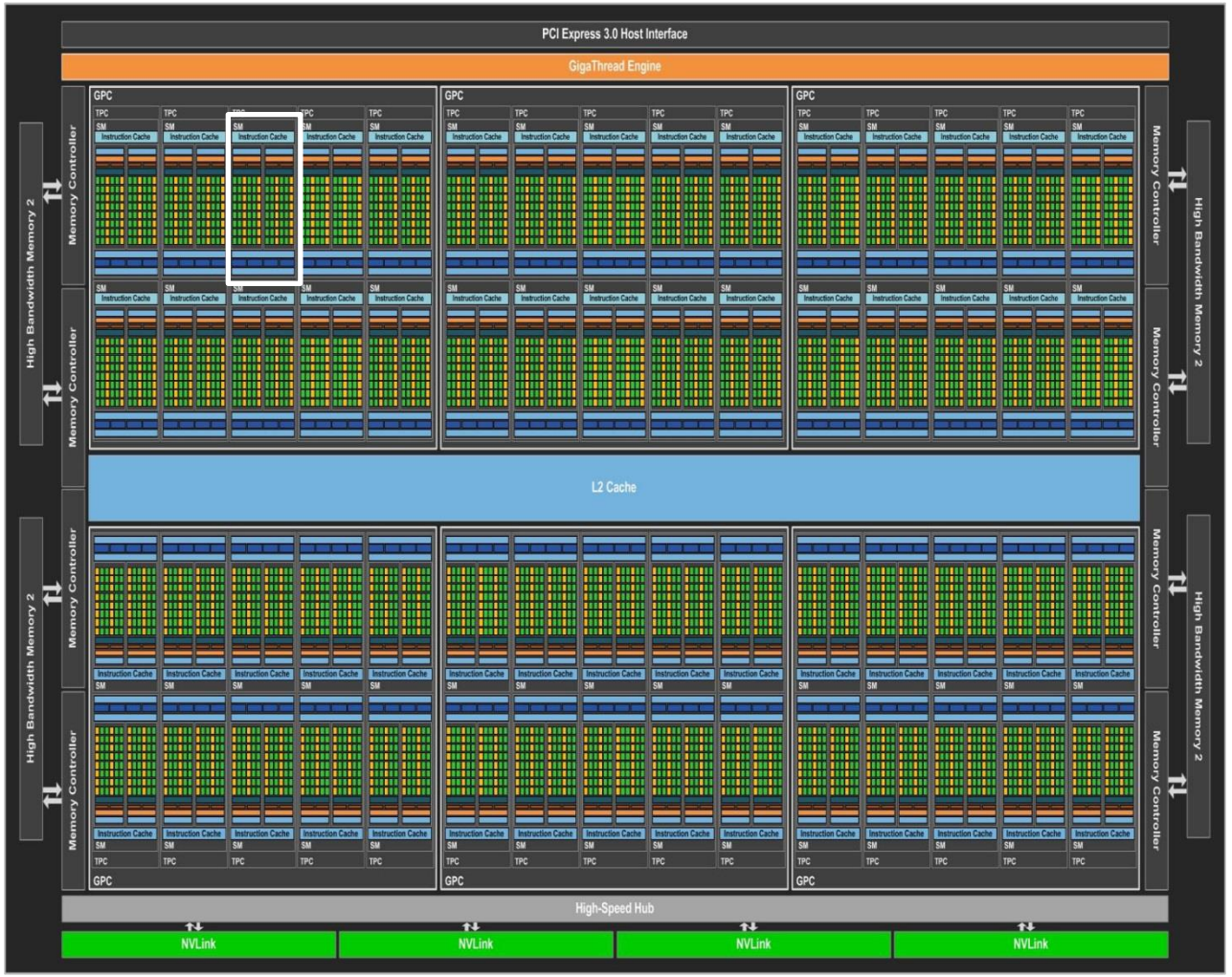


Pascal

05/08/2020

Pascal GP100 Full GPU with 60 SM Units

- 60 SM on die
  - P100: 56



GP100 Streaming Multiprocessor

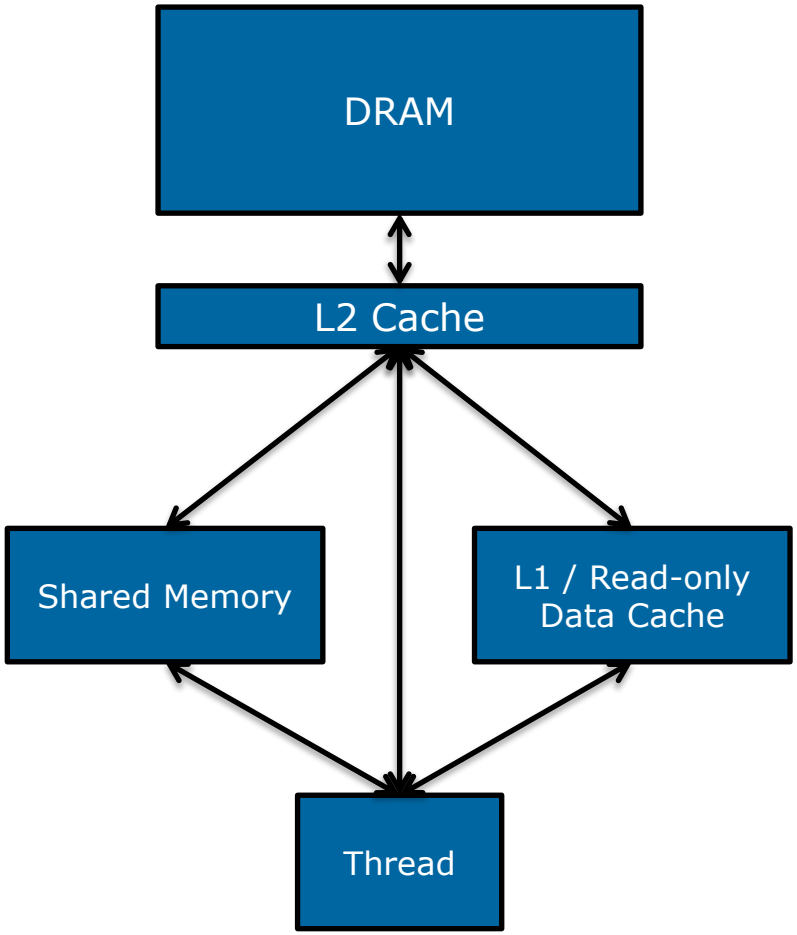
- 64 CUDA cores for 32-bit floating-point arithmetic operations (add,mul,muladd)
- 32 double-precision units
- 16 special function units for single-precision floating-point transcendental functions
- 2 warp schedulers:
  - 4 dispatch units
- **Unified** L1 Cache / Read-Only Data Cache
- Shared Memory

Pascal Specificities

- Shared Memory and L1 Cache fixed:
  - no more need to configure ressources
- 2 warp schedulers and 4 dispatch units:
  - 1 memory instruction and 1 instruction other than memory by cycle:
    - but only if contiguous in the instruction flow
    - otherwise 1 instruction/cycle
- Nvlink:
  - GPU (CPU) interconnection
  - 6 links: total bandwidth 300 Go/s
- Support of Unified Memory
  - GPU can access “any” page of the entire system memory
  - migrate the data on-demand to its own memory for high bandwidth access

Memory Paths on Pascal (P100)

- HBM2:
  - 12Go: 549 GB/s
  - 16Go: 732 GB/s
- L2: 4096KB
- Unified L1/Read-Only Data Cache (texture cache):
  - 24 KB
  - P4: 24 KB / P40 : 48KB
- Shared Memory :
  - 64 KB
  - P4: 64 KB / P40 : 96 KB
  - **48 KB shmem max per thread block (occupancy)**



Volta

05/08/2020

Volta GV100 Full GPU with 84 SM Units

- 84 SM on die
  - V100: 80

GV100 Streaming Multiprocessor

- 64 CUDA cores for 32-bit floating-point arithmetic operations (add,mul,muladd)
- 32 CUDA cores for double-precision
- 16 special function units (SFU) for single-precision floating-point transcendental functions
- 4 warp schedulers
  - 4 dispatch units
- Unified Shared Memory, L1 Cache and Read-Only Data Cache

Volta Specificities

- Shared Memory and L1 Cache configuration is automatic
- Memory accesses performance reduced between shared memory and L1 cache
- SIMT execution model
  - **Independent Thread Scheduling among threads in a warp:**
    - the GPU maintains execution state per thread, including a program counter and call stack
    - Enables intra-warp synchronization patterns previously unavailable
    - Assumptions about warp-synchronicity of previous hardware architectures may lead to wrong code on Volta
      - __syncwarp() forces threads within a warp to synchronize

Be careful of Warp-synchronicity

- Before Volta

```
// Intra-warp reduction
   // Butterfly reduction simplifies syncwarp mask
   if (tid < 32) {
     s_buff[tid] += s_buff[tid^16];
     s_buff[tid] += s_buff[tid^ 8];
     s_buff[tid] += s_buff[tid^ 4];
     s_buff[tid] += s_buff[tid^ 2];
}
```

- Since Volta

```
// Intra-warp reduction
   // Butterfly reduction simplifies syncwarp mask
   if (tid < 32) {
      float temp;
      temp = s_buff[tid^16]; __syncwarp();
      s_buff[tid] += temp;   __syncwarp();
      temp = s_buff[tid^ 8]; __syncwarp();
      s_buff[tid] += temp;   __syncwarp();
      temp = s_buff[tid^ 4]; __syncwarp();
      s_buff[tid] += temp;   __syncwarp();
      temp = s_buff[tid^ 2]; __syncwarp();
      s_buff[tid] += temp;   __syncwarp();
   }
```

Be careful of Warp-synchronicity

- Be really careful:

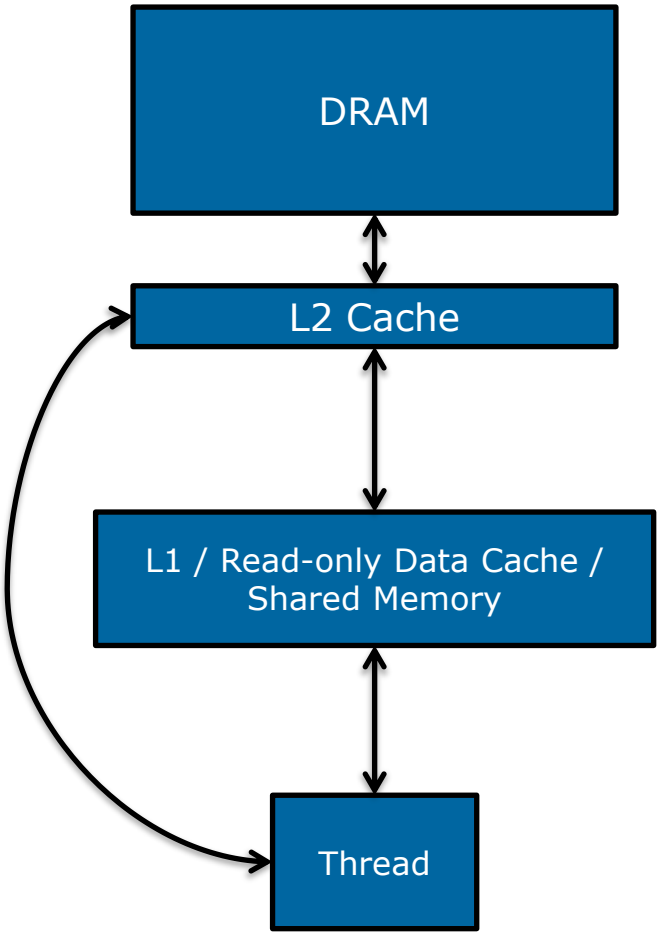
```
// Intra-warp reduction
   // Butterfly reduction simplifies syncwarp mask
   if (tid < 32) {
     s_buff[tid] += s_buff[tid^16]; __syncwarp();
     s_buff[tid] += s_buff[tid^ 8]; __syncwarp();
     s_buff[tid] += s_buff[tid^ 4]; __syncwarp();
     s_buff[tid] += s_buff[tid^ 2]; __syncwarp();
}
```

Incorrect !!!

```
// Intra-warp reduction
   // Butterfly reduction simplifies syncwarp mask
   if (tid < 32) {
      float temp;
      temp = s_buff[tid^16]; __syncwarp();
      s_buff[tid] += temp;   __syncwarp();
      temp = s_buff[tid^ 8]; __syncwarp();
      s_buff[tid] += temp;   __syncwarp();
      temp = s_buff[tid^ 4]; __syncwarp();
      s_buff[tid] += temp;   __syncwarp();
      temp = s_buff[tid^ 2]; __syncwarp();
      s_buff[tid] += temp;   __syncwarp();
   }
```

Memory Paths on Volta (V100)

- HBM2:
  - 16Go: 900 GB/s
- L2: 6144 KB
- Unified Shared Memory and Data Cache:
  - 128 KB
    - shared memory: 0 to 96 KB
    - remaining for L1/Read-Only Data Cache/Texture Cache



Ampere

05/08/2020

Moving from Volta to Ampere

Architecture	GV100	GA100
Streaming Multiprocessors	80	108
Clock speed (boost)	1530 MHz	1410 MHz
Global Memory	32 GB (16 GB) (877 MHz)	40 GB (1215 MHz)
Memory Bandwidth	900 GB/s	1600 GB/s
L2 Cache Size	6144 KB	40960 KB
Shared Memory / SM	Up to 96 KB per SM	Up to 164 per SM
	No tensor cores for FP64	Support for Bfloat16, Int8, TF32, **FP64 Tensor (ieee)**

CUDA 11 with A100

- CUDA 11 has been extended by native support of CUB
  - cuBlas
  - cuSparse
  - cuTensor
  - cuSolver
  - cuFFT
  - CUDA Math API
  - …

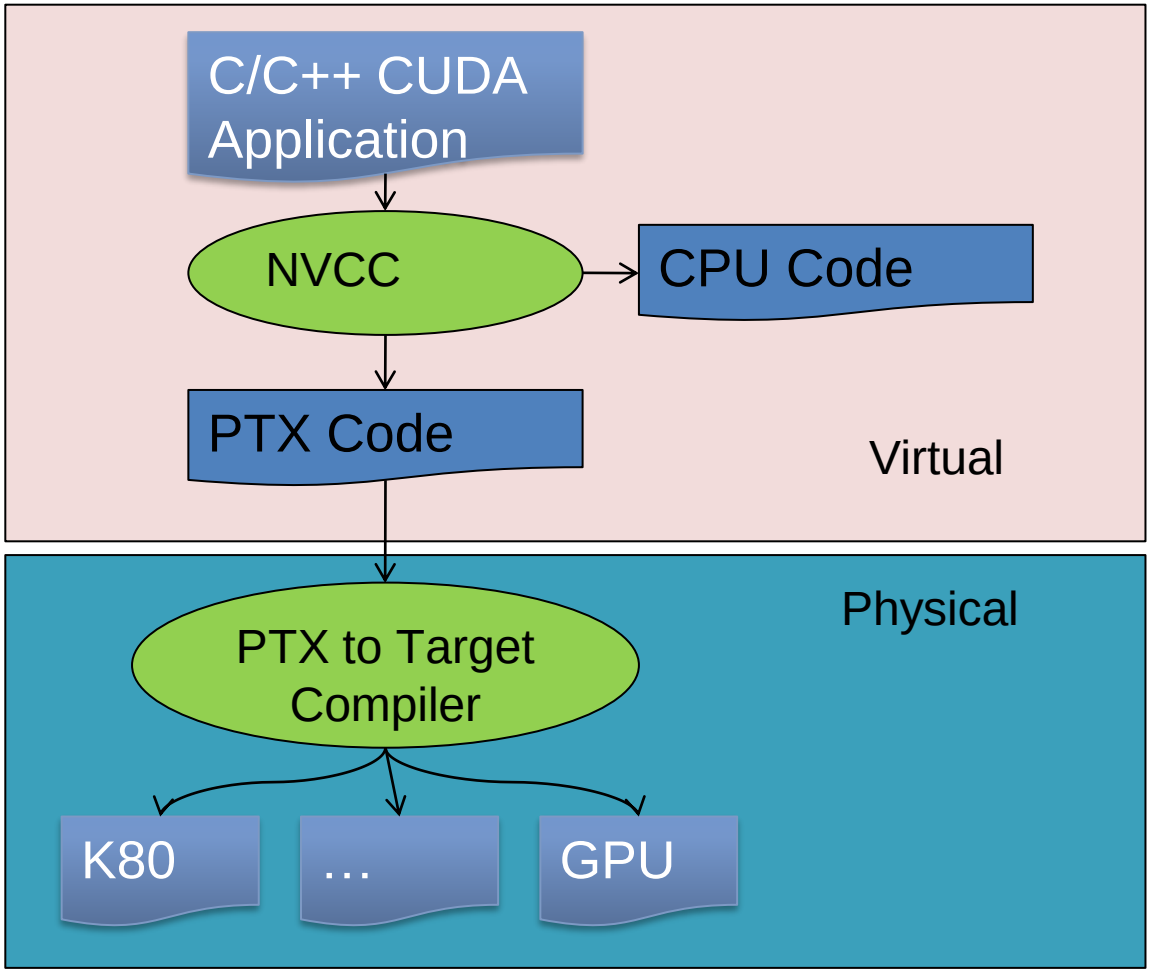
Hardware Specific Compilation

05/08/2020

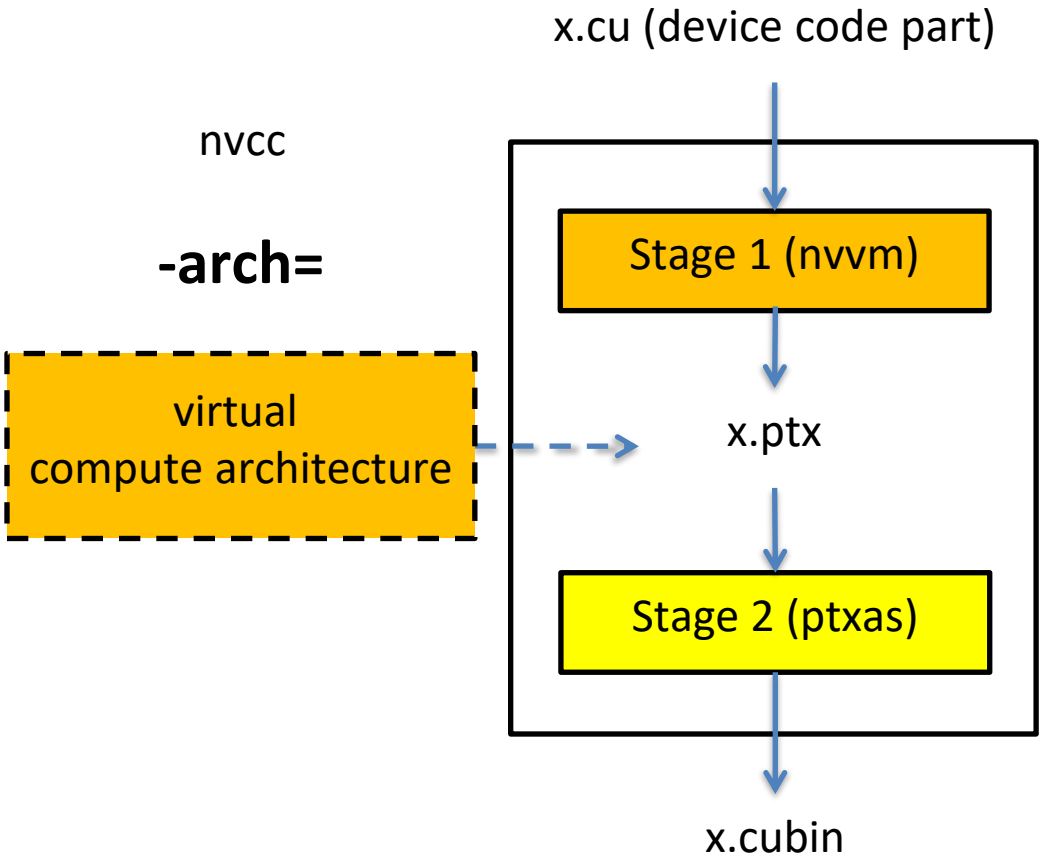
Compilation

- Any source file containing CUDA language extensions must be compiled with **nvcc**
  - *.cu files
- nvcc is a compiler driver
  - Works by invoking all the necessary tools and compilers like nvvm, g++, ...
- nvcc can output:
  - Either C code
    - That must then be compiled with the rest of the application using another tool
  - Or object code directly

Compilation

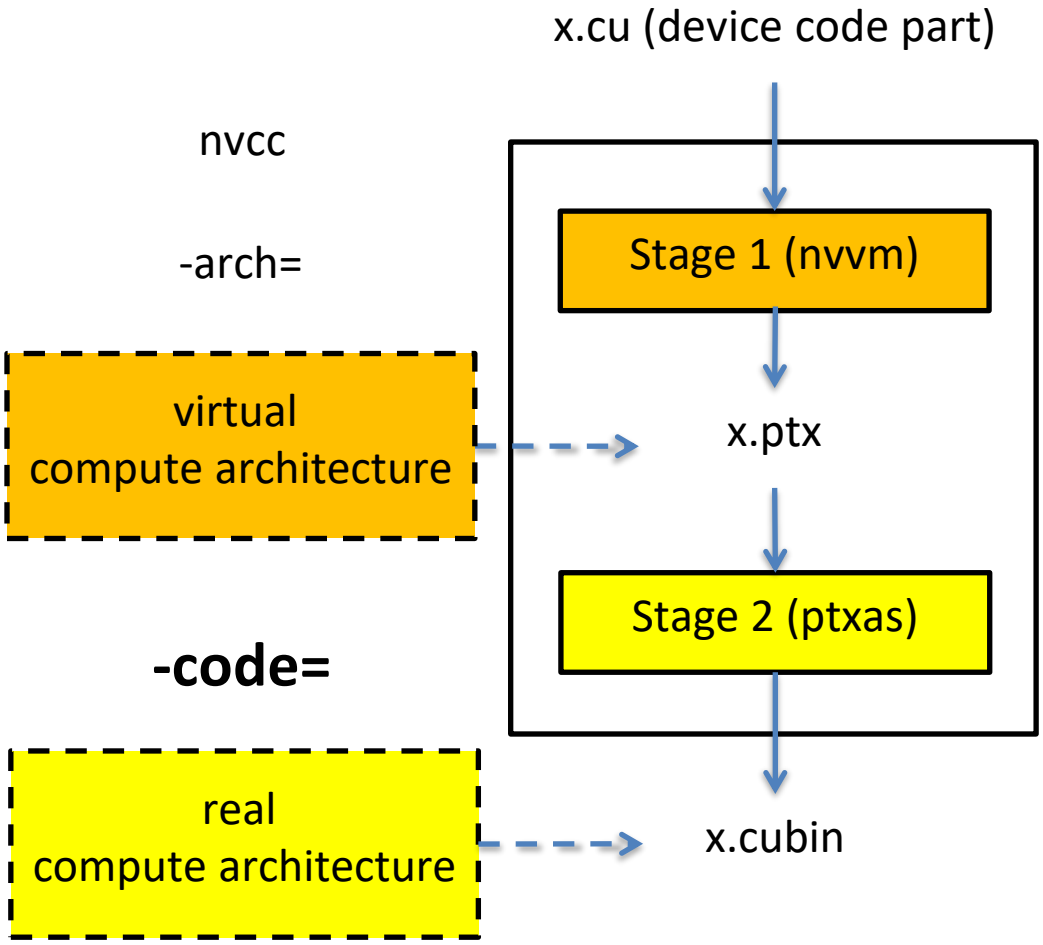


Compiling for the Right Architecture



Specify the name of the class of NVIDIA *virtual* GPU architecture for which the CUDA input files must be compiled.

Compiling for the Right Architecture



Specify the name of the NVIDIA GPU to assemble and optimize PTX for.

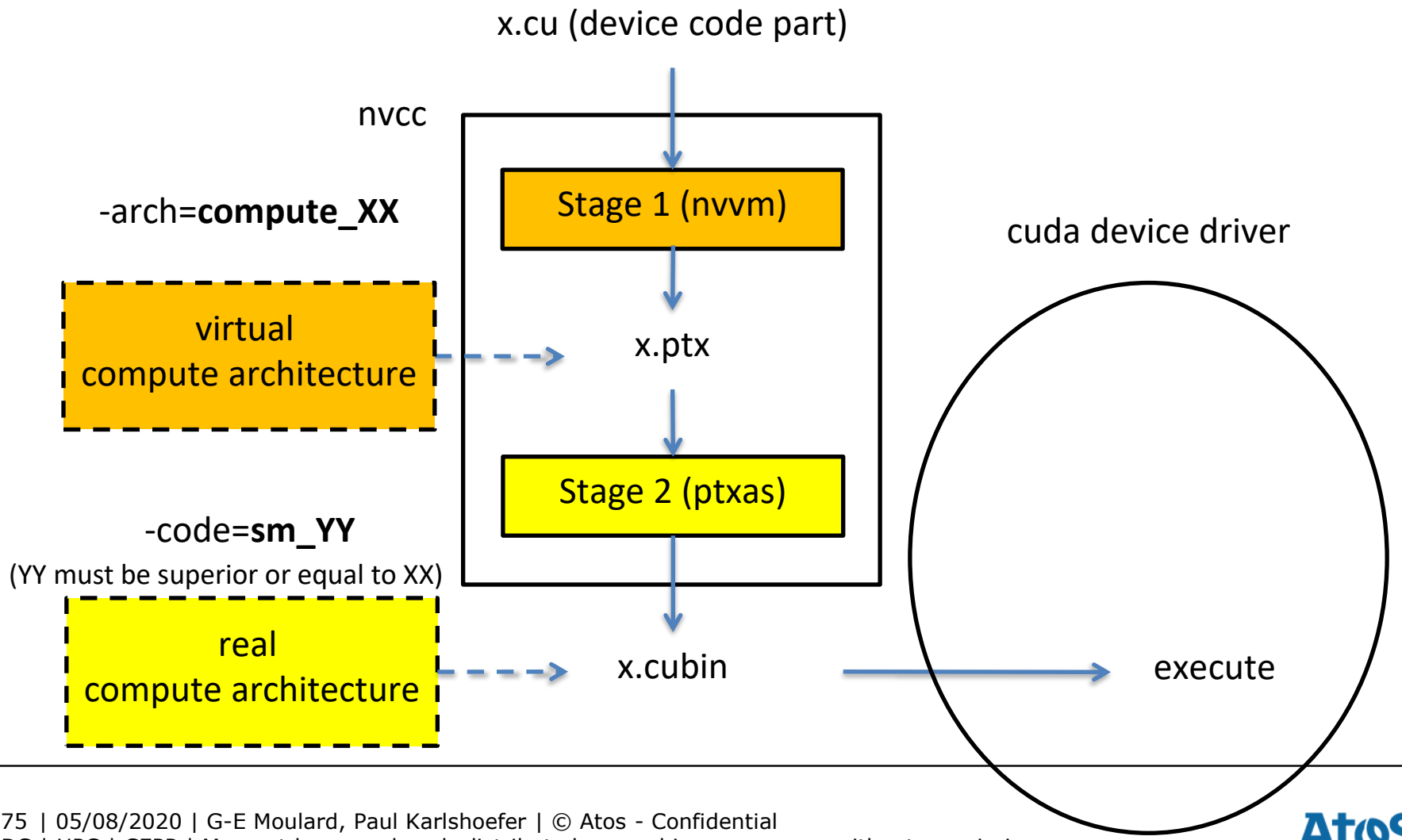
Virtual Architecture Feature List

compute_30 and compute_32	Basic features + Kepler support + Unified memory programming
compute_35	+ Dynamic parallelism support
compute_50, compute_52, and compute_53	+ Maxwell support
compute_60, compute_61, and compute_62	+ Pascal support
compute_70	+ Volta support

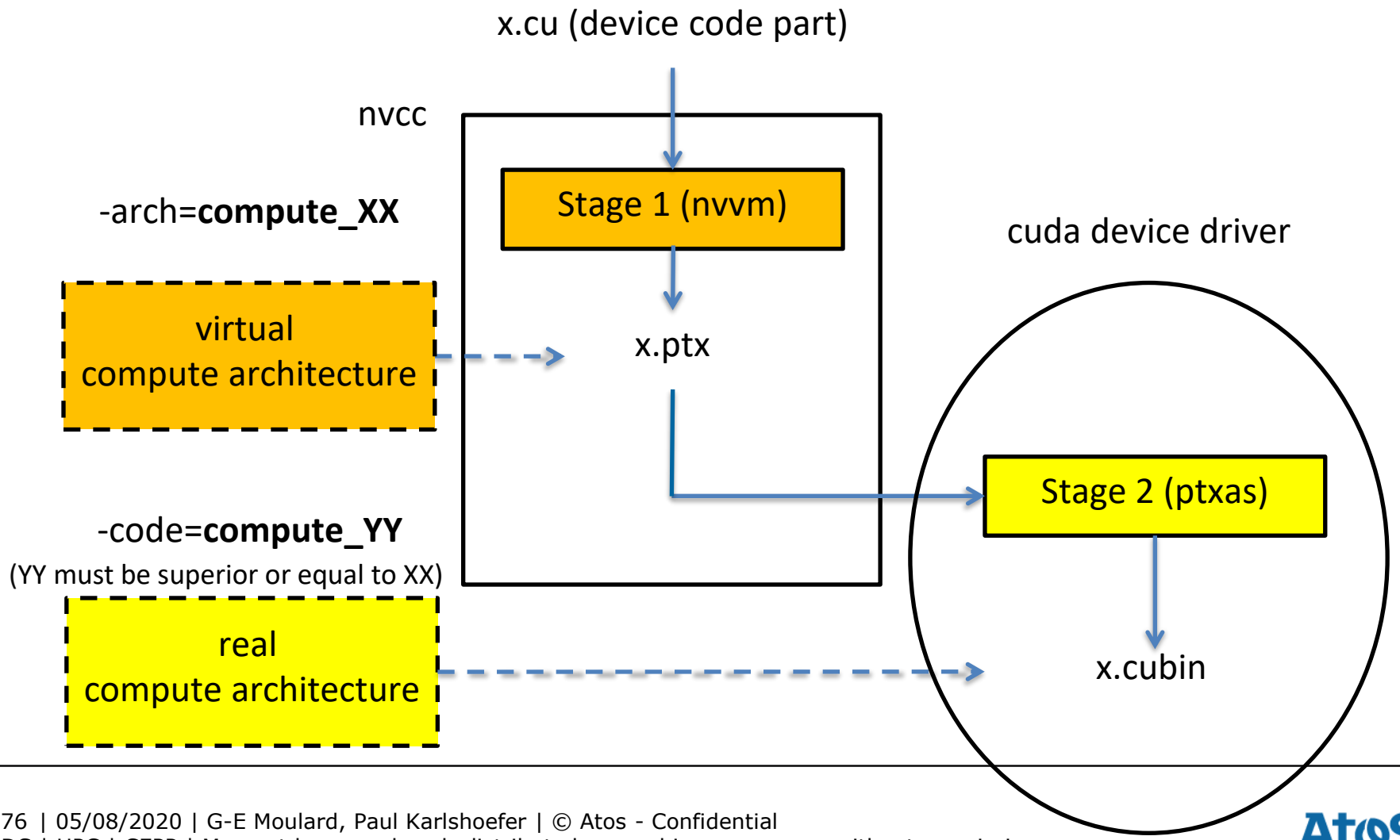
GPU Feature List

sm_30 and sm_32	Basic features + Kepler support + Unified memory programming
sm_35	+ Dynamic parallelism support
sm_50, sm_52, and sm_53	+ Maxwell support
sm_60, sm_61, and sm_62	+ Pascal support
sm_70	+ Volta support

Compiling for the Right Architecture



Compiling for the Right Architecture



Compiling for the Right Architecture

- Fat binary (multiple architecture binaries in one)

```
nvcc x.cu  -arch=compute_30  -code=compute_30,sm_30,sm_35
```

  - generates exact code for **two Kepler variants**, plus **PTX code for use by JIT** in case a next-generation GPU is encountered

- Defining multiple architectures

```
nvcc  -gencode=arch=XX1,code=YY1  -gencode=arch=XX2,code=YY2 …
```

- Shortcuts

```
nvcc x.cu  -arch=sm_35
=> nvcc x.cu  -arch=compute_35  -code=sm_35,compute_35
```

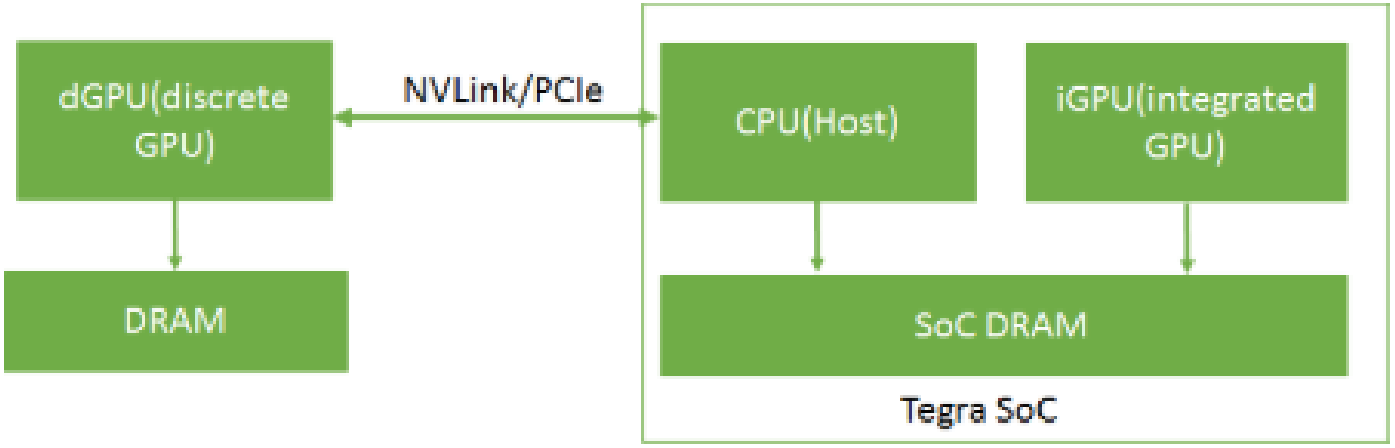
```
nvcc x.cu  -arch=compute_35
=> nvcc x.cu  -arch=compute_35  -code=compute_35
```

CUDA for Tegra

05/08/2020

Tegra architecture

- CPU and integrated GPU (iGPU) share the same RAM
  - Thus, device, host and unified memory are all allocated on the same physical buffer



Characteristics

- Not all the memory is accessible to the integrated GPU
  - Device memory **OK** (cached)
    - Use for memory with is only accessed by the iGPU
  - Pageable host memory **KO**
    - Use this memory only for memroy which is accessed by the CPU
  - Pinned host memory **OK** (unached)
    - CPU access time might be higher (depends on compute capability and if system is IO coherent)
    - Little overhead
    - Use for small buffers
  - Unified memory **OK** (cached)
    - Better caching effects
    - Use for large buffers, which are frequently used

Tesla (/GeForce) -> Tegra ?

- **Rethink your memory management**

- Not all function might be available (mostly targeting memory operations)
  - E.g. no `cudaStreamAttachMemAsync()` available
    - Different ways to prefetch memory

- Have a look into **EGL Interoperability**
  - Not covered here (not part of HPC)

Roofline Model

05/08/2020

Arithmetic Intensity

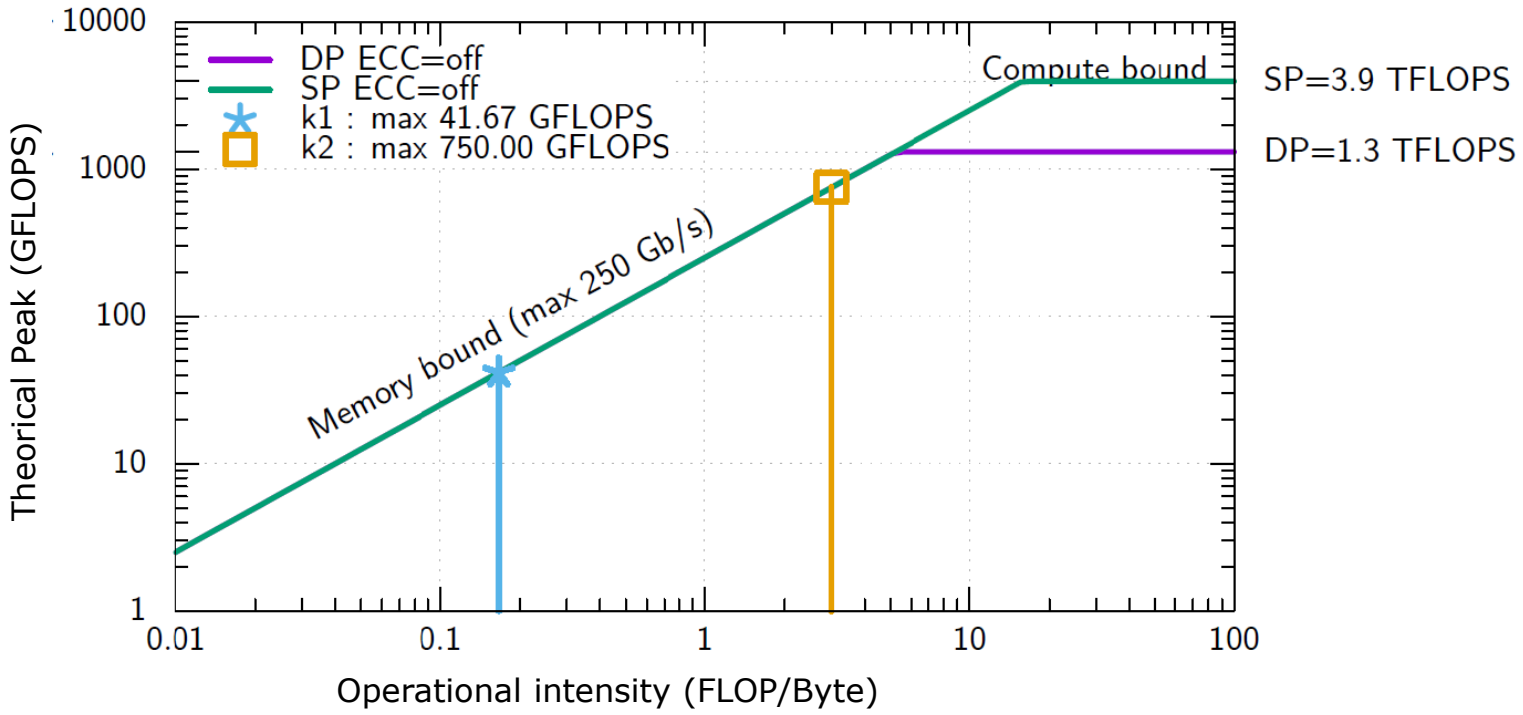
$$AI = \frac{number\ of\ arithmetic\ instructions}{size\ of\ global\ memory\ accesses}$$

```
__global__ void k1( float *a, float *b) {
  float x = a[ threadIdx .x];  // read of 4 bytes
  float y = b[ threadIdx .x];  // read of 4 bytes
  // 2 instructions and 1 write of 4 bytes
  a[ threadIdx .x] = x * x + y;
}

__global__ void k2( float *a) {
  float x = threadIdx .x;
  // 12 instructions and 1 write of 4 bytes
  a[ threadIdx .x] = x*x*x*x + 3.14 f*x*x*x - 5.3 f*x*x + 0.5 f*x;
}
```

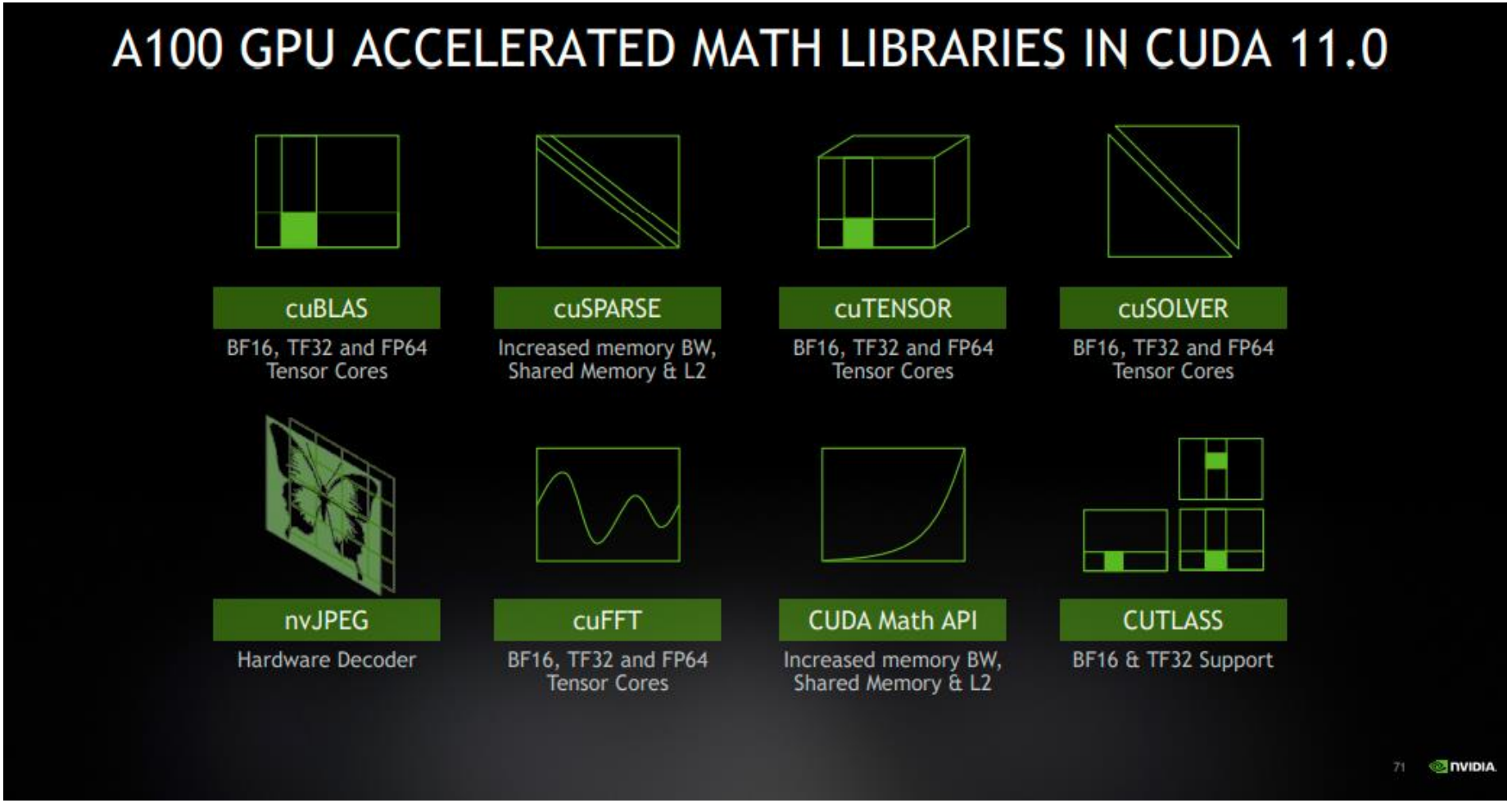
- A1(k1) = 2/12 (3*4) = 0,1666 A1(k1) = 12/4 = 3

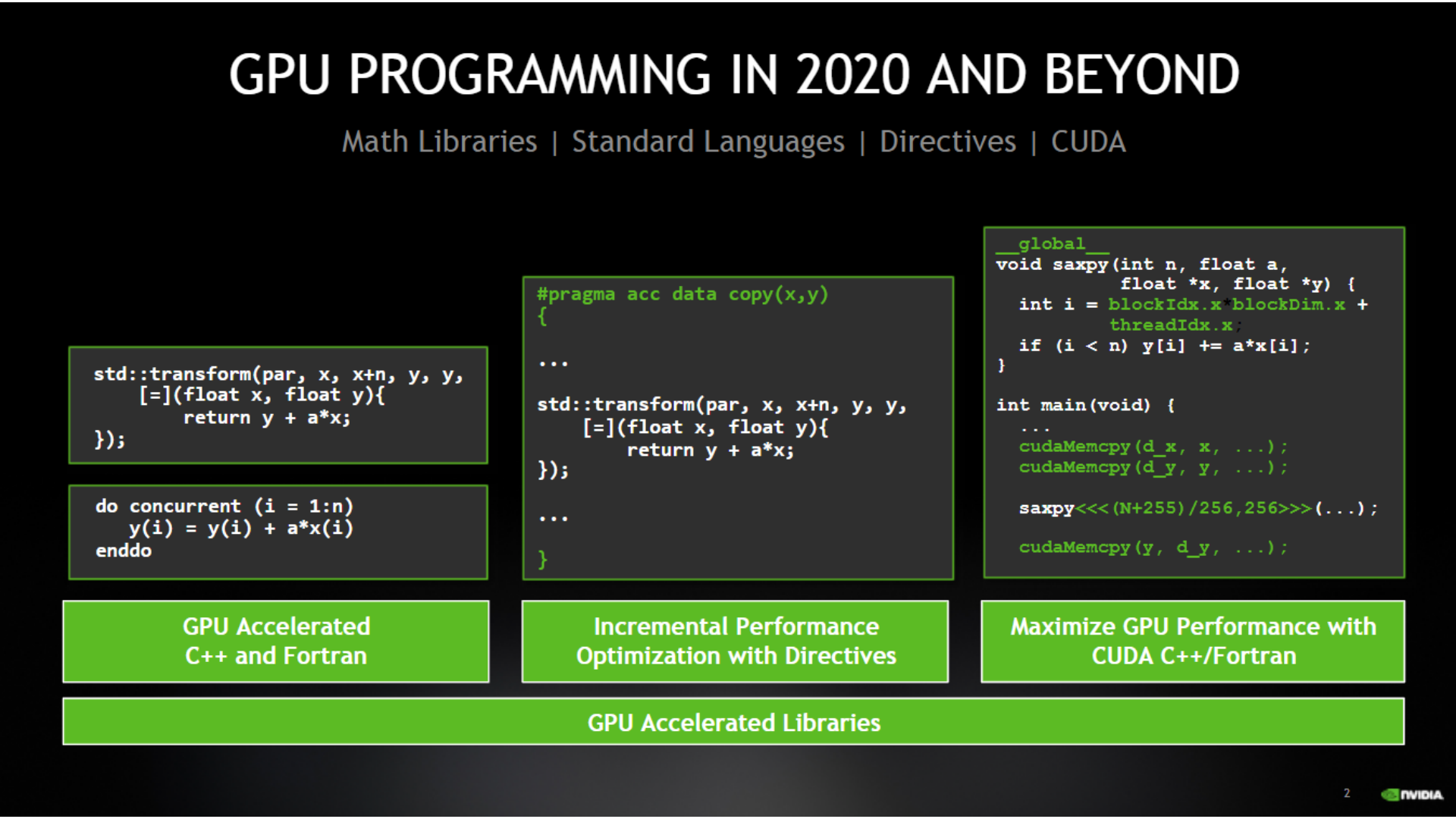
Roofline Model

- *Roofline: An Insightful Visual Performance Model for Multicore Architectures* (S. Williams, 2009, "Communications of the ACM", vol.52 n.4)

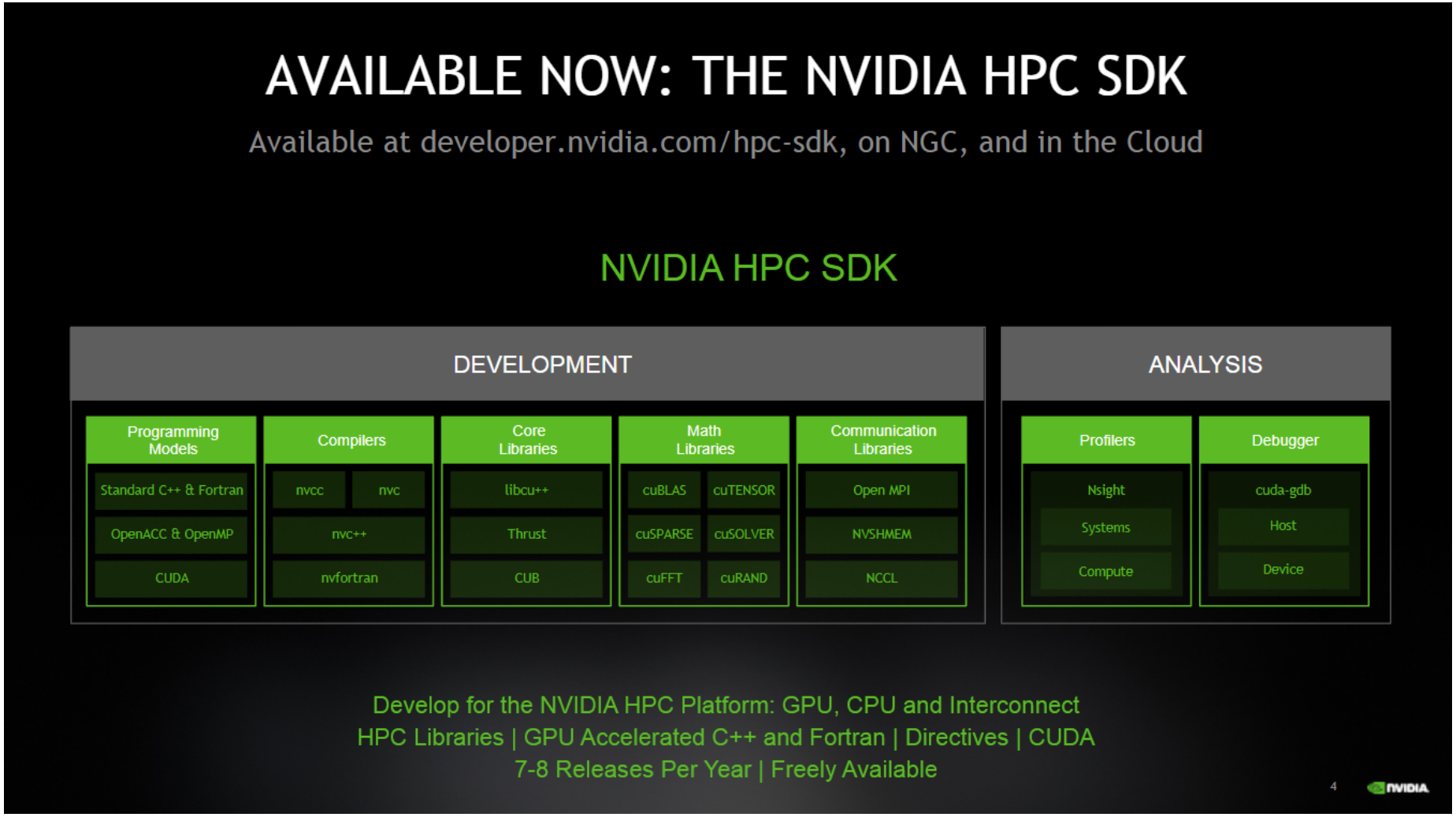


GTC 2020




GPU PROGRAMMING IN 2020 AND BEYOND
Math Libraries | Standard Languages | Directives | CUDA
std::transform(par, x, x+n, y, y,
[=](float x, float y){
return y + a*x;
});
do concurrent (i = 1:n)
y(i) = y(i) + a*x(i)
enddo
#pragma acc data copy(x,y)
{
...
std::transform(par, x, x+n, y, y,
[=](float x, float y){
return y + a*x;
});
...
}
__global__
void saxpy(int n, float a,
float *x, float *y) {
int i = blockIdx.x*blockDim.x +
threadIdx.x;
if (i < n) y[i] += a*x[i];
}
int main(void) {
...
cudaMemcpy(d_x, x, ...);
cudaMemcpy(d_y, y, ...);
saxpy<<<(N+255)/256,256>>>(...);
cudaMemcpy(y, d_y, ...);
GPU Accelerated
C++ and Fortran
Incremental Performance
Optimization with Directives
Maximize GPU Performance with
CUDA C++/Fortran
GPU Accelerated Libraries
2
NVIDIA

GTC 2020


AVAILABLE NOW: THE NVIDIA HPC SDK
Available at developer.nvidia.com/hpc-sdk, on NGC, and in the Cloud
NVIDIA HPC SDK
DEVELOPMENT
ANALYSIS
Programming Models
Standard C++ & Fortran
OpenACC & OpenMP
CUDA
Compilers
nvcc
nvc
nvc++
nvfortran
Core Libraries
libcu++
Thrust
CUB
Math Libraries
cuBLAS
cuTENSOR
cuSPARSE
cuSOLVER
cuFFT
cuRAND
Communication Libraries
Open MPI
NVSHMEM
NCCL
Profilers
Nsight Systems
Nsight Compute
Debugger
cuda-gdb
Host
Device
Develop for the NVIDIA HPC Platform: GPU, CPU and Interconnect
HPC Libraries | GPU Accelerated C++ and Fortran | Directives | CUDA
7-8 Releases Per Year | Freely Available
4
NVIDIA

Thanks

For more information please contact:
Georges-Emmanuel Moulard
M+ 33 6 85529054
georges-emmanuel.moulard@atos.net
Paul Karlshöfer paul.karlshoefer@atos.net

05/08/2020