LESSON 3

Throughput

TOPICS

- Divergence
- Intrinsinc functions
- Thread Level Parallelization (TLP)
- Instruction Level Parallelization (ILP)

- Key Words: *Divergence, Thread Level Parallelization, Instruction Level Parallelization.*

Execution configuration

- We have talked about the *execution configuration* that is used to configure the device resources in order to execute our application.
- The resources of the GPU are organized in streaming multiprocessors (SM); the quantity of SM depends on the hardware.
- The thread blocks in the GPU are executed according with the GigaThread global scheduler who is in charge of assigning them to each SM automatically, new blocks are assigned to the SMs as soon as the first ones have their own work done.
- As we saw in the device query practice, the hardware has limitations in threads and blocks that can be executed.

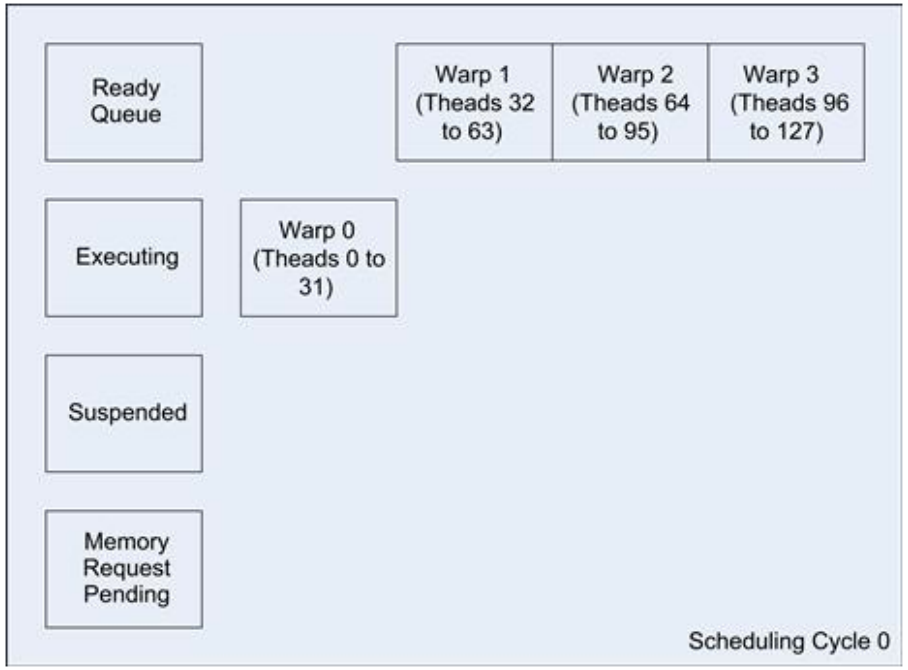
- Once a block is assigned to a SM, the multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*.

- The warp is the unit for scheduling purposes on the SMs; for example if a SM supports 1024 threads, we can say that 1024/32 = 32 warps.

- Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state, therefore free to branch and execute independently.

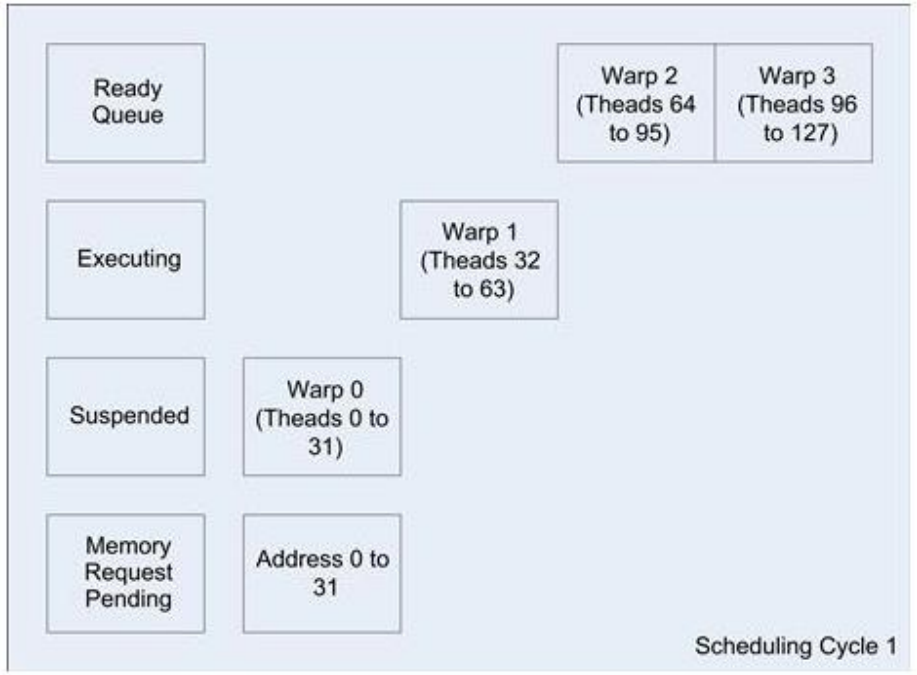
- A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path.

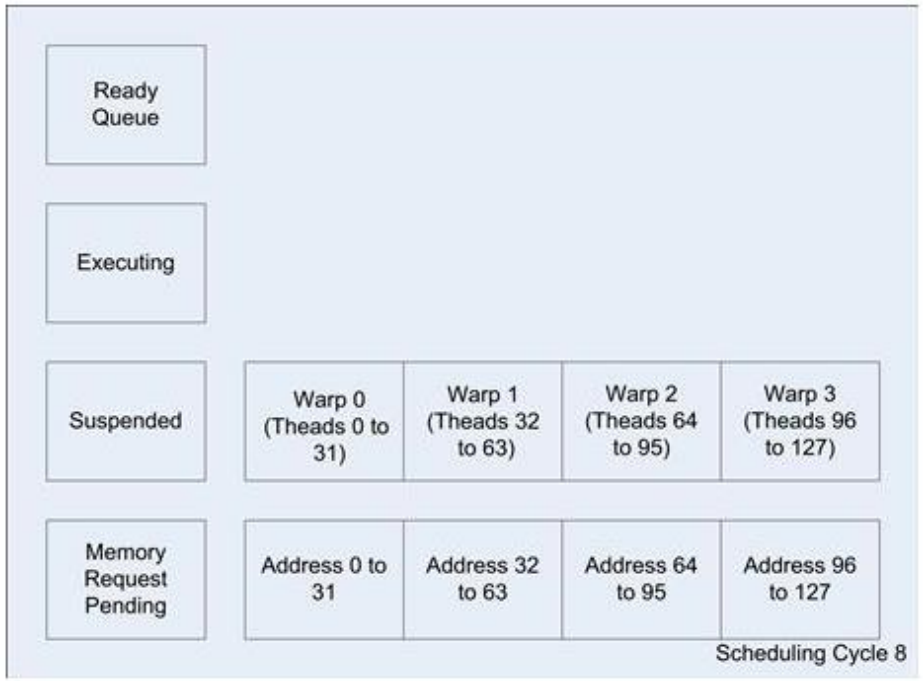
- Once a block is assigned to a SM,its threads are divided in warps —group of 32 threadseach one—. The warp is the unit for scheduling purposes on the SMs; for example if a SM supports 1024 threads, we can say that 1024/32 = 32 warps.



Cycle 0:

- The 128 threads are grouped in blocks of 32 threads.
- The first set, runs to extract the thread ID and to calculate the address in the array, and to issue a fetch memory request.
- The next instruction, requires both operands to be executed, the threads are suspended.
- While the first warp is suspended, the hardware switched to another warp.

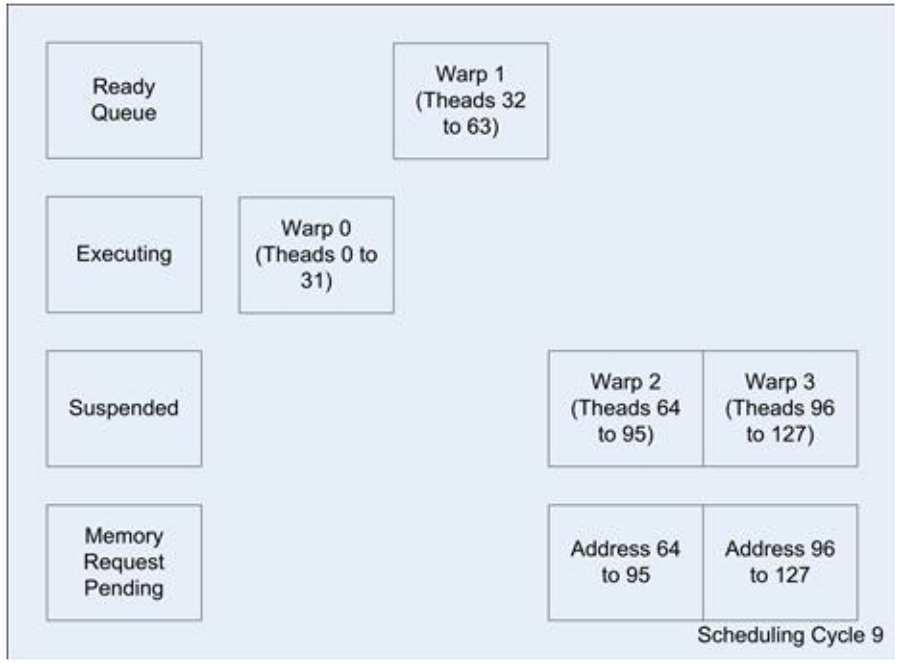




The GPU continues in this way until all the warps are moved to the suspend state.

- The memory fetch returns with the data for a whole group of threads, enough to enable a entire warp.

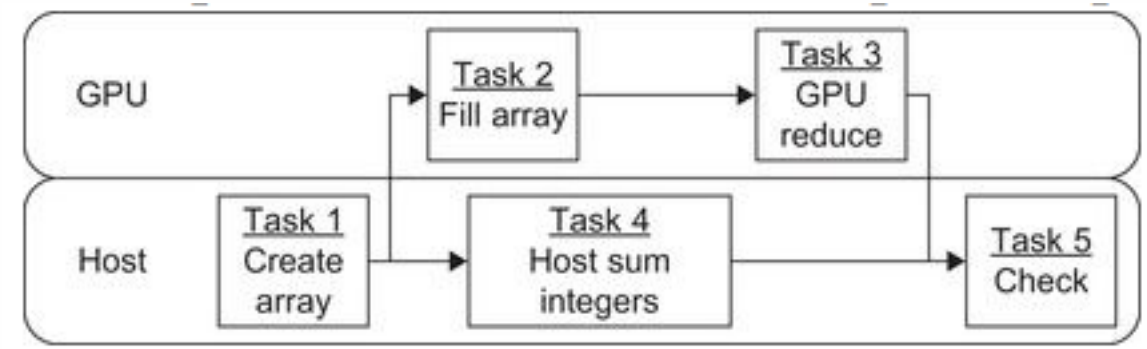
- These threads are placed in the ready state.

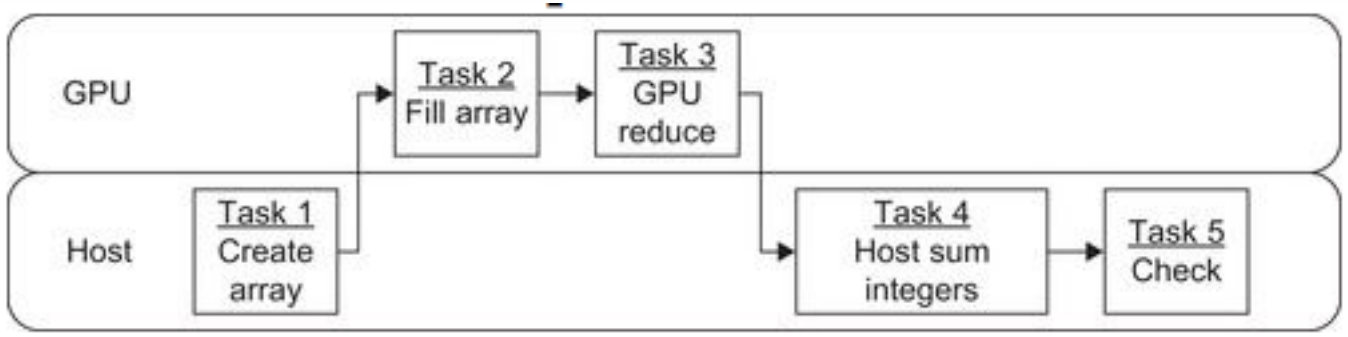


- Once warp 0 has been executed, it is completed and retired.

- After all the warps have been processed, the kernel is complete.

- *Kernels calls are asynchronous, meaning that the host queues a kernel for execution only in the GPU and does not wait for it to finish but rather continues on to perform me other work.*





Three rules of GPGPU programming

1. Get the data in the GPU and keep it there.

2. Give to the GPU enough work to do.

3. Focus in data reuse within he GPU to avoid memory bandwidth limitations.

Divergence

- Sometimes we need to execute an *if*, *switch*, *do*, or *while* statement, which imply that some threads are executing an instruction, but others are not. This is well known as divergence.

- If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.

- .

- Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths

- When there are different branches, they must be serialized; once the branches were executed, the threads converge.

- The number of blocks and warps that can reside and be processed together on the multiprocessor for a given kernel depends on the amount of registers and shared memory used by the kernel and the amount of registers and shared memory available on the multiprocessor.

Intrinsic functions

- To maximize instruction throughput, it is possible to use intrinsic functions as long as it does not affect the end result.
- The -use_fast_math compiler option of nvcc coerces every functionName() call to the equivalent __functionName() call.
- http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#intrinsic-functions

Operator/Function	Device Function
x/y	__fdividef(x,y)
sinf(x)	__sinf(x)
cosf(x)	__cosf(x)
tanf(x)	__tanf(x)
sincosf(x,sptr,cptr)	__sincosf(x,sptr,cptr)
logf(x)	__logf(x)
log2f(x)	__log2f(x)
log10f(x)	__log10f(x)
expf(x)	__expf(x)
exp10f(x)	__exp10f(x)
powf(x,y)	__powf(x,y)

Numerical Accuracy and Precision

- Devices of compute capability 1.3 and higher provide native support for double-precision floating-point values (that is, values 64 bits wide).

- Whenever doubles are used, use at least the -arch=sm_13 switch on the nvcc command line. The -arch compiler option specifies the compute capability that is assumed when compiling C to *PTX* code.

- So, code that contains double-precision arithmetic, must be compiled with -arch=sm_13 (or higher compute capability), otherwise double-precision arithmetic will get demoted to single-precision arithmetic.

- The order in which arithmetic operations are performed is important. If A, B, and C are floating-point values, (A+B)+C is not guaranteed to equal A+(B+C) as it is in symbolic math.

- If we have the next code segment:

```
float a;
a = a*1.02;
```

- If the code were performed on the host, the literal 1.02 would be interpreted as a double-precision quantity and a would be promoted to a double, the multiplication would be performed in double precision, and the result would be truncated to a float —thereby yielding a slightly different result.

- If, however, the literal 1.02 were replaced with 1.02f, the result would be the same in all cases because no promotion to doubles would occur.

- To ensure that computations use single-precision arithmetic, always use float literals.

- Integer division and modulo operations are particularly costly and should be avoided or replaced with bitwise operations whenever possible: If *n* is a power of 2, (*i* / *n*) is equivalent to (*i* ≫ *log2 n*) and (*i* % *n*) is equivalent to (*i* & *n* - 1).

- Loop counters must be declared as signed in order to allow the compiler to use optimizations where the overflow semantics are undefined.

Thread Level Parallelization (TLP)

- TLP is based on hiding ALU and memory latencies to keep the execution units working.
- When the warps have all the data dependences and resources resolved, the SM schedule is able to select another warp to be executed, if there are unresolved dependencies, the next instruction cannot be issued.
- The main point behind TLP is to give the scheduler as many threads as possible in order to keep them working.
- To have a high occupancy, the SM scheduler must have many warps to choose, which increase the likelihood to have one wrap with all the dependencies resolved.

- The occupancy is measured as the number of warps running concurrently on a multiprocessor divided by the maximum number of warps that can reside on a SM.

- In order to apply this, there are some points to be considered:

  - Nvidia provides the occupancy calculator to help choosing the execution configuration due to the fact that there are a lot of GPUs with different hardware.

  - Configure more blocks than SM, a multiple of the number of SM, to be sure that there is at least one block to execute.

  - In some GPUs architectures there could be executed concurrent kernels, in order to increase application performance using free SM.

  - To have the blocks busy, the __syncthreads function should be averted in order to keep the hardware waiting.

Instruction Level Parallelization

- ILP can hide latency by keeping the SIMD cores busy; it also has enough transactions on the flight to saturate the memory bus, using fewer threads that consume fewer resources.

- The more threads working in a block, the less registers that can be used per thread, and the registers are the most valuable resource; it is the memory that can achieve the maximum peak GPU performance.

- Using registers we can avoid the use of shared memory, which implies fewer accesses to read it or write it.

- In some experimentation, using more operation per thread increases the performance [1].

- For example, if we have a warp of 32 threads, and in our kernel we have 2 or more independent operations to be executed. It is possible, due to the warp scheduling, to execute instructions in parallel. If we start to create more and more warps, we will get in some time the maximum performance with low occupancy.

[1] Vasily Volkov (2010). *Use registers and multiple outputs per thread on GPU. UC Berkeley, http://www.eecs.berkeley.edu/~volkov/volkov10-PMAA.pdf*

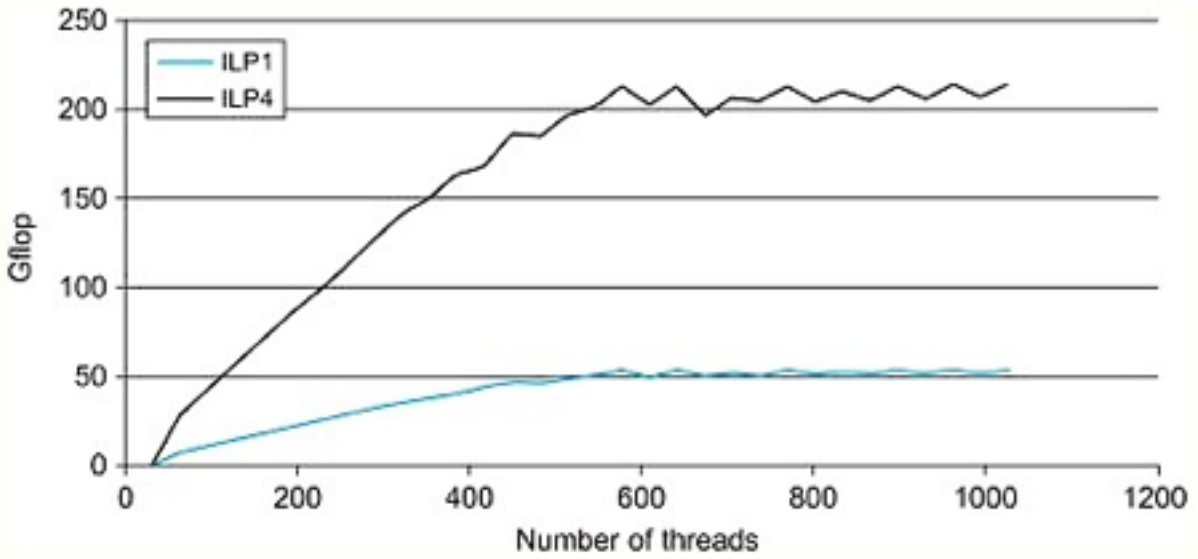
```
register float d=a, e=a, f=a;
#pragma unroll 16
 for(int i=0; i < NUM_ITERATIONS; i++) {
   a = a * b + c;
   d = d * b + c;
   e = e * b + c;
   f = f * b + c;
 }
```

- The #pragma unroll directive however can be used to control unrolling of any given loop. It must be placed immediately before the loop and only applies to that loop. It is optionally followed by a number that specifies how many times the loop must beunrolled.

- The compiler will also insert code to ensure correctness, to ensure that there will only be n iterations, if n is less than 16, for example.

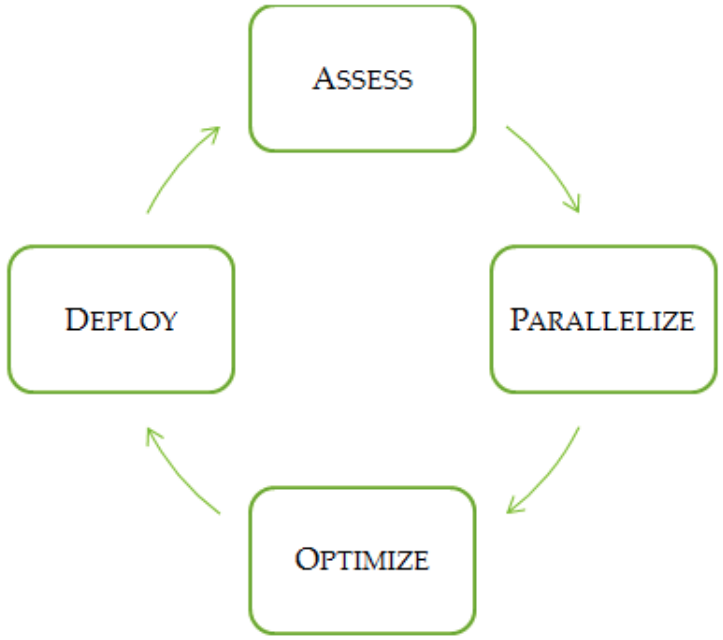
```
#pragma unroll 16
 for(int i=0; i < NUM_ITERATIONS; i++) {
   a = a * b + c;
 }
```

- The code was executed on a device that supports 1024 threads per block. The number of warps were increased from 32 to 1024, obtaining the figure shown below. The first peak was obtained with 512 threads, which is the half of the utter occupancy.



Assess, Parallelize, Optimize, Deploy (APOD)

- Design cycle for applications with the goal of helping application developers to rapidly identify the portions of their code that would most readily benefit from GPU acceleration, rapidly realize that benefit, and begin leveraging the resulting speedups in production as early as possible.



- Assess

- Evaluate the application to locate the parts of the code that are responsible for the bulk of the execution time.

- Armed with this knowledge, the developer can evaluate these bottlenecks for parallelization and start to investigate GPU acceleration.

- Parallelize

- Having identified the hotspots and having done the basic exercises to set goals and expectations, the developer needs to parallelize the code.

- Some applications' designs will require some amount of refactoring to expose their inherent parallelism

- Optimize

- After each round of application parallelization is complete, the developer can move to optimizing the implementation to improve performance.

- Since there are many possible optimizations that can be considered, having a good understanding of the needs of the application can help to make the process as smooth as possible.

- The available profiling tools are invaluable for guiding this process, as they can help suggest a next-best course of action for the developer's optimization efforts and provide references into the relevant portions of the optimization section of this guide.

- Deploy

- Having completed the GPU acceleration of one or more components of the application it is possible to compare the outcome with the original expectation.

Assess, Parallelize, Optimize, Deploy Example

- In order to do a complete example of an application we are going to use the transpose matrix operation to get the best implementation using all the topics already seen.

Serial code on the Host

```
//Host function
void transposedMatrixHost(Matrix<int> d_a, Matrix<int> d_b){
    // start timer
    CpuTimer timer;
    timer.Start();

    int i, j;

    for(i = 0; i < d_a.width; i++){
        for(j = 0; j < d_a.height; j++){
            d_b.setElement( i, j, d_a.getElement(j, i) );
        }
    }
    // stop timer
    timer.Stop();

    // print time
    printf( "Time Host:  %f ms\n", timer.Elapsed() );

}
```

- Serial code on the Device (1 thread, 1 block)

```
// Kernel v1 using 1 thread and 1 block
__global__ void transposedMatrixKernel(Matrix<int> d_a, Matrix<int> d_b){

    int i = 0;
    int j = 0;

    while( i < d_a.width){
        j = 0;
        while( j < d_a.height){
            d_b.setElement( i, j, d_a.getElement(j, i) );
            j++;
        }
        i++;
    }

}
```

- Parallel code (1 block, N thredas)

```
// Kernel v2 using the max number of threads in 1 block
__global__ void transposedMatrixKernel_threads(Matrix<int> d_a, Matrix<int> d_b, int THREADS){

    int i = threadIdx.x;
    int j = 0;

    while (i < N){
        while( j < N ){
            d_b.setElement( i, j, d_a.getElement(j, i) );
            j++;
        }
        i += THREADS;
    }
}
```

- Parallel code (K threads, N/K blocks)

```
// Kernel v3 using K threads and N/K blocks
// Try this example with 8, 16 and 32 threads by block
__global__ void transposedMatrixKernel_threads_blocks(Matrix<int> d_a, Matrix<int> d_b){


    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int j = threadIdx.y + blockDim.y * blockIdx.y;

    d_b.setElement( i, j, d_a.getElement(j, i) );

}
```

Memory Bandwidth

- Memory bandwidth is the rate at which data can be read from or stored into a memory sistem.

- Theoretical Bandwidth

- Theoretical bandwidth can be calculated using hardware specifications available in the product literature.

$$TBw = (((Clock\ rate * 10^2) * (memory\ bus\ width / 8)*2)) / 10^9$$

- Effective Bandwidth Calculation

- Effective bandwidth is calculated by timing specific program activities and by knowing how data is accessed by the program.

$$EBw = ((Br + Bw)[Gb] / time [s])$$

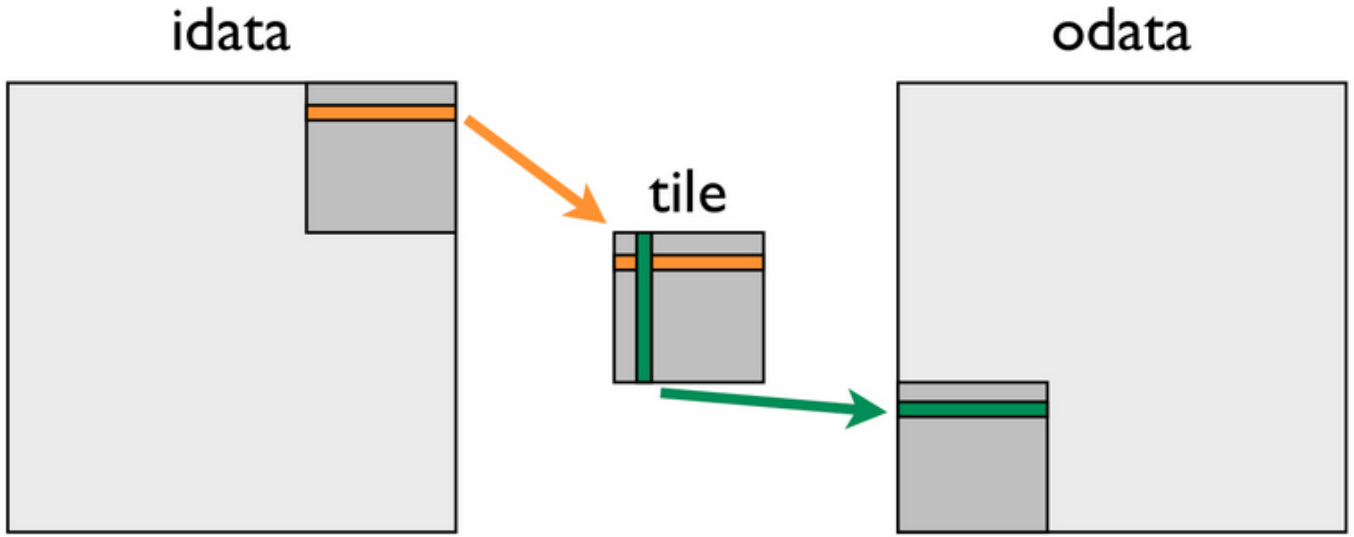
- B_r is the number of bytes read per kernel, B_w is the number of bytes written per kernel, and time is given in seconds.

Example

- 01matrix_transpose: This example shows different impletentation of the transpose operation and the calculos of the bandwidth.

Tiling

- As we saw in the shared memory examples, this technique is used to execute the operations in the cache memory, avoiding the reading and writing operations on the global memory many times.



Example

- 02matrix_transpose_tile: This examples shows the use of tiles in the shared memory in order to reduce the operations over the global memory.

Practice

- 03matrix_transpose: Complete the code to use big matrices for the transpose operation.