Performance issues

**Memory Coalescing**: When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads.

GPU cache lines are of size 128bytes each, and they are aligned. Coalesced memory access or memory coalescing refers to combining multiple memory accesses into a single transaction. 128bytes of data refer to 32 integer or 32 single precision Floating Point (FP), elements. Thus, a warp (32 consecutive threads) can access 32 elements in a single memory transaction. This is the best-case scenario, and this is not always possible.

**Utilizing Shared memory**: Shared memory consists of 32 banks of width 4 bytes. Element ‘i’ is in bank ‘i % 32’. A bank conflict occurs when 2 threads in a warp access different elements in the same bank. Successive 4byte words are assigned to successive banks. Bank conflicts cause serial memory accesses (poor performance).

**Choosing the number of threads in a block:** The number of threads per block should be chosen as a multiple of the warp size to avoid wasting computing resources with under-populated warps as much as possible. **If the number of threads per block is not a multiple of 32, then not all the GPU cores are used at all times and thus performance is degraded.**

**Occupancy:** In an NVIDIA GPU, the basic unit of execution is the warp. A warp is a collection of threads, 32 in current implementations, that are executed simultaneously by an SM. There is a maximum number of warps which can be concurrently active on a Streaming Multiprocessor (SM). **Occupancy is defined as the ratio of active warps on an SM to the maximum number of active warps supported** by the SM. Occupancy varies over time as warps begin and end, and can be different for each SM. <https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm>

Each SM manages the creation, execution, synchronization and destruction of concurrent threads in hardware, with zero scheduling overhead, and this is one of the key factors in achieving high performance. Each parallel thread is mapped to a core for execution, and each thread maintains it’s own register state. The SM creates and manages threads in groups of 32, and each such group is called a warp. A warp is the smallest unit of scheduling within each SM. The GPU achieves efficiency by splitting it’s work-load into multiple warps and multiplexing many warps onto the same SM. When a warp that is scheduled attempts to execute an instruction whose operands are not ready (due to an incomplete memory load, for example), the SM switches context to another warp that is ready to execute, thereby hiding the latency of slow operations such as memory loads.

1st example Matrix Transpose

**Task1**: Try to implement in CUDA a loop kernel that reads a square 2d array and generates its transpose. The serial code is shown below; this program is very similar to the example you did previous week (matrix addition).

*For (m=0;m<N;m++)*

*For (k=0;k<N;k++)*

*Atranspose[m][k]=A[k][m];*

In ‘transpose\_comp3001.cu’ file, there are three different implementations of this algorithm. In transpose\_ver1(), each thread copies just one element, while in the transpose\_ver4() routine each thread copies multiple elements. The latter is a more efficient strategy.

The ‘if (i < N && j < N)’ is not needed, but it will avoid incorrect results. If you are sure that the above command is not needed, then delete it to improve performance.

Fig.1 and Fig.2, visualize the process of transpose\_ver1() and transpose\_ver4(), respectively. In transpose\_ver1(), a block of threads consists of 1024 threads. All the threads of a block will be executed in the same SM. One SM will process multiple blocks (different every time).

*// dim3 dimBlock(32, 32, 1);*

*//dim3 dimGrid( N / 32, N / 32, 1);*

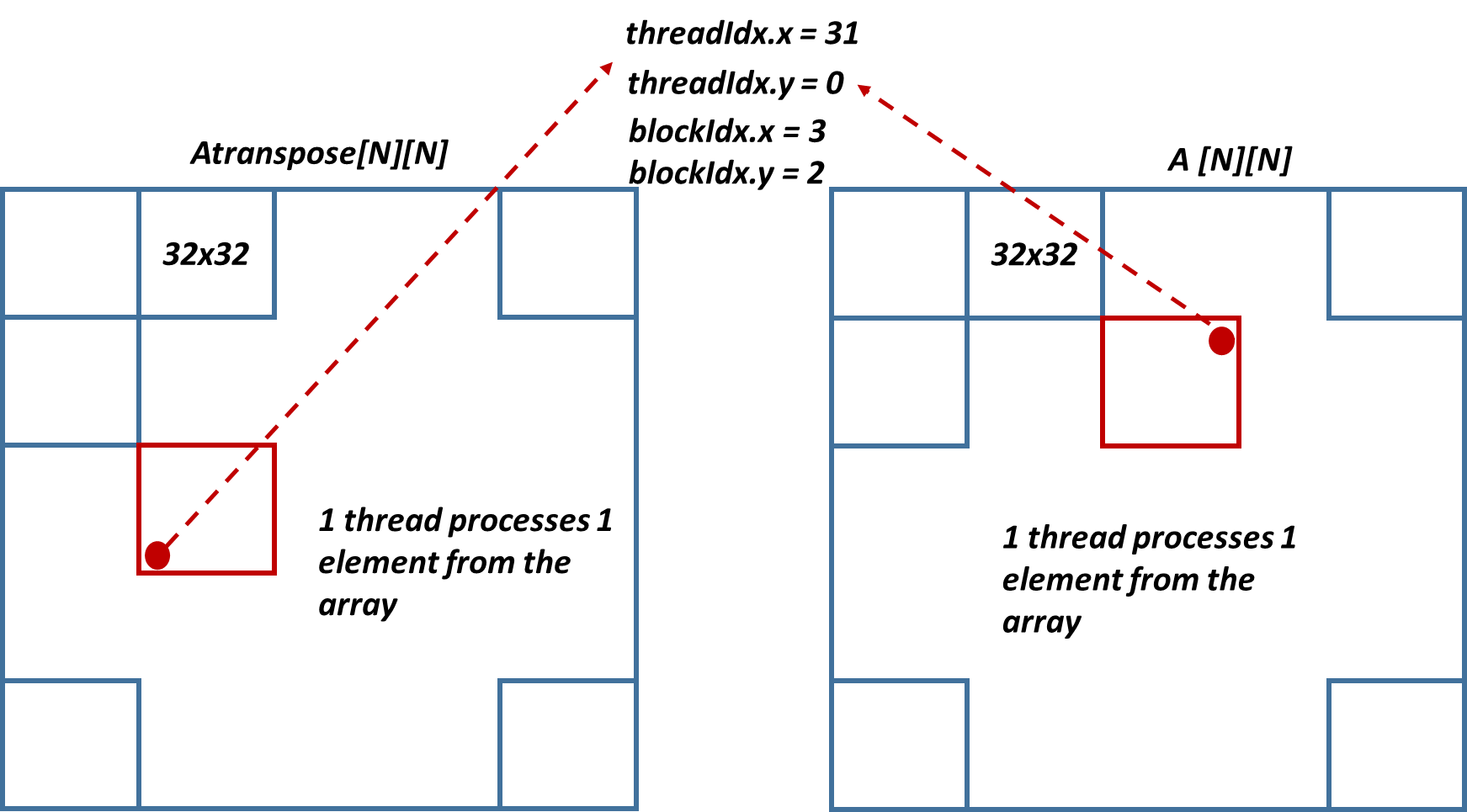
*\_\_global\_\_ void transpose\_ver1() {*

*int i = blockIdx.x \* blockDim.x + threadIdx.x;*

*int j = blockIdx.y \* blockDim.y + threadIdx.y;*

*device\_Atranspose[i][j] = device\_A[j][i];*

*}*

**

*Fig.1 Visual representation of* transpose\_ver1() routine

In transpose\_ver4(), each thread processes 4 elements. The tile size is still 32x32, but 32x8 threads process it.

*// dim3 dimBlock(32, 8, 1);*

*//dim3 dimGrid( N / 32, N / 32, 1);*

*\_\_global\_\_ void transpose\_ver4() {*

*int x = blockIdx.x \* 32 + threadIdx.x;*

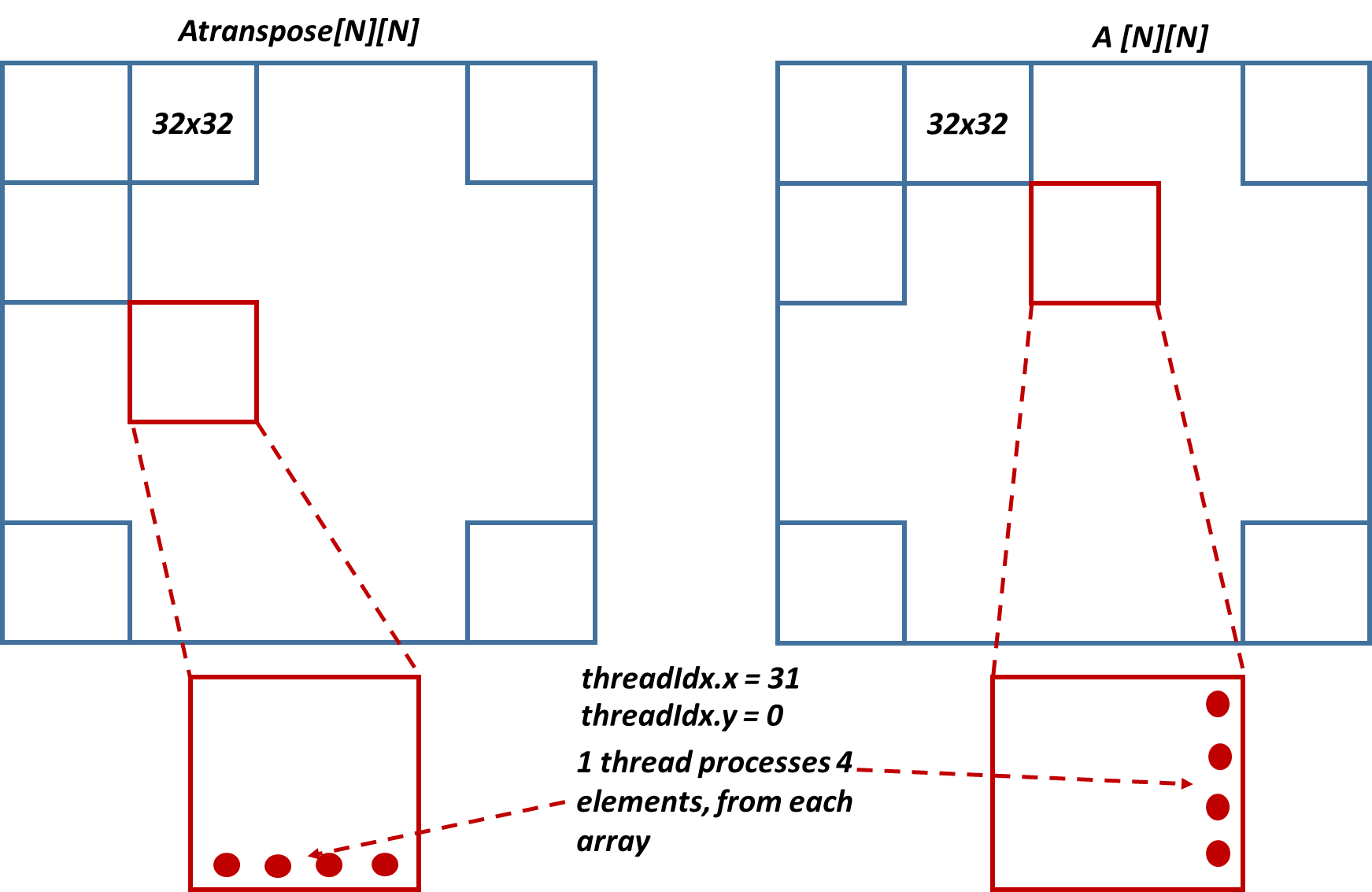
*int y = blockIdx.y \* 32 + threadIdx.y;*

*for (int m = 0; m < 32; m += 8) {*

*device\_Atranspose[x][y+m] = device\_A[y + m][x];*

*}*

*}*

**

*Fig.2. Visual representation of* transpose\_ver4() routine

2nd example – Matrix-Matrix Multiplication

#pragma omp is used to specify Directives and Clauses. If /openmp isn't specified in a compilation, the compiler ignores OpenMP clauses and directives. OpenMP Function calls are processed by the compiler even if /openmp isn't specified. Visual Studio 2019 also now offers SIMD functionality. To use SIMD, compile by using the /openmp:experimental option. This option enables both the usual OpenMP features, and additional OpenMP SIMD features not available when using the /openmp switch. So, select ‘project properties’, ‘configuration properties’, ‘CUDA C/C++’, ‘host’, ‘additional compiler options’ and then write: ‘/openmp:experimental’.