

 King Fahd University of Petroleum and Minerals

 Department of Computer Engineering

[Parallel and Vector Architectures](http://faculty.kfupm.edu.sa/coe/mayez/ps-coe502/coe502.html) CSE 661

[Parallel Processing Architectures COE 502](http://faculty.kfupm.edu.sa/coe/mayez/ps-coe502/coe502.html)

Dr. Mayez Al-Mouhamed

**Some CUDA Programming Rules**

**Table of Contents**

 [1](#_Toc382238113)

[King Fahd University of Petroleum and Minerals 1](#_Toc382238114)

[Department of Computer Engineering 1](#_Toc382238115)

[Rule 1: 1-D Linear GPU Global Memory 3](#_Toc382238116)

[Rule 2: A Kernel is a Block of Threads 3](#_Toc382238117)

[Kernel = {Workers each worker is a thread} = {Threads} 3](#_Toc382238118)

[2.1. Using 2-D kernel 3](#_Toc382238119)

[2.2. Using 1-D kernel 4](#_Toc382238120)

[2.3. Using 1-D, 2-D, or 3-D kernels 4](#_Toc382238121)

[Rule 3: Tiling 4](#_Toc382238122)

[3.1. Tiling the rows 5](#_Toc382238123)

[3.2. Tiling the columns 5](#_Toc382238124)

[Rule 4: The Different Possibilities of Tiling 5](#_Toc382238125)

[Rule 5: Global memory is parallel and interleaved 8](#_Toc382238126)

[Rule 6: Take advantage of data locality 8](#_Toc382238127)

[6.1. Loop on rows (not coalesced) 8](#_Toc382238128)

[6.2. Loop on columns (coalesced) 8](#_Toc382238129)

[6.3. Loop on rows and columns (coalesced) 9](#_Toc382238130)

[6.4. Loop on rows and columns (not coalesced) 9](#_Toc382238131)

1. 1-D Linear GPU Global Memory

The global memory GM in GPU is linear of one dimension. Therefore, multi-dimensional arrays indexing have to be mapped to one dimension index. For example, consider a 2-D array A with *ASIZE* column and *ASIZE* rows. Then, to access an element *A*[*j*, *i*], where *j* is the row number and *i* is the column number. Assuming a row-major storage, the effective address of a 1-D GM is given by: *EA* = *ASIZE* × *j* + *i*.

|  |
| --- |
| **r1****r2****r3****r1****r2****r3****r4****r5****⇒****Two Dimensional Array A[j, i]****Global Memory 1-D** |
| Figure 1 : Row-Major Storrage of a 2-D array onto a 1-D GM memory where each row is stored in contiguous memory locations, one row at a time, one row after. In other words, the other : the GM effective Address of *A*[*j*, *i*] is *j* × *ASIZE* + *i* |

1. A Kernel is a Block of Threads

 Kernel = {Workers each worker is a thread} = {Threads}

To ease the mapping of threads to results, the dimension of the kernel is to be mapped to the dimension of parallel processing results (each thread is to compute one result).

* 1. Using 2-D kernel

If we have a 2-D array of results we may use a 2-D Kernel such that each thread will compute one result: *result*(*j*, *i*) is mapped to thread *th*(*j*, *i*). Each thread must have IDs like its block number and its offset within a block (Th(Bkj, thj) ).

|  |
| --- |
| ***i*****Result*****j*****Data Parallel***thj, i***Kernel** |
| Figure 2 : 2-D array of results and 2-D kernel:One thread maps to one result. |

* 1. Using 1-D kernel

Assume a 2-D array of results and 1-D kernel. To increase the grain size of the thread (amount of work by one thread), we may set use a 1-D kernel in which each thread will work on one row of results or one column of results. e.g. if the dimension of the array is greater than the dimension of kernel, each thread will locally run on all the un-mapped dimension.

|  |
| --- |
| **Result****Data Parallel*****th0******th1******th2******…*****Kernel****Result****Data Parallel** ***th0 th1 th2  …*****Kernel****(a) One thread will work on one Column of results****(b) One thread will work on one Row of results** |
| Figure 3 : A 2-D array of results and 1-D kernel: each column (or row) in the result is mapped to one thread. |

* 1. Using 1-D, 2-D, or 3-D kernels

A kernel can have up to 3-dimensions in GPU/CUDA.

1. Tiling

A GPU is a collection of Streaming Multiprocessors (SMs), each SM has a small Shared-Memory (ShM). Tiling aims at modifying the loop structure to adapt to the small ShM size. For example, a double nested loop referencing a 2-D array cannot load all the array data in ShM (ShM: 64 KB)) of an SM due to the limitation of ShM shared memory. Therefore, one way to process the array is to load a tile of array data, do processing, and then load next tile, and so on. For this we need to tile the original loop. HERE.

Tiling consists of subdividing the rows (or columns) into tiles of rows (or tile of columns).

|  |
| --- |
| ***TPB*****Result***Bk0**Bk1**…**Bkn* |
| Figure 4 : Tiling: subdividing rows into groups of rows. |

* 1. Tiling the rows

Tiling on j

$$j=Bk\_{j}\*TPB+th\_{j}$$

where:

$$0\leq th\_{j}\leq TPB-1$$

$$0\leq Bk\_{j}\leq \frac{ASIZE}{TPB}-1$$

$$Bk\_{j}=\left⌊\frac{j}{TPB}\right⌋$$

*TPB* = **T**hreads **P**er **B**lock

*Bk* = Block id

*th* = Thread id

For example:

If *ASIZE* = 2048 ⇒ *ASIZE* = 211 = 25 \* 26 ⇒ 25 threads \* 26 Blocks ⇒ *TPB* = 25 = 32

* 1. Tiling the columns

Tiling on i

$$i=Bk\_{i}\*TPB+th\_{i}$$

1. The Different Possibilities of Tiling

 $EA=j×ASIZE+i$

1. For one-dimensional kernel:

|  |
| --- |
| *th*BK × TPB$$\{Bk×TPB+th\}$$ |
| Figure 5 : Tiling a one-dimensional kernel. |

* 1. Tile i:

Every thread will do all of the new loop:

$$L\_{(0\leq i\leq ASIZE-1)}$$

 $j×ASIZE+i$

$⇓$

$(BK\_{j}×TPB+th\_{j})×ASIZE+i$

* 1. Tile j:

$$L\_{(0\leq j\leq ASIZE-1)}$$

 $j×ASIZE+i$

$⇓$

$j×ASIZE+(Bk\_{i}\*TPB+th\_{i})$

1. For two-dimensional kernel:

The kernel is a set of blocks

|  |
| --- |
| $$\{Bk\_{y}×TPB+th\_{y} , Bk\_{x}×TPB+th\_{x}\}$$*TPB***Kernel***Bky* × *TPB**Bkx* × *TPB**thy**thx* |
| Figure 6 : Tiling a two-dimensional kernel. The kernel is a set of blocks. |

 $j×ASIZE+i$

$⇓$

$(Bk\_{y}×TPB+th\_{y})×ASIZE+Bk\_{x}×TPB+th\_{x}$

1. Tiling of both rows and columns.

Tiling of both dimensions using one-dimensional kernel.

Each thread is responsible of *ASIZE* results. It does *TPB* elements per iteration.

Two loops in each thread body $L\_{jj}$ and $L\_{II}$

 $j×ASIZE+i$

$⇓$

$\left(Bk\_{j}×TPB+jj\right)×ASIZE+II×TPB+th\_{j}$

*while:*

$0\leq jj\leq TPB-1$

$0\leq II\leq Bk\_{n}$

*Interpretation: thread (Bkj, thj) will do:*

|  |
| --- |
| **Kernel**$$BK\_{j}×TPB$$*jj*$$II×TPB$$*thi**thi**thi**jj*$$j×ASIZE+i$$$$⇓$$$$\left(Bk\_{j}×TPB+jj\right)×ASIZE+II×TPB+th\_{j}$$$$while:$$$$0\leq jj\leq TPB-1$$$$0\leq II\leq Bk\_{n}$$ |
| Figure 7 : Tiling a two-dimensional kernel. Tiling on both (rows and columns). |

 $j×ASIZE+i$

$⇓$

$\left(JJ×TPB+th\_{j}\right)×ASIZE+Bk\_{j}×TPB+ii$

*while:*

$0\leq ii\leq TPB-1$

$0\leq JJ\leq Bk\_{n}$

|  |
| --- |
| $$j×ASIZE+i$$$$⇓$$$$\left(JJ×TPB+th\_{j}\right)×ASIZE+Bk\_{j}×TPB+ii$$$$while:$$$$0\leq JJ\leq \frac{ASIZE}{TPB}-1$$$$0\leq ii\leq TPB-1$$**Kernel**$$JJ×TPB$$$$BK\_{i}×TPB$$*ii**thi* |
| Figure 8 : Tiling a two-dimensional kernel. Tiling on both (rows and columns). |

1. Global memory is parallel and interleaved

Design threads in such a way they access successive data.

*Interleaved memory*: Global memory consists of *N* parallel memories. The data are interleaved among these memories. The first element is stored in M1, second element is stored in M2, and so on. The last *n*-bits (*n*=*log2* *N*) of an effective address (EA) represents the index of the memory that stores the data.

For example: if there are four memories, then the memory index is the last two bits (*n* = *log2* 4). To access the effective address is 0x0036, the data is located in the second memory on the address 0x000D. (0x0036 = 0b0000\_0000\_0011\_0110).

*Coalesced memory access*: If we design threads in such a way, they access successive data.

1. Take advantage of data locality

Move the data to the shared memory.

Use TPB = 32 ⇒ one warp.

There are four possibilities for partitioning the 2D-result

* 1. Loop on rows (not coalesced)

The thread program contains one loop $L\_{i}$.

$$EA=(Bk\_{j}×TPB+th\_{j})×ASIZE+i$$

where $0\leq th\_{j}\leq TPB-1$ and $ 0\leq Bk\_{j}\leq \frac{ASIZE}{TPB}-1$

The difference in addressing from a thread to another is equal to *ASIZE* × (*thj1* - *thj2*). So memory accessing is **not coalesced** in this case.

* 1. Loop on columns (coalesced)

The thread program contains one loop $L\_{j}$.

$$EA=j×ASIZE+Bk\_{i}×TPB+th\_{i}$$

where $0\leq th\_{i}\leq TPB-1$ and $ 0\leq Bk\_{i}\leq \frac{ASIZE}{TPB}-1$

The difference in addressing from a thread to another is equal to (*thi1* – *thi2*). So memory accessing is **coalesced** in this case.

* 1. Loop on rows and columns (coalesced)

The thread program contains two nested loops: $L\_{0\leq jj\leq TPB-1}$ and $L\_{0\leq II\leq \frac{ASIZE}{TPB}-1}$

$$EA=(Bk\_{j}×TPB+jj)×ASIZE+II×TPB+th\_{i}$$

where $0\leq th\_{i}\leq TPB-1$ and $ 0\leq Bk\_{j}\leq \frac{ASIZE}{TPB}-1$

The difference in addressing from a thread to another is equal to (*thi1* - *thi2*). So memory accessing is **coalesced** in this case.

* 1. Loop on rows and columns (not coalesced)

The thread program contains two nested loops: $L\_{0\leq JJ\leq \frac{ASIZE}{TPB}-1}$ and $L\_{0\leq ii\leq TPB-1}$

$$\left(jj×TPB+th\_{i}\right)×ASIZE+Bk\_{i}×TPB+ii$$

where $0\leq th\_{i}\leq TPB-1$ and $ 0\leq Bk\_{i}\leq \frac{ASIZE}{TPB}-1$

The difference in addressing from a thread to another is equal to *ASIZE* × (*thj1* - *thj2*). So memory accessing is **coalesced** in this case.