

GPU Programming

Rupesh Nasre.

IIT Madras
January 2022

Agenda

- **Computation**
- Memory
- Synchronization
- Functions
- Support
- Streams
- Topics
- Case Study – Graphs

Hello World.

```
#include <stdio.h>

int main() {
    printf("Hello World.\n");
    return 0;
}
```

Compile: nvcc hello.cu
Run: a.out

GPU Hello World.

Kernel

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel() {
    printf("Hello World.\n");
}
```

Kernel Launch



```
dkernel<<<1, 1>>>();
return 0;
}
```

Compile: nvcc hello.cu

Run: ./a.out

– No output. --

GPU Hello World.

```
#include <stdio.h>

#include <cuda.h>

__global__ void dkernel() {
    printf("Hello World.\n");
}

int main() {
    dkernel<<<1, 1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Compile: nvcc hello.cu

Run: ./a.out

Hello World.

Takeaway

CPU function
and GPU kernel
run asynchronously.

GPU Hello World.

```
#include <stdio.h>

#include <cuda.h>

__global__ void dkernel() {
    printf("Hello World.\n");
}

int main() {
    dkernel<<<1, 1>>>();
    dkernel<<<1, 1>>>();
    dkernel<<<1, 1>>>();
    cudaDeviceSynchronize();
    printf("on CPU\n");
    return 0;
}
```

Takeaway

Kernels (by default) are executed one after another.

CPU launches them and moves ahead.

CPU waits at CDS.

Homework

```
__global__ void dkernel() {  
    printf("Hello World.\n");  
}  
  
int main() {  
    dkernel<<<1, 1>>>();  
    printf("CPU one\n");  
    dkernel<<<1, 1>>>();  
    printf("CPU two\n");  
    dkernel<<<1, 1>>>();  
    printf("CPU three\n");  
    cudaDeviceSynchronize();  
    printf("on CPU\n");  
    return 0;  
}
```

Identify which printf's can execute in parallel.

Homework

- Find out where *nvcc* is.
- Find out the CUDA version.
- Find out where *deviceQuery* is.

GPU Hello World in Parallel.

```
#include <stdio.h>
#include <cuda.h>

__global__ void dkernel() {
    printf("Hello World.\n");
}

int main() {
    dkernel<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Compile: nvcc hello.cu

Run: ./a.out

Hello World.

Hello World.

...

32 times {

Parallel Programming Concepts

- **Process:** a.out, notepad, chrome
- **Thread:** light-weight process
- **Operating system:** Windows, Android, Linux
 - OS is a software, but it manages the hardware.
- **Hardware**
 - Cache, memory
 - Cores
- **Core**
 - Threads run on cores.
 - A thread may jump from one core to another.

Classwork

Can this be made parallel?

- Write a CUDA code corresponding to the following sequential C code.

```
#include <stdio.h>
#define N 100
int main() {
    int i;
    for (i = 0; i < N; ++i)
        printf("%d\n", i * i);
    return 0;
}
```

```
#include <cuda.h>
#define N 100
__global__ void fun() {
    for (int i = 0; i < N; ++i)
        printf("%d\n", i * i);
}
int main() {
    fun<<<1, 1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Classwork

- Write a CUDA code corresponding to the following sequential C code.

```
#include <stdio.h>
#define N 100
int main() {
    int i;
    for (i = 0; i < N; ++i)
        printf("%d\n", i * i);
    return 0;
}
```

```
#include <cuda.h>
#define N 100
__global__ void fun() {
    printf("%d\n", threadIdx.x *
           threadIdx.x);
}
int main() {
    fun<<<1, N>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Note that there is no loop here.

Classwork

- Write a CUDA code corresponding to the following sequential C code.

```
#include <stdio.h>

#define N 100

int main() {
    int a[N], i;
    for (i = 0; i < N; ++i)
        a[i] = i * i;
    return 0;
}
```

```
#include <stdio.h>
#include <cuda.h>
#define N 100
__global__ void fun(int *a) {
    a[threadIdx.x] = threadIdx.x * threadIdx.x;
}
int main() {
    int a[N], *da;
    int i;
    cudaMalloc(&da, N * sizeof(int));
    fun<<<1, N>>>(da);
    cudaMemcpy(a, da, N * sizeof(int),
               cudaMemcpyDeviceToHost);
    for (i = 0; i < N; ++i)
        printf("%d\n", a[i]);
    return 0;
}
```

Observation

No cudaDeviceSynchronize required.

GPU Hello World with a Global.

```
#include <stdio.h>
#include <cuda.h>
const char *msg = "Hello World.\n";
__global__ void dkernel() {
    printf(msg);
}
int main() {
    dkernel<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

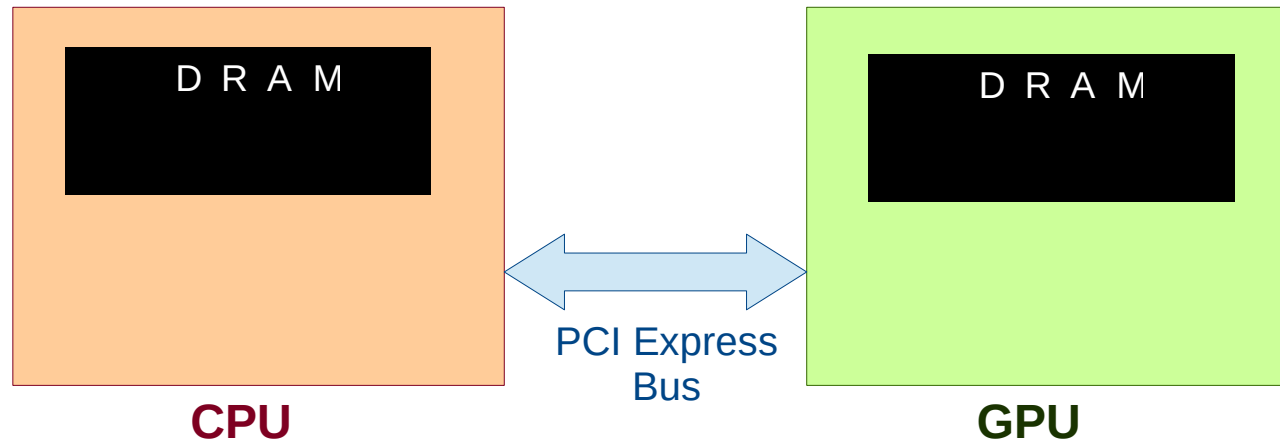
Takeaway

CPU and GPU memories are separate (for discrete GPUs).

#define msg "Hello World.\n" is okay.

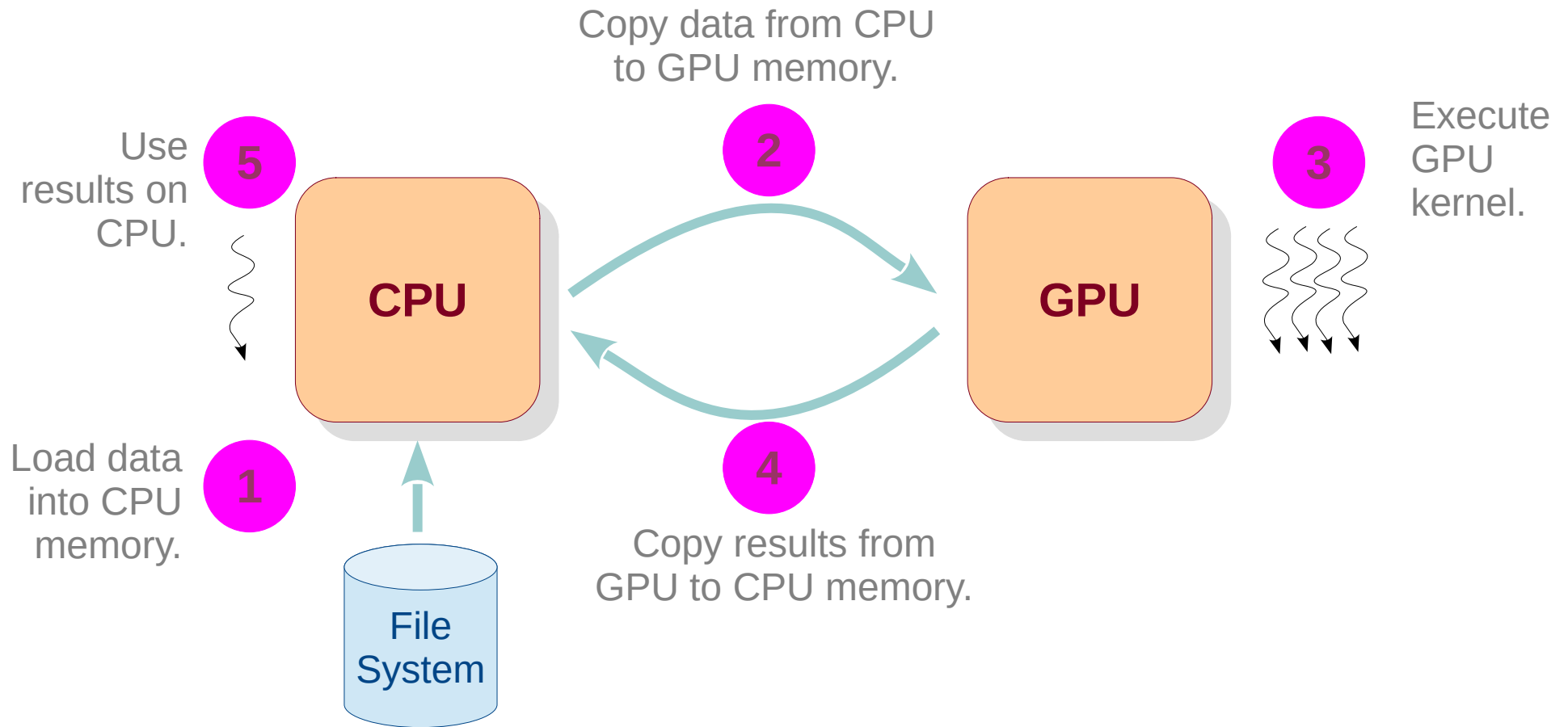
Compile: nvcc hello.cu
error: identifier "msg" is undefined in device code

Separate Memories



- CPU and its associated (discrete) GPUs have separate physical memory (RAM).
- A variable in CPU memory cannot be accessed directly in a GPU kernel.
- A programmer needs to maintain copies of variables.
- It is programmer's responsibility to keep them in sync.

Typical CUDA Program Flow



Typical CUDA Program Flow

1 Load data into CPU memory.

- fread / rand

2 Copy data from CPU to GPU memory.

- cudaMemcpy(..., cudaMemcpyHostToDevice)

3 Call GPU kernel.

- mykernel<<<x, y>>>(...)

4 Copy results from GPU to CPU memory.

- cudaMemcpy(..., cudaMemcpyDeviceToHost)

5 Use results on CPU.

Typical CUDA Program Flow

2 Copy data from CPU to GPU memory.

- `cudaMemcpy(..., cudaMemcpyHostToDevice)`

This means we need two copies of the same variable – one on CPU another on GPU.

e.g., `int *cpuarr, *gpuarr;`

Matrix `cpumat, gpumat;`

Graph `cpug, gpug;`

CPU-GPU Communication

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(char *arr, int arrlen) {
    unsigned id = threadIdx.x;
    if (id < arrlen) {
        ++arr[id];
    }
}
```

```
int main() {
    char cpuarr[] = "Gdkkn\x1fVnqkc-",
        *gpuarr;

    cudaMalloc(&gpuarr, sizeof(char) * (1 + strlen(cpuarr)));
    cudaMemcpy(gpuarr, cpuarr, sizeof(char) * (1 + strlen(cpuarr)), cudaMemcpyHostToDevice);
    dkernel<<<1, 32>>>(gpuarr, strlen(cpuarr));
    cudaDeviceSynchronize(); // unnecessary, but okay.
    cudaMemcpy(cpuarr, gpuarr, sizeof(char) * (1 + strlen(cpuarr)), cudaMemcpyDeviceToHost);
    printf(cpuarr);

    return 0;
}
```

Classwork

1. Write a CUDA program to initialize an array of size 32 to all zeros in parallel.
2. Change the array size to 1024.
3. Create another kernel that adds i to $array[i]$.
4. Change the array size to 8000.
5. Check if answer to problem 3 still works.

Homework ($z = x^2 + y^3$)

- Read a sequence of integers from a file.
- Square each number.
- Read another sequence of integers from another file.
- Cube each number.
- Sum the two sequences element-wise, store in the third sequence.
- Print the computed sequence.

Thread Organization

- A kernel is launched as a grid of threads.
- A grid is a 3D array of thread-blocks (`gridDim.x`, `gridDim.y` and `gridDim.z`).
 - Thus, each block has `blockIdx.x`, `.y`, `.z`.
- A thread-block is a 3D array of threads (`blockDim.x`, `.y`, `.z`).
 - Thus, each thread has `threadIdx.x`, `.y`, `.z`.

Grids, Blocks, Threads

Each thread uses IDs to decide what data to work on.

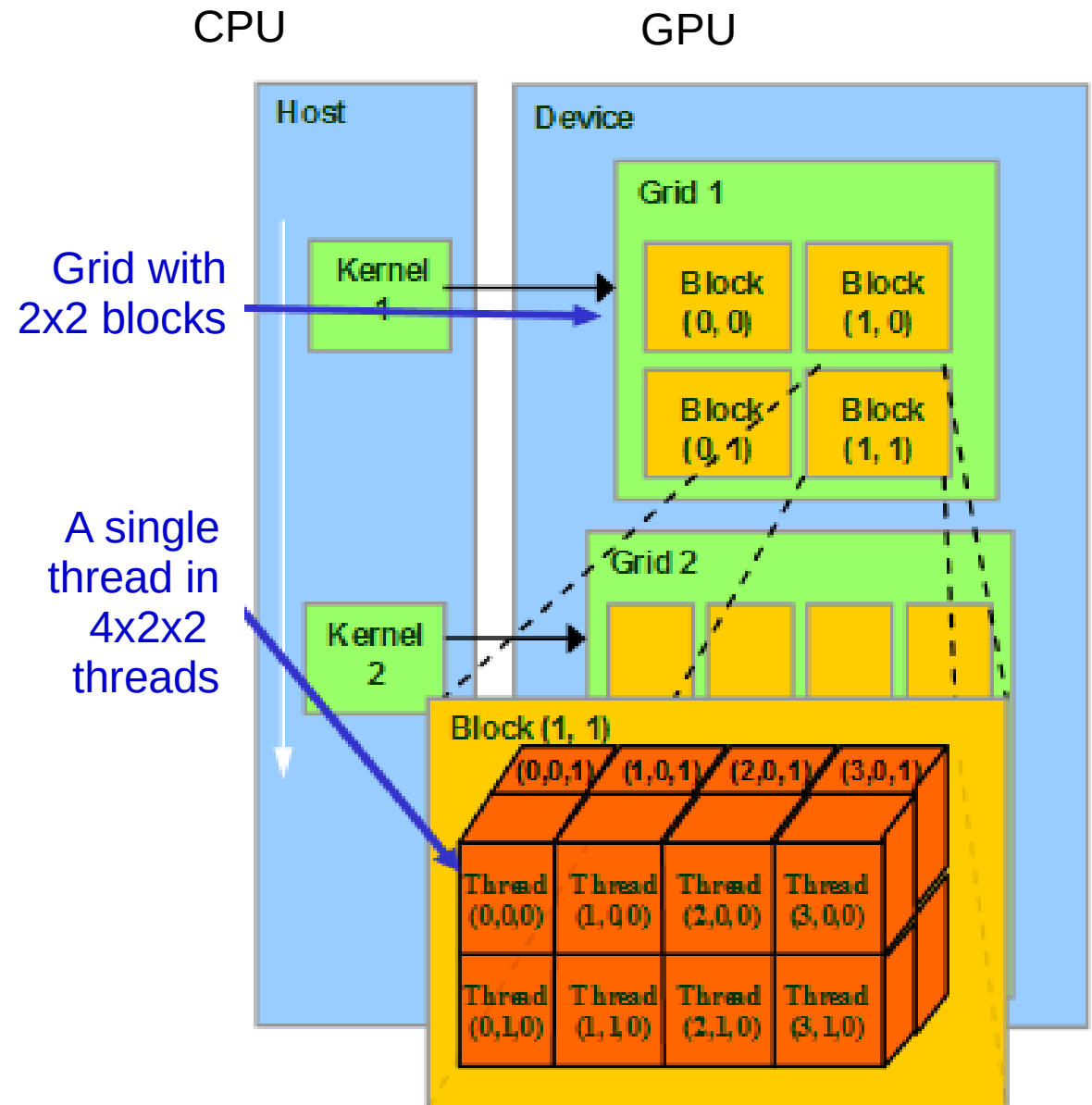
- Block ID: 1D, 2D, or 3D
- Thread ID: 1D, 2D, or 3D

Simplifies memory addressing when processing multi-dimensional data

- Image processing
- Solving PDEs on volumes
- ...

Typical configuration:

- 1-5 blocks per SM
- 128-1024 threads per block.
- Total 2K-100K threads.
- You can launch a kernel with millions of threads.



Accessing Dimensions

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel() {
    if (threadIdx.x == 0 && blockIdx.x == 0 &&
        threadIdx.y == 0 && blockIdx.y == 0 &&
        threadIdx.z == 0 && blockIdx.z == 0) {
        printf("%d %d %d %d %d %d.\n", gridDim.x, gridDim.y, gridDim.z,
            blockDim.x, blockDim.y, blockDim.z);
    }
}
int main() {
    dim3 grid(2, 3, 4);
    dim3 block(5, 6, 7);
    dkernel<<<grid, block>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

How many times the kernel printf gets executed when the *if* condition is changed to *if (threadIdx.x == 0)* ?

Number of threads launched = $2 * 3 * 4 * 5 * 6 * 7$.
Number of threads in a thread-block = $5 * 6 * 7$.
Number of thread-blocks in the grid = $2 * 3 * 4$.

ThreadId in x dimension is in [0..5).
BlockId in y dimension is in [0..3).

2D

Write the kernel to initialize the matrix to unique ids.

What is the output of this program?

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(unsigned *matrix) {
    unsigned id = threadIdx.x * blockDim.y + threadIdx.y;
    matrix[id] = id;
}
#define N    5
#define M    6

int main() {
    dim3 block(N, M, 1);
    unsigned *matrix, *hmatrix;

    cudaMalloc(&matrix, N * M * sizeof(unsigned));
    hmatrix = (unsigned *)malloc(N * M * sizeof(unsigned));

    dkernel<<<1, block>>>(matrix);
    cudaMemcpy(hmatrix, matrix, N * M * sizeof(unsigned), cudaMemcpyDeviceToHost);

    for (unsigned ii = 0; ii < N; ++ii) {
        for (unsigned jj = 0; jj < M; ++jj) {
            printf("%2d ", hmatrix[ii * M + jj]);
        }
        printf("\n");
    }
    return 0;
}
```

```
$ a.out
 0  1  2  3  4  5
 6  7  8  9 10 11
12 13 14 15 16 17
18 19 20 21 22 23
24 25 26 27 28 29
```

1D

Write the kernel to initialize the matrix to unique ids.

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(unsigned *matrix) {
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    matrix[id] = id;
}
#define N    5
#define M    6
int main() {
    unsigned *matrix, *hmatrix;

    cudaMalloc(&matrix, N * M * sizeof(unsigned));
    hmatrix = (unsigned *)malloc(N * M * sizeof(unsigned));

    dkernel<<<N, M>>>(matrix);
    cudaMemcpy(hmatrix, matrix, N * M * sizeof(unsigned), cudaMemcpyDeviceToHost);

    for (unsigned ii = 0; ii < N; ++ii) {
        for (unsigned jj = 0; jj < M; ++jj) {
            printf("%2d ", hmatrix[ii * M + jj]);
        }
        printf("\n");
    }
    return 0;
}
```

Takeaway

One can perform computation on multi-dimensional data using a one-dimensional block.

If I want the launch configuration to be `<<<2, X>>>`, what is X?
The rest of the code should be intact.

Launch Configuration for Huge Data

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(unsigned *vector) {
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    vector[id] = id;
}
#define BLOCKSIZE 1024
int main(int nn, char *str[]) {
    unsigned N = atoi(str[1]);
    unsigned *vector, *hvector;
    cudaMalloc(&vector, N * sizeof(unsigned));
    hvector = (unsigned *)malloc(N * sizeof(unsigned));

    unsigned nblocks = ceil(N / BLOCKSIZE);
    printf("nblocks = %d\n", nblocks);

    dkernel<<<nblocks, BLOCKSIZE>>>(vector);
    cudaMemcpy(hvector, vector, N * sizeof(unsigned), cudaMemcpyDeviceToHost);
    for (unsigned ii = 0; ii < N; ++ii) {
        printf("%4d ", hvector[ii]);
    }
    return 0;
}
```

Access out-of-bounds

Find two issues with this code.

Needs floating-point division

Launch Configuration for Large Size

```
#include <stdio.h>
#include <cuda.h>
__global__ void dkernel(unsigned *vector, unsigned vectorsize) {
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id < vectorsize) vector[id] = id;
}
#define BLOCKSIZE    1024
int main(int nn, char *str[]) {
    unsigned N = atoi(str[1]);
    unsigned *vector, *hvector;
    cudaMalloc(&vector, N * sizeof(unsigned));
    hvector = (unsigned *)malloc(N * sizeof(unsigned));

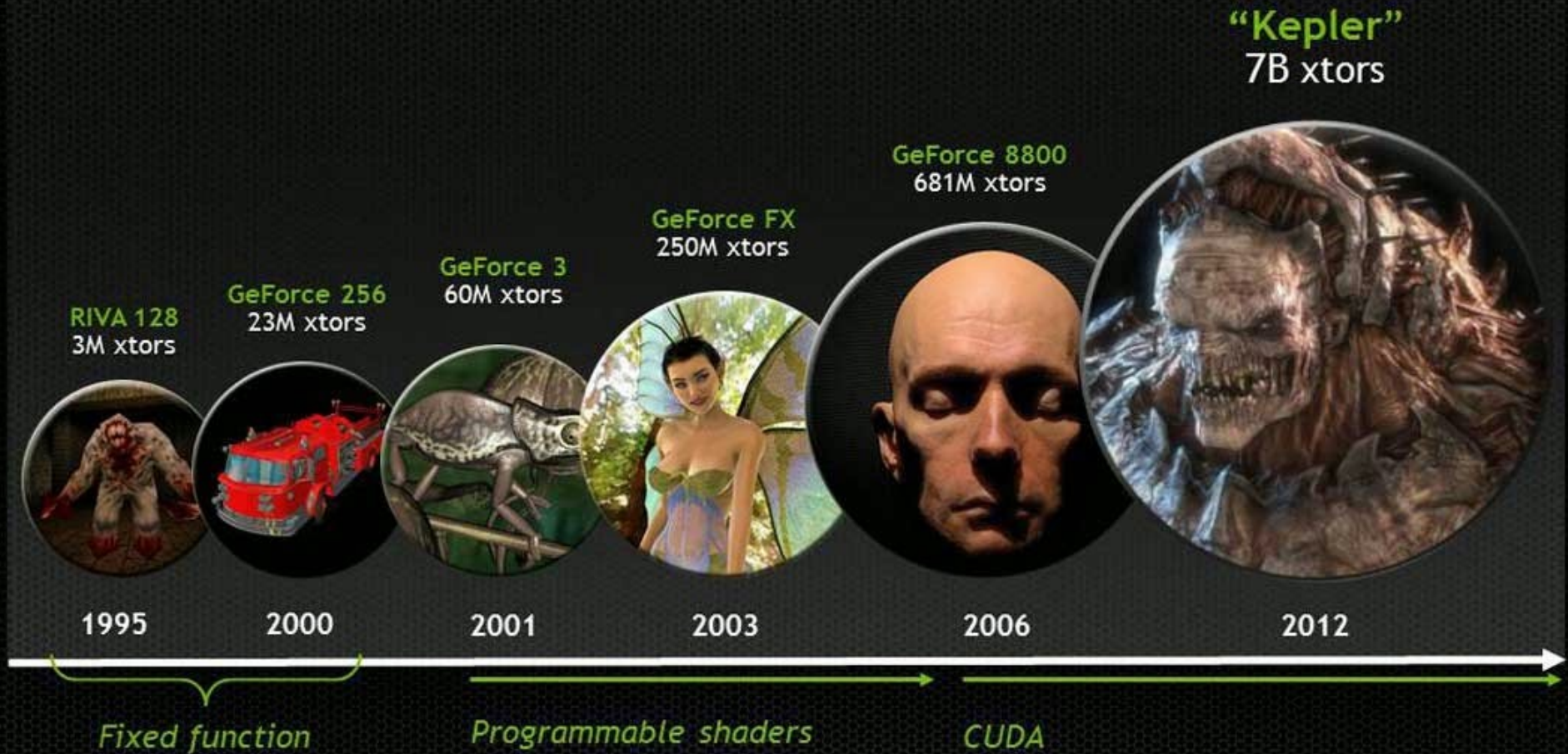
    unsigned nblocks = ceil((float)N / BLOCKSIZE);
    printf("nblocks = %d\n", nblocks);

    dkernel<<<nblocks, BLOCKSIZE>>>(vector, N);
    cudaMemcpy(hvector, vector, N * sizeof(unsigned), cudaMemcpyDeviceToHost);
    for (unsigned ii = 0; ii < N; ++ii) {
        printf("%4d ", hvector[ii]);
    }
    return 0;
}
```

Classwork

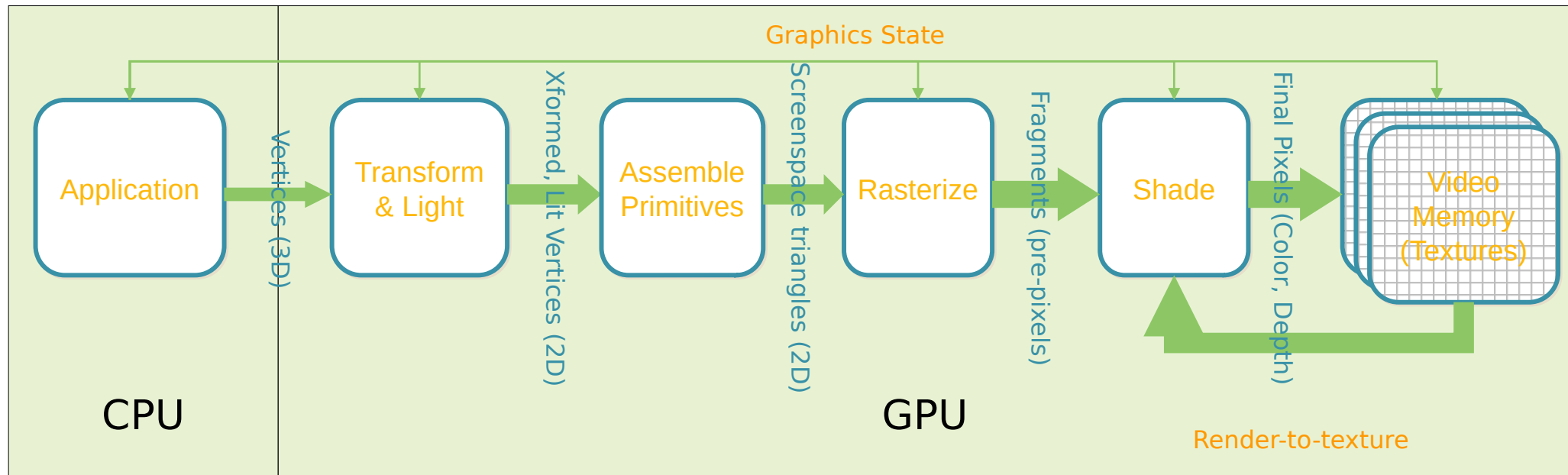
- Read several points as (x, y) coordinates from input.
- For each pair of points, compute euclidean distance $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ in parallel.
- Print the maximum distance.

Evolution of GPUs



Earlier GPGPU Programming

GPGPU = General Purpose Graphics Processing Units.



- Applications: Protein Folding, Stock Options Pricing, SQL Queries, MRI Reconstruction.
- Required intimate knowledge of graphics API and GPU architecture.
- Program complexity: Problems expressed in terms of vertex coordinates, textures and shaders programs.
- Random memory reads/writes not supported.
- Lack of double precision support.

GPU Vendors

- NVIDIA
- AMD
- Intel
- Qualcomm
- ARM
- Broadcom
- Matrox Graphics
- Vivante
- Samsung
- ...

GPU Languages

- **CUDA** (*compute unified device language*)
 - Proprietary, NVIDIA specific
- **OpenCL** (*open computing language*)
 - Universal, works across all computing devices
- **OpenACC** (*open accelerator*)
 - Universal, works across all accelerators
- There are also interfaces:
 - Python → CUDA
 - Javascript → OpenCL
 - LLVM → PTX

Two Configurations

Feature	P100	V100
# of SMX Units	56	80
# of CUDA Cores	3584	5120
# Tensor Cores	NA	640
Peak FP64 FLOPS	5.3 TF	7.5 TF
Register File Size	~14 MB	~20 MB
Compute Capability	6.0	7.0
Onboard GDDR5 Memory	16 GB	16 / 32 GB

top500.org

- Listing of most powerful machines.
 - Ranked by performance (FLOPS)
- As of November 2021
 - Rank 1: Fugaku from Japan (over 7.6 million cores)
 - Rank 2: Summit from USA (over 2.4 million cores)
 - Rank 3: Sierra from USA (over 1.5 million cores)
 - Rank 4: TaihuLight from China (10.6 million cores)
 - Rank 5: Perlmutter from USA (over 0.7 million cores)

Homework: What is India's rank? Where is this computer? How many cores?

Matrix Squaring

```
void squarecpu(unsigned *matrix, unsigned *result,
               unsigned matrixsize /* = 64 */) {
    for (unsigned ii = 0; ii < matrixsize; ++ii) {
        for (unsigned jj = 0; jj < matrixsize; ++jj) {

            for (unsigned kk = 0; kk < matrixsize; ++kk) {
                result[ii * matrixsize + jj] +=
                    matrix[ii * matrixsize + kk] * matrix[kk * matrixsize + jj];
            }
        }
    }
}
```

CPU time = 1.527 ms

Matrix Squaring (version 1)

```
square<<<1, N>>>(matrix, result, N); // N = 64
```

```
__global__ void square(unsigned *matrix,  
                        unsigned *result,  
                        unsigned matrixsize) {  
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;  
    for (unsigned jj = 0; jj < matrixsize; ++jj) {  
        for (unsigned kk = 0; kk < matrixsize; ++kk) {  
            result[id * matrixsize + jj] +=  
                matrix[id * matrixsize + kk] *  
                matrix[kk * matrixsize + jj];  
        }  
    }  
}
```

CPU time = 1.527 ms, GPU v1 time = 6.391 ms

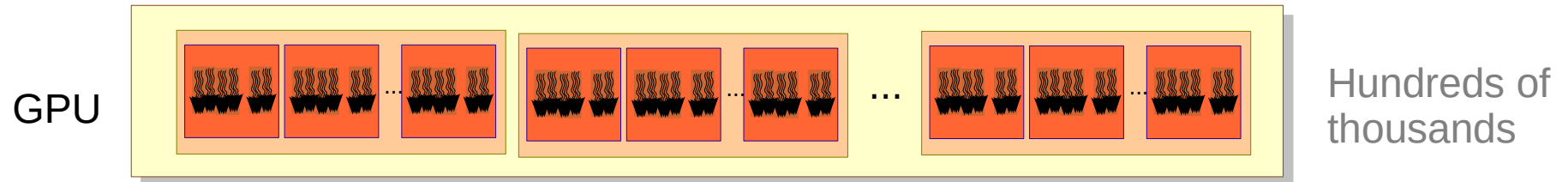
Matrix Squaring (version 2)

```
square<<<N, N>>>(matrix, result, N); // N = 64
```

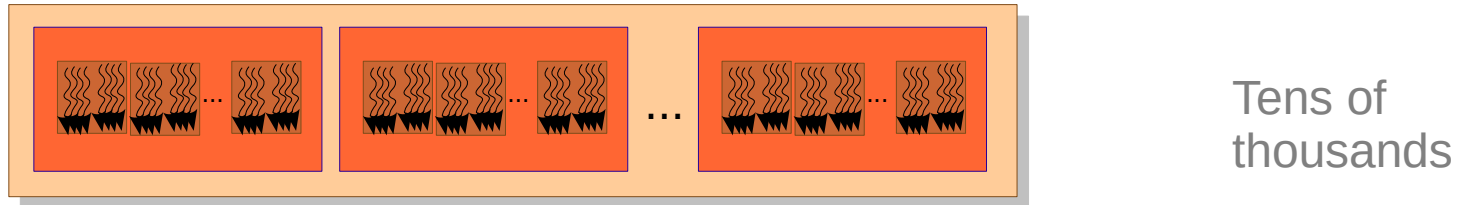
```
__global__ void square(unsigned *matrix,  
                      unsigned *result,  
                      unsigned matrixsize) {  
  
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;  
    unsigned ii = id / matrixsize;  
    unsigned jj = id % matrixsize; } Homework: What if you  
    for (unsigned kk = 0; kk < matrixsize; ++kk) { interchange ii and jj?  
        result[ii * matrixsize + jj] += matrix[ii * matrixsize + kk] *  
                                           matrix[kk * matrixsize + jj];  
    }  
}
```

CPU time = 1.527 ms, GPU v1 time = 6.391 ms,
GPU v2 time = 0.1 ms

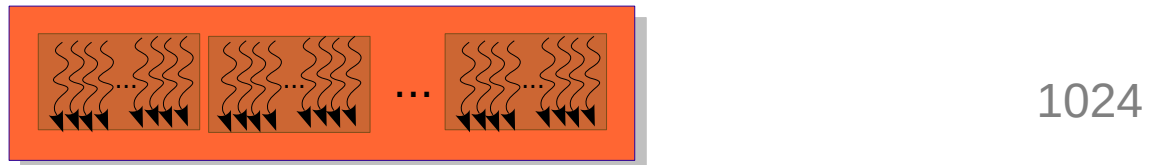
GPU Computation Hierarchy



Multi-processor



Block



Warp



Thread



What is a Warp?



Warp

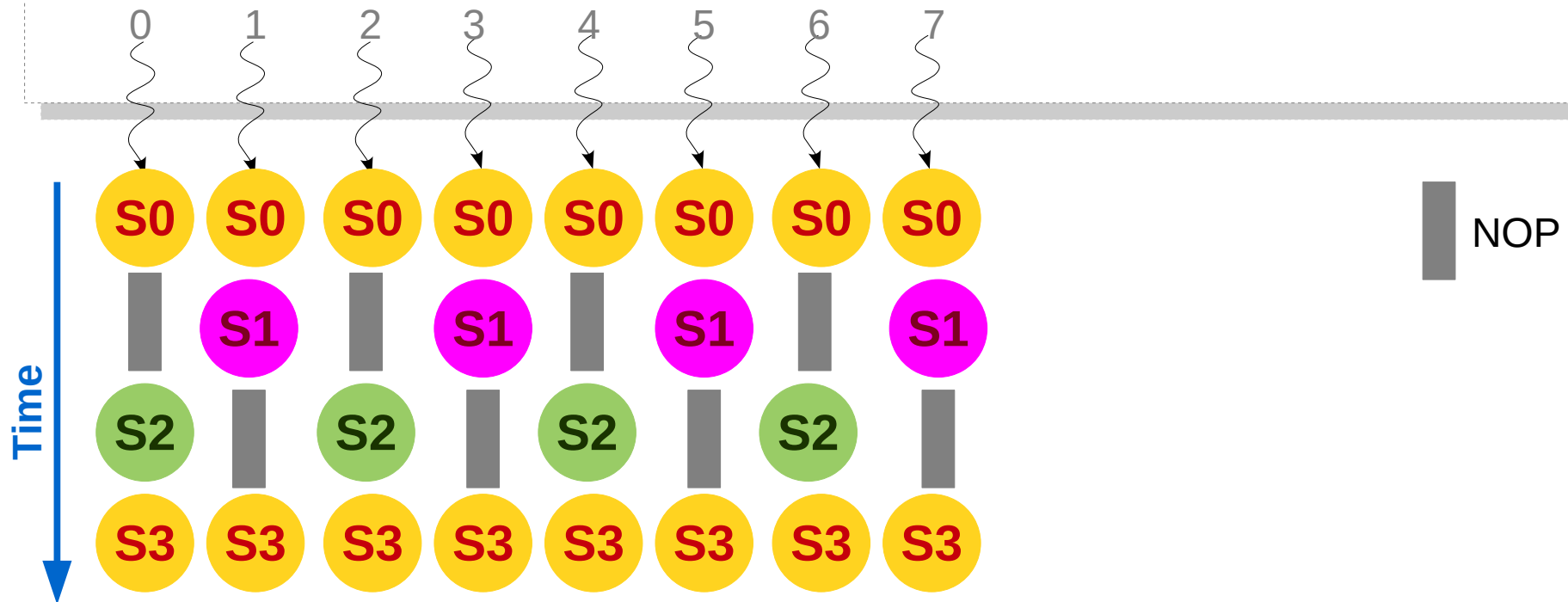
- A set of consecutive threads (currently 32) that execute in **SIMD** fashion.
- SIMD == Single Instruction Multiple Data
- Warp-threads are fully synchronized. There is an implicit barrier after each step / instruction.
- **Memory coalescing** is closely related to warps.

Takeaway

It is a misconception that all threads in a GPU execute in lock-step. Lock-step execution is true for threads only within a warp.

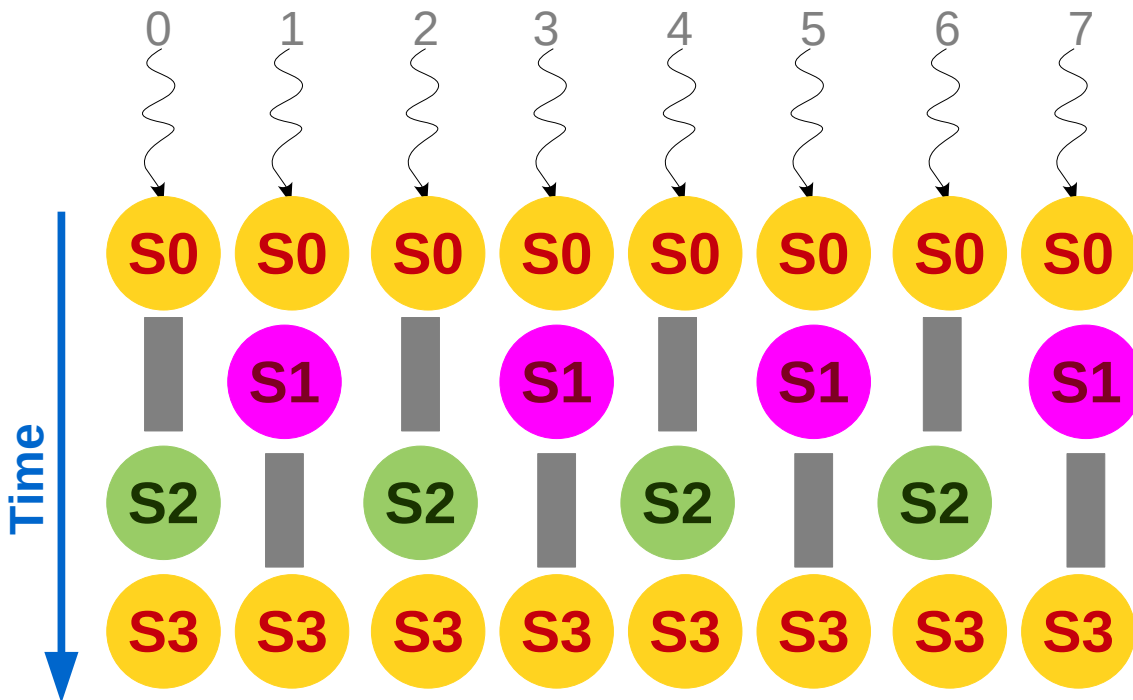
Warp with Conditions

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize)  
{  
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x; S0  
    if (id % 2) vector[id] = id; S1  
    else vector[id] = vectorsize * vectorsize; S2  
    vector[id]++; S3  
}
```



Warp with Conditions

- When different warp-threads execute different instructions, threads are said to diverge.
- Hardware executes threads satisfying same condition together, ensuring that other threads execute a no-op.
- This adds sequentiality to the execution.
- This problem is termed as **thread-divergence**.



Classwork

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize)
{
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    for (unsigned ii = 0; ii < id; ++ii)
        vector[id] += ii;
}
```

Does this code diverge?

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize)
{
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    if (id % 2) vector[id] = id;
    else if (vector[id] % 2) vector[id] = id / 2;
    else vector[id] = id * 2;
}
```

Does this code diverge further?

vector is initialized to {0, 1, 2, 3, ...}.

Thread-Divergence

- Since thread-divergence makes execution sequential, conditions are evil in the kernel codes?

```
if (vectorsize < N) S1; else  
S2;
```

Condition but no divergence

- Then, conditions evaluating to different truth-values are evil?

```
if (id / 32) S1; else S2;
```

Different truth-values but no divergence

Takeaway

Conditions are not bad;
they evaluating to different truth-values is also not bad;
they evaluating to different truth-values for warp-threads is bad.

Classwork

- Rewrite the following program fragment to remove thread-divergence.

```
// assert(x == y || x == z);  
if (x == y) x = z;  
else x = y;
```

Classwork

- Find the maximum in a large array as follows:
 - Let the array have N elements.
 - Launch a kernel with N/K threads.
 - Each thread finds the maximum among K elements.
 - The K elements are written to same or different array.
 - The same kernel is launched with K threads to find the final maximum.
- Find an element in parallel.
 - Return its index.

Homework

- Write kernels to **encrypt** and **decrypt** messages. Assume that the message contains only a..z.
 - *Encrypt*: each character c becomes $c+1$. z becomes a .
 - *Encrypt*: each i^{th} character c becomes $c+i$.
- Parallelize run-length-encoding to **compress** data.
 - e.g., if input is 0001101000100011110111010001 then the output is 032113134131131. The initial bit is same as input, followed by frequencies of that bit and **its negation**.
 - For the same input, another compression output is 427111154213261301. This stores **index** and frequency.