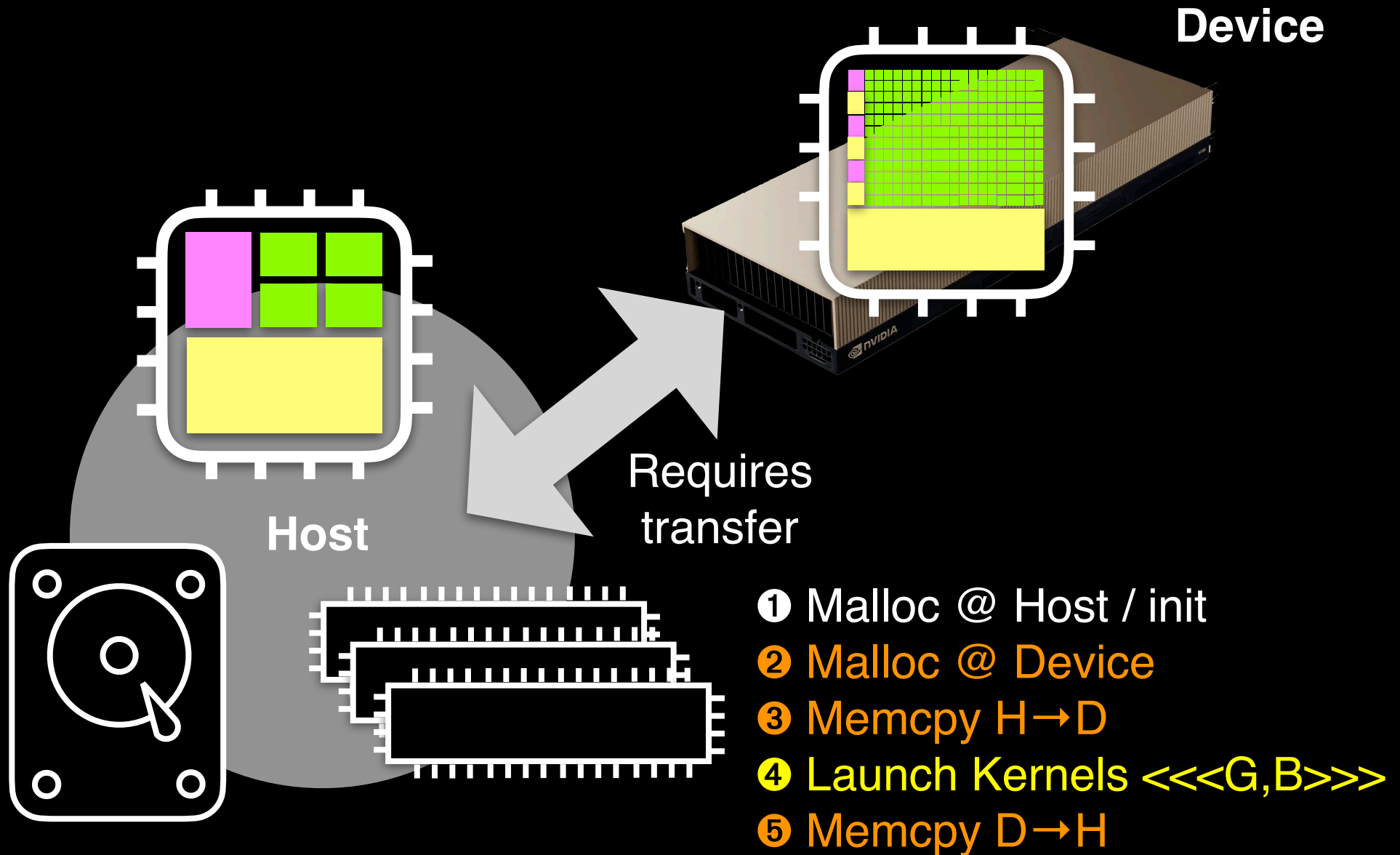


# Recap



# CUDA API examples

## ② Malloc @ Device

```
cudaMalloc((void**) &A_device, n_data * sizeof(float));
```

## ③ Memcpy H→D

```
cudaMemcpy(A_device, A_host, n_data * sizeof(float), cudaMemcpyHostToDevice);
```

## ④ Launch Kernels <<<G,B>>>

```
vec_add<<<grid_size, block_size>>>(A_device, B_device, C_device, n_data, n_ops);
```

## ⑤ Memcpy D→H

```
cudaMemcpy(C_host, C_device, n_data * sizeof(float), cudaMemcpyDeviceToHost);
```

# blockDim blockIdx threadIdx

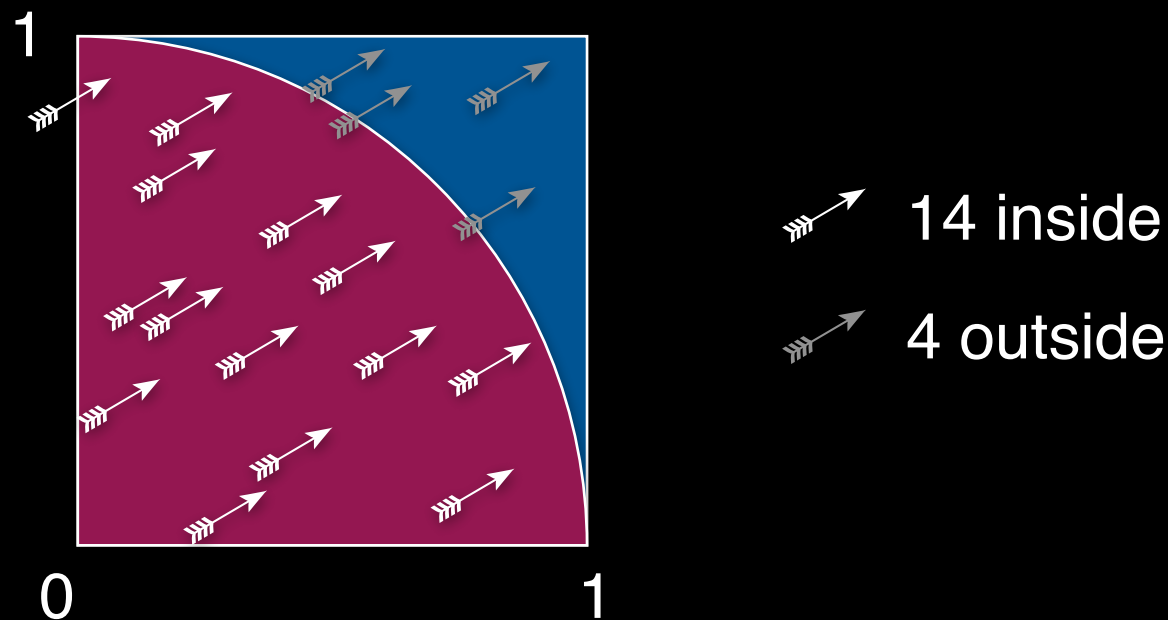
```
__global__ void vec_add(const float* A, const float* B, float* C, unsigned
{
    unsigned long long int i_data = blockDim.x * blockIdx.x + threadIdx.x;
    if (i_data < n_data)
    {
        for (unsigned i = 0; i < n_ops; ++i)
        {
            C[i_data] = A[i_data] + B[i_data];
        }
    }
}
```



# Computing $\pi$

This time we will try to compute  $\pi$

Basic idea of computing  $\pi$  will be via throwing “darts” randomly at a quarter of a unit circle



Since the area of the quarter of a unit circle is  $\pi/4$  we can estimate  $\pi$  as

$$\pi_{\text{est}} = 14 / (14+4) \times 4 = 3.11\dots$$

# Title

tests - JupyterLab

hsf-india-examples/rand.cu at n

https://jupyterhub.ssl-hep.org/user/p.chang@uf

File Edit View Run Kernel Tabs Settings Help

jovyan@jupyter-p-2echang

```
1 #include <iostream>
2 #include <stdio.h>
3
4 int main()
5 {
6     std::cout << "#####" << std::endl;
7     std::cout << "#          #" << std::endl;
8     std::cout << "#    Computing Pi via Darts    #" << std::endl;
9     std::cout << "#          #" << std::endl;
10    std::cout << "#####" << std::endl;
11
12    // we will launch 65536 blocks
13    int grid_size = pow(2, 16);
14
15    // we will generate 512 points each block
16    int block_size = 512;
17
18    // total threads
19    int n_total_threads = grid_size * block_size;
20 }
```

"rand.cu" 20L, 598B written

20,1 All

Simple 1 0 Mem: 167.9... jovyan@jupyter-p-2echang-40ufl-2eedu--research-2dsoftwa-2df-2dindia-2... 0

Check it compiles

```
$ nvcc rand.cu -o rand
```

# Random number generation

```
#include <curand.h>
#include <curand_kernel.h>
```

We will use the CUDA's API tool to perform RNG

We will throw `n_total_threads` worth of "darts" so we setup states for those

```
// pointer to the array of "curandState" on the device
curandState* state_device;

// malloc array of random state
cudaMalloc((void**) &state_device, n_total_threads * sizeof(curandState));
```

Then we set their states using a GPU kernel defined like:

```
__global__ void setup_curandState(curandState* state)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    curand_init(1234, idx, 0, &state[idx]);
}
```

Then we launch the kernel in a grid

```
setup_curandState<<grid_size, block_size>>(state_device);
```



tests - JupyterLab    hsf-india-examples/rand.cu at n X

https://jupyterhub.ssl-hep.org/user/p.chang@uf

File Edit View Run Kernel Tabs Settings Help

jovyan@jupyter-p-2echang X

```

1 #include <iostream>
2 #include <stdio.h>
3 #include <curand.h>
4 #include <curand_kernel.h>
5
6 int main()
7 {
8     std::cout << "#####" << std::endl;
9     std::cout << "#
10    std::cout << "#    Computing Pi via Darts    #" << std::endl;
11    std::cout << "#
12    std::cout << "#####" << std::endl;
13
14    // we will launch 65536 blocks
15    int grid_size = pow(2, 16);
16
17    // we will generate 512 points each block
18    int block_size = 512;
19
20    // total threads
21    int n_total_threads = grid_size * block_size;
22
23    // pointer to the array of "curandState" on the device
24    curandState* state_device;
25
26    // malloc array of random state
27    cudaMalloc((void**) &state_device, n_total_threads * sizeof(curandState));
28 }

```

23,5    All

Simple 1 \$ 0 Mem: 167.9... jovyan@jupyter-p-2echang-40ufl-2eedu--research-2dsoftwa-2df-2dindia-2... 0

# Title

Check it compiles

# Title

The screenshot shows a JupyterLab window with a terminal and a code editor. The terminal output shows the file "rand.cu" has been written with 40 lines and 1207 bytes. The code editor contains the following C++ code:

```
2 #include <stdio.h>
3 #include <curand.h>
4 #include <curand_kernel.h>
5
6 __global__ void setup_curandState(curandState* state)
7 {
8     int idx = blockDim.x * blockIdx.x + threadIdx.x;
9     curand_init(1234, idx, 0, &state[idx]);
10 }
11
12 int main()
13 {
14     std::cout << "#####" << std::endl;
15     std::cout << "#                #" << std::endl;
16     std::cout << "#    Computing Pi via Darts    #" << std::endl;
17     std::cout << "#                #" << std::endl;
18     std::cout << "#####" << std::endl;
19 }
```

Below the code, the terminal shows: "rand.cu" 40L, 1207B written. At the bottom right of the terminal area, the text "10,1" and "4%" is visible.

The JupyterLab interface includes a menu bar (File, Edit, View, Run, Kernel, Tabs, Settings, Help) and a sidebar with icons for file explorer, home, and settings. The browser address bar shows the URL: https://jupyterhub.ssl-hep.org/user/p.chang@uf.

tests - JupyterLab    hsf-india-examples/rand.cu at n X

https://jupyterhub.ssl-hep.org/user/p.chang@uf

File Edit View Run Kernel Tabs Settings Help

jovyan@jupyter-p-2echang X

```

7 {
8     std::cout << "#####" << std::endl;
9     std::cout << "#           #" << std::endl;
10    std::cout << "#   Computing Pi via Darts   #" << std::endl;
11    std::cout << "#           #" << std::endl;
12    std::cout << "#####" << std::endl;
13
14    // we will launch 65536 blocks
15    int grid_size = pow(2, 16);
16
17    // we will generate 512 points each block
18    int block_size = 512;
19
20    // total threads
21    int n_total_threads = grid_size * block_size;
22
23    // pointer to the array of "curandState" on the device
24    curandState* state_device;
25
26    // malloc array of random state
27    cudaMalloc((void**) &state_device, n_total_threads * sizeof(curandState));
28
29    // actually setup each random state with different index
30    setup_curandState<<<grid_size, block_size>>>(state_device);
31
32    // wait until all threads are done
33    cudaDeviceSynchronize();
34 }

```

'rand.cu' 34L, 1051B written

31,0-1    Bot

Simple 1 \$ 0 Mem: 167.9... jovyan@jupyter-p-2echang-40ufl-2eedu--research-2dsoftwa-2df-2dindia-2... 0

# Title

Check it compiles

# Now we setup a counter

A counter in the device will count whether each dart thrown fell inside the quarter circle or not

```

30  curandState* state_device;
31
32  // malloc array of random state
33  cudaMalloc((void**) &state_device, n_total_threads * sizeof(curandState));
34
35  // actually setup each random state with different index
36  setup_curandState<<<grid_size, block_size>>>(state_device);
37
38  // wait until all threads are done
39  cudaDeviceSynchronize();
40
41  // setup a counter
42  int* n_inside_device;
43
44  // allocate memory
45  cudaMalloc((void**) &n_inside_device, sizeof(int));
46
47 }

```

"rand.cu" 47L, 1338B written

45,52 Bot

Simple 1 \$ 0 Mem: 167.9... jovyan@jupyter-p-2echang-40ufl-2eedu--research-2dsoftwa-2df-2dindia-2... 0

# “Inside? or outside?” kernel

We define a “dart throwing” function like the following

```

__global__ void throw_dart(curandState* state, int* n_inside)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    double x = curand_uniform(&state[idx]);
    double y = curand_uniform(&state[idx]);
    double d = sqrt(x * x + y * y);
    if (d <= 1)
    {
        *n_inside += 1;
    }
}

```

parse thread idx  
of this thread

get two  
random  
numbers

get  
distance

Is this OK??

Now we throw darts like the following

```

throw_dart<<grid_size, block_size>>(state_device, n_inside_device);

```

# “Inside? or outside?” kernel

We define a “dart throwing” function like the following

```

__global__ void throw_dart(curandState* state, int* n_inside)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    double x = curand_uniform(&state[idx]);
    double y = curand_uniform(&state[idx]);
    double d = sqrt(x * x + y * y);
    if (d <= 1)
    {
        atomicAdd(n_inside, 1);
    }
}

```

Now we throw darts like the following

```

throw_dart<<grid_size, block_size>>(state_device, n_inside_device);

```



# atomicAdd

Each thread will try to count up the same memory  
This can create a race condition

Race condition is when the result can depend on which thread finishes first (or when)

To avoid this we need to “block” the counting so that no two process can access the same memory

`atomicAdd` provides such feature

```
atomicAdd(n_inside, 1);
```

multiple threads will try to increase `n_inside` but now it will be properly counted

# Other atomic operations

## ☐ 7.14. Atomic Functions

### ☐ 7.14.1. Arithmetic Functions

7.14.1.1. atomicAdd()

7.14.1.2. atomicSub()

7.14.1.3. atomicExch()

7.14.1.4. atomicMin()

7.14.1.5. atomicMax()

7.14.1.6. atomicInc()

7.14.1.7. atomicDec()

7.14.1.8. atomicCAS()

# Title

The screenshot shows a JupyterLab browser window with a terminal and a code editor. The terminal shows the user 'jovyan' at 'jupyter-p-2echang'. The code editor displays C++ code for setting up a curand state and throwing darts. A red box highlights the 'throw\_dart' function. The status bar at the bottom shows 'Simple' mode, a shell icon, '1' line, '0' column, 'Mem: 166.7...', and the user 'jovyan' at 'jupyter-p-2echang-40ufl-2eedu--research-2dsoftwa-2df-2dindia-2...'. The cursor is at line 13, column 1, and the scroll position is 6%.

```
5 |
6 | __global__ void setup_curandState(curandState* state)
7 | {
8 |     int idx = blockDim.x * blockIdx.x + threadIdx.x;
9 |     curand_init(1234, idx, 0, &state[idx]);
10 | }
11 |
12 | __global__ void throw_dart(curandState* state, int* n_inside)
13 | {
14 |     int idx = blockDim.x * blockIdx.x + threadIdx.x;
15 |     double x = curand_uniform(&state[idx]);
16 |     double y = curand_uniform(&state[idx]);
17 |     double d = sqrt(x * x + y * y);
18 |     if (d <= 1)
19 |     {
20 |         atomicAdd(n_inside, 1);
21 |     }
22 | }
```

13,1 6%

Simple 1 \$ 0 Mem: 166.7... jovyan@jupyter-p-2echang-40ufl-2eedu--research-2dsoftwa-2df-2dindia-2... 0

tests - JupyterLab | hsf-india-examples/rand.cu at n X

https://jupyterhub.ssl-hep.org/user/p.chang@uf

File Edit View Run Kernel Tabs Settings Help

jovyan@jupyter-p-2echang X

```
49 | setup_curandState<<<grid_size, block_size>>>(state_device);
50 |
51 | // wait until all threads are done
52 | cudaDeviceSynchronize();
53 |
54 | // setup a counter
55 | int* n_inside_device;
56 |
57 | // allocate memory
58 | cudaMalloc((void**) &n_inside_device, sizeof(int));
59 |
60 | throw_dart<<<grid_size, block_size>>>(state_device, n_inside_device);
61 |
62 | // wait until all threads are done
63 | cudaDeviceSynchronize();
64 |
65 |
66 | }
```

"rand.cu" 66L, 1780B written

62,5 Bot

Simple 1 \$ 0 Mem: 167.81... jovyan@jupyter-p-2echang-40ufl-2eedu--research-2dsoftwa-2df-2dindia-2... 0

# Retrieving the result

Make a memory on host and copy back

```
// create a counter on host to copy device number to
int* n_inside_host = new int;

// copy the result to host
cudaMemcpy(n_inside_host, n_inside_device, sizeof(int), cudaMemcpyDeviceToHost);
```

Then use the value to compute pi

```
// estimate pi by counting fraction
double pi_estimate = (double) *n_inside_host / n_total_threads * 4.;

// print pi_estimate
std::cout << " --- Result ---" << std::endl;
std::cout << " pi_estimate: " << pi_estimate << std::endl;
```

```
64 |   cudaDeviceSynchronize();
65 |
66 |   // create a counter on host to copy device number to
67 |   int* n_inside_host = new int;
68 |
69 |   // copy the result to host
70 |   cudaMemcpy(n_inside_host, n_inside_device, sizeof(int), cudaMemcpyDeviceToHost);
71 |
72 |   // estimate pi by counting fraction
73 |   double pi_estimate = (double) *n_inside_host / n_total_threads * 4.;
74 |
75 |   // print pi_estimate
76 |   std::cout << " --- Result ---" << std::endl;
77 |   std::cout << " pi_estimate: " << pi_estimate << std::endl;
78 |
79 |   return 0;
80 |
81 | }
```

81,1 Bot

Simple 1 \$ 0 Mem: 166.7... jovyan@jupyter-p-2echang-40ufl-2eedu--research-2dsoftwa-2df-2dindia-2... 0

# Title

More refined version is here:

[https://raw.githubusercontent.com/sgnoohc/hsf-india-examples/main/  
rand.cu](https://raw.githubusercontent.com/sgnoohc/hsf-india-examples/main/rand.cu)

# Result

```
(myenv) jovyan@jupyter-p--research-2dsoftwa-2---dindia-2dmay2024-2d30ogek5h:~/hsf-india-examples$ ./rand
#####
#                                     #
#   Computing Pi via Darts           #
#                                     #
#####
--- Input data ---
grid_size      = 65536
block_size     = 512
total darts thrown = 33554432

--- Result ---
pi_estimate: 3.14147
```



# Title

# Matrix Summation

This time we will try adding a matrix to another matrix

For simplicity we will declare one matrix of  $2048 \times 2048$  size  
with element of all set to 1

# dim3

This time we will use a different object called “dim3” dim3 is basically a three tuple (x, y, z) that can hold three integer

example:

```
dim3 block_size_ex1(16, 16, 16);  
dim3 block_size_ex2(16, 16, 1);
```

We can use this to launch 3d grid / 3d blocks

# dim3

In our case we want to launch a 1 grid of 16 x 16 block  
So we define them like the following

```
// we will perform each element as one thread
int block_len = 16;

// then the block dimensions are defined
dim3 block_size(block_len, block_len, 1);

// compute number of blocks in each dimension
int grid_len = int(m_dim - 0.5) / block_len + 1;

// then for grid size needs to be computed to cover the entire elements
dim3 grid_size(grid_len, grid_len, 1);
```

And we can use it like the following

```
kernel<<<grid_size, block_size>>>(...)
```

# cudaMallocHost

Previously we have done something like this

```
float* A_host = new float[n_data];
float* B_host = new float[n_data];
float* C_host = new float[n_data];
```

But one could have instead done this

```
float* A_host;
float* B_host;
float* C_host;
cudaMallocHost((void**) &A_host, n_data * sizeof(float))
cudaMallocHost((void**) &B_host, n_data * sizeof(float))
cudaMallocHost((void**) &C_host, n_data * sizeof(float))
```

*Why.....??*

# Copying data WHILE processing

One of the biggest power of GPU is that it can process data while copying stuff in the background!

This can help eliminate or reduce overhead!

For example consider the normal case

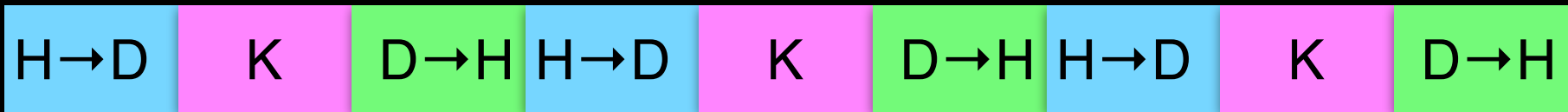
```
cudaMemcpy(..., cudaMemcpyHostToDevice);
kernel<<<...>>>(...);
cudaMemcpy(..., cudaMemcpyDeviceToHost);
```

This will process



# If you repeatedly process this

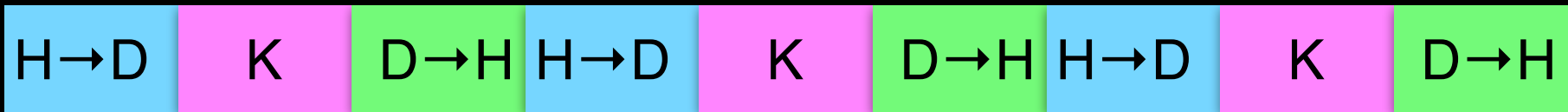
Things will all happen in sequence



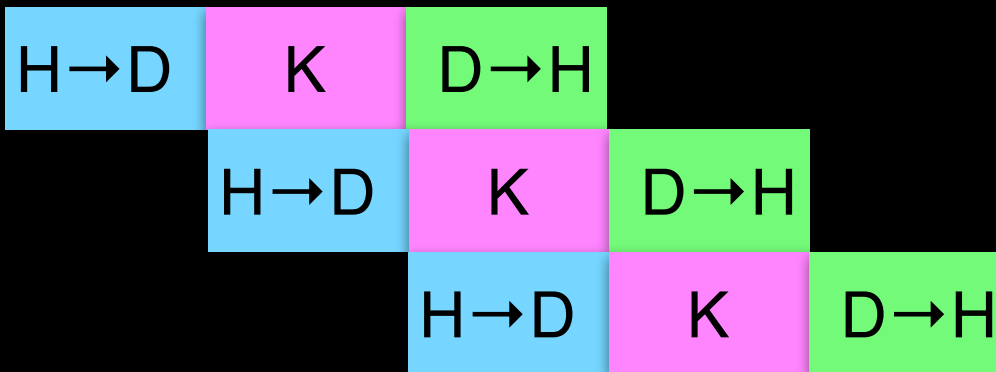
What if you could stagger?

# If you repeatedly process this

Things will all happen in sequence



What if you could stagger?



You would win!

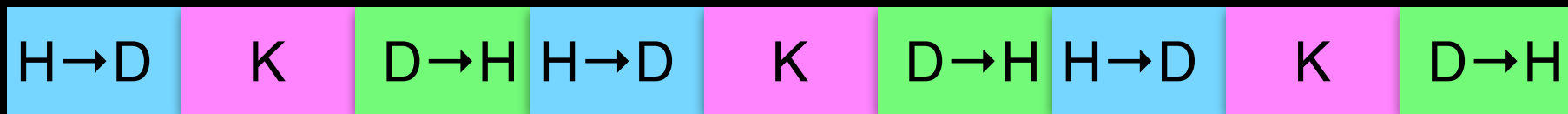


# cudaStream

in order to stagger and schedule the cuda API or kernel calls, once has to define “lanes” or “streams”

Previously when nothing was specified they were all running on the so-called “default lane”

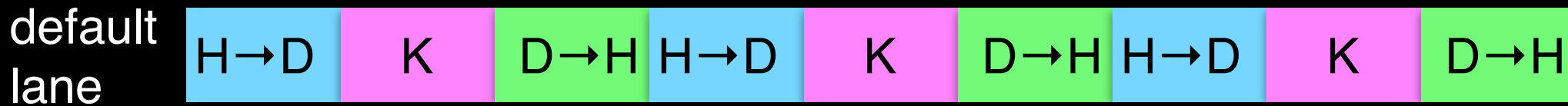
default  
lane



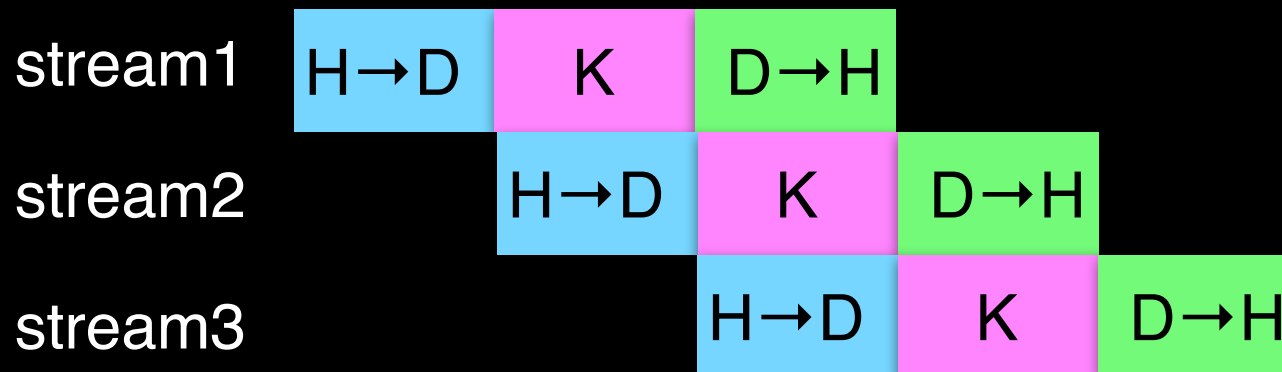
# cudaStream

in order to stagger and schedule the cuda API or kernel calls, once has to define “lanes” or “streams”

Previously when nothing was specified they were all running on the so-called “default lane”



Instead one can define different streams and schedule them



# Creating cudaStream

```
// create cuda streams
cudaStream_t stream[n_repeat];
for (int i = 0; i < n_repeat; ++i)
{
    cudaStreamCreate(&stream[i]);
}
```

Simply create cudaStream\_t objects

# How do I schedule different cudaAPI/kernel to different streams?

For memory copy, we use

```
cudaMemcpyAsync
```

Assuming we have stream[0], stream[1], ... created, we would do

```
cudaMemcpyAsync(a_device,  
                a_host,  
                ntot*sizeof(float),  
                cudaMemcpyHostToDevice,  
                stream[1])
```

# For Kernel calls

For kernel calls we add it to the fourth arguments

```
kernel<<<grid_size, block_size, 0, stream[1]>>>
```

(The third argument is not discussed today, it has to do with shared memory, but I have not particularly found good use of it, so I set it to 0 the default value)

Finish coding up

Refined example here:

[https://raw.githubusercontent.com/sgnoohc/hsf-india-examples/main/  
madd.cu](https://raw.githubusercontent.com/sgnoohc/hsf-india-examples/main/madd.cu)

# Title

```
jovyan@jupyter-p-2echang-40ufl-2eedu--research-2dsoftwa-2df-2dindia-2d:~$ ./madd
#####
#                               #
#      Matrix Sum                #
#      (Overlap Transfer)        #
#                               #
#####
--- Sequential Run ---
Time total (ms): 17.308865

--- Overlapping Run ---
Time total (ms): 9.332256
```



# Profiler

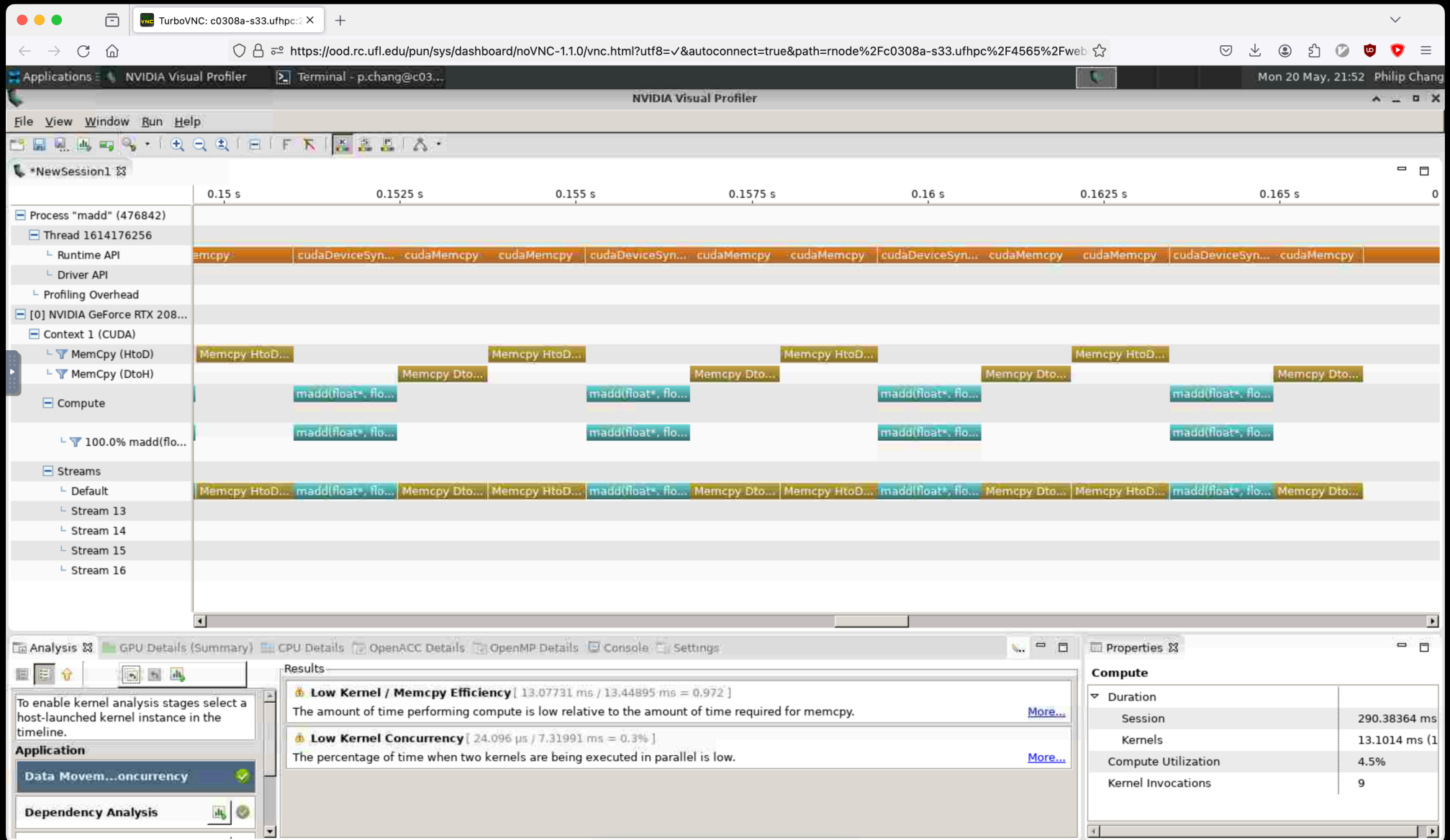
There are several profilers in Nvidia toolkit

Today I will use Nvidia Visual Profiler (nvvp) to show how the staggering of the data copy call vs. kernel calls look like

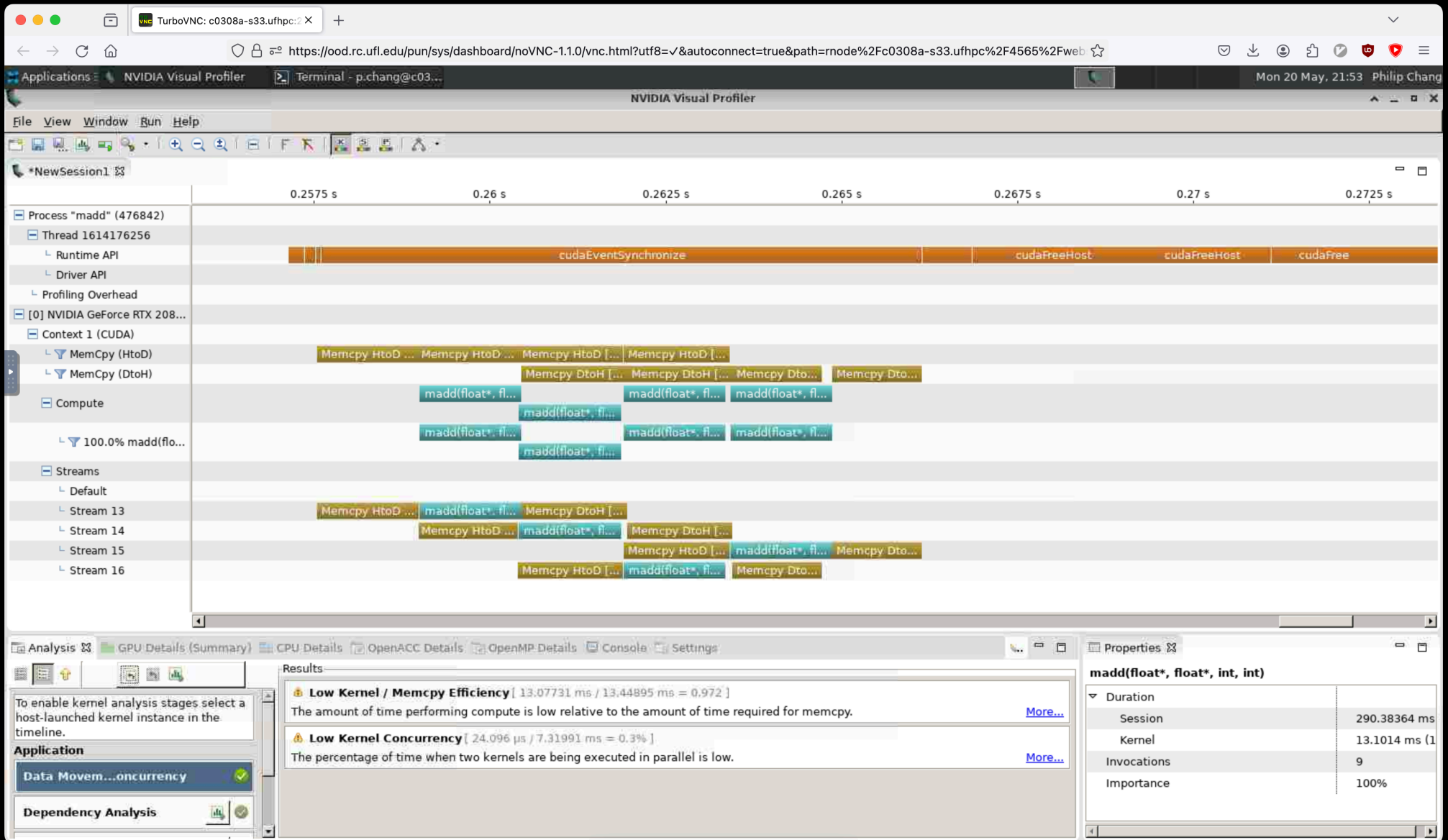
# Once the program is compiled

```
nvvp ./madd
```

# Non-staggered example



# Staggered example



# Title

# Take away messages

Future includes many core approach  $\Rightarrow$  We must be prepared  
(GPU: good growth in processing / energy)

GPU cannot be the solution for all  
(carefully need to approach heterogeneous future computing)

We discussed basic examples in CUDA  
(CUDA is one example there are more)

There are many optimizations “tricks”  
(e.g. optimizing data transfer)

# Some tools

## Parsing command line for large number

```
#include <cstdlib>

int main(int argc, char** argv)
{
    unsigned long long int N_data = strtoull(argv[1], nullptr, 10);
}
```

## Printing out information and putting requirements on input arguments

```
#include <iostream>

if (argc < 2)
{
    std::cout << "Usage:" << std::endl;
    std::cout << std::endl;
    std::cout << "    " << argv[0] << " N_data" << std::endl;
    std::cout << std::endl;
    std::cout << std::endl;
    return 1;
}
```