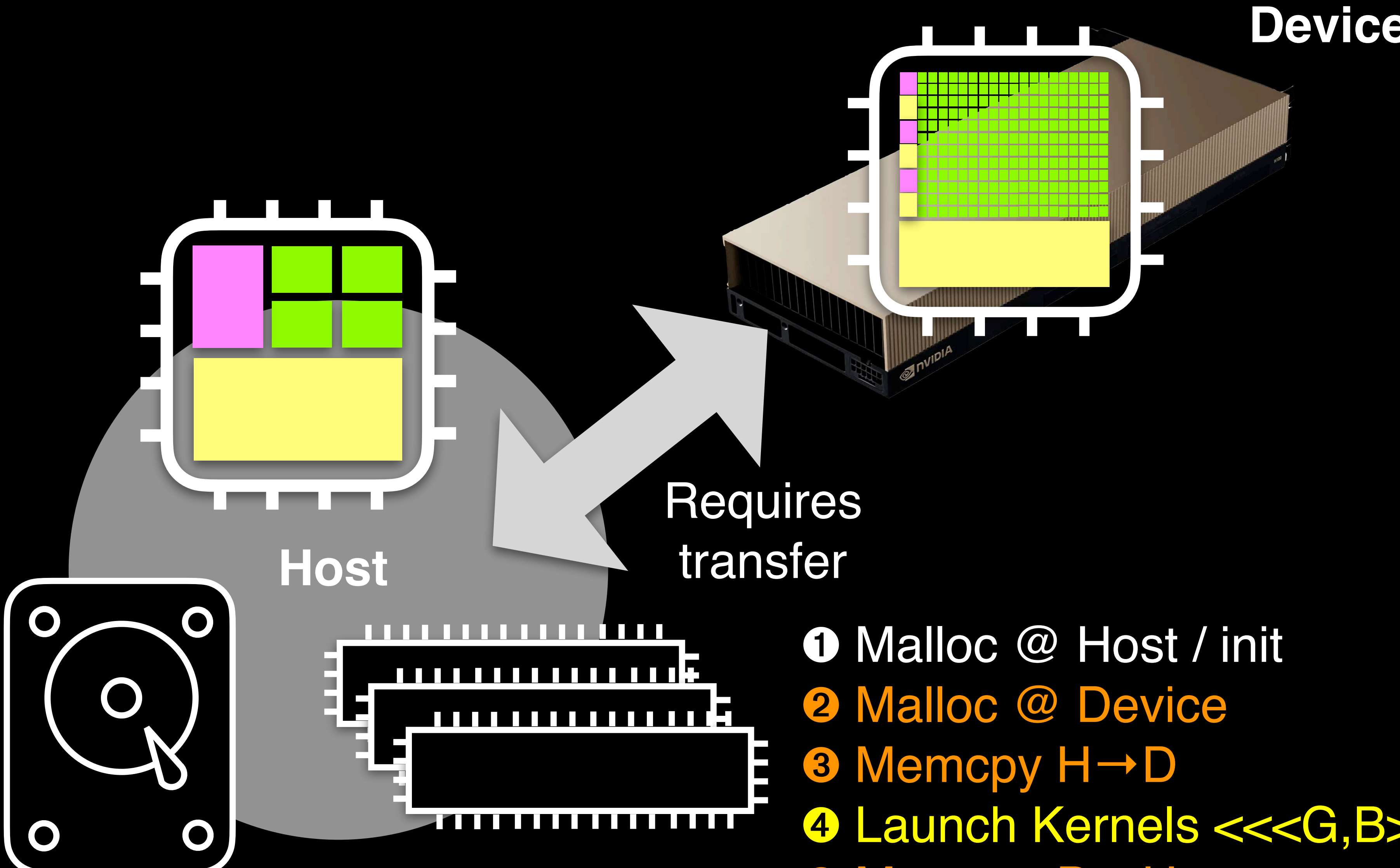
Recap

UF Chang Florida



❶ Malloc @ Host / init
❷ Malloc @ Device
❸ Memcpy H→D
❹ Launch Kernels <<<G,B>>>
❺ Memcpy D→H

CUDA API examples

❷ Malloc @ Device

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

❸ Memcpy H→D

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

❹ Launch Kernels <<<G,B>>>

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

❺ Memcpy D→H

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

blockDim blockIdx threadIdx

UF Chang Florida

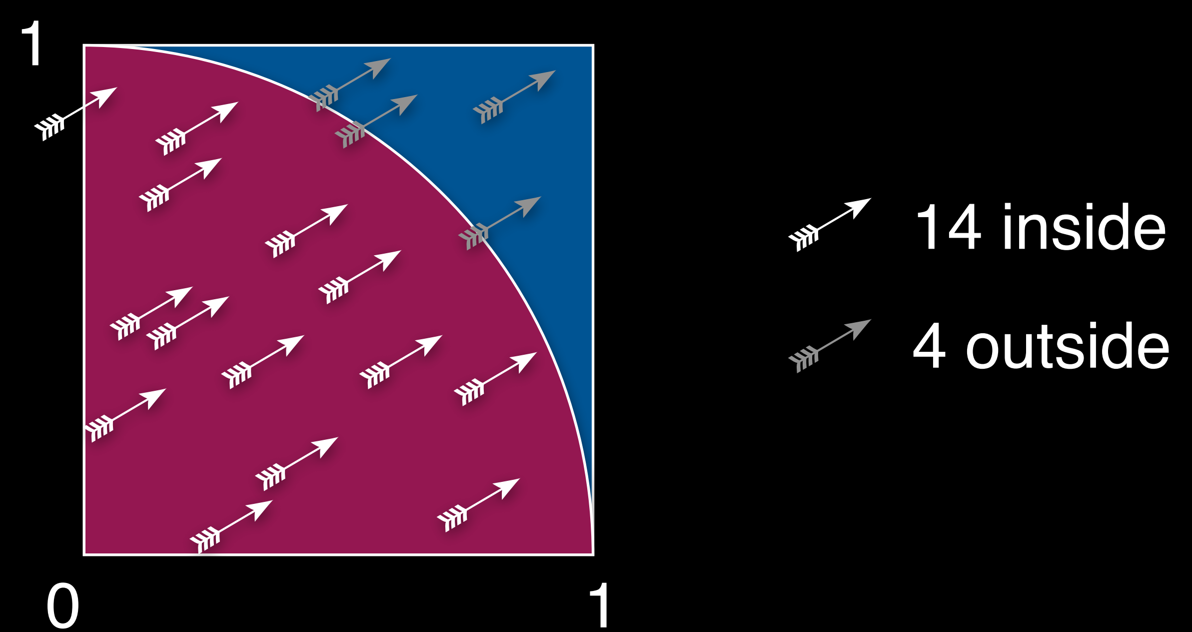
```
__global__ void vec_add(const float* A, const float* B, float* C, unsigned
{

    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 π

UF Chang Florida

This time we will try to compute π

Basic idea of computing π will be via throwing "darts" randomly at a quarter of a unit circle



Since the area of the quarter of a unit circle is π/4 we can estimate π as

π_{est} = 14 / (14+4) × 4 = 3.11…..

Title

```cpp
#include <iostream>
#include <stdio.h>

int main()
{
    std::cout << "################################" << std::endl;
    std::cout << "#                              #" << std::endl;
    std::cout << "#    Computing Pi via Darts    #" << std::endl;
    std::cout << "#                              #" << std::endl;
    std::cout << "################################" << std::endl;

    // we will launch 65536 blocks
    int grid_size = pow(2, 16);

    // we will generate 512 points each block
    int block_size = 512;

    // total threads
    int n_total_threads = grid_size * block_size;
}
~
~
"rand.cu" 20L, 598B written
                                                        20,1          All
```

UF
Chang
Florida

Check it compiles

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

Random number generation

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

UF Chang Florida

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);
```

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

int main()
{
    std::cout << "################################" << std::endl;
    std::cout << "#                              #" << std::endl;
    std::cout << "#    Computing Pi via Darts    #" << std::endl;
    std::cout << "#                              #" << std::endl;
    std::cout << "################################" << std::endl;

    // we will launch 65536 blocks
    int grid_size = pow(2, 16);

    // we will generate 512 points each block
    int block_size = 512;

    // total threads
    int n_total_threads = grid_size * block_size;

    // 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));
}
```

Title

Check it compiles

Title

UF Chang Florida

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

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

int main()
{
    std::cout << "################################" << std::endl;
    std::cout << "#                              #" << std::endl;
    std::cout << "#    Computing Pi via Darts    #" << std::endl;
    std::cout << "#                              #" << std::endl;
    std::cout << "################################" << std::endl;

"rand.cu" 40L, 1207B written
                                                  10,1          4%
```

UF Chang Florida

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

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

File Edit View Run Kernel Tabs Settings Help

jovyan@jupyter-p-2echang

```
{
    std::cout << "################################" << std::endl;
    std::cout << "#                              #" << std::endl;
    std::cout << "#    Computing Pi via Darts    #" << std::endl;
    std::cout << "#                              #" << std::endl;
    std::cout << "################################" << std::endl;

    // we will launch 65536 blocks
    int grid_size = pow(2, 16);

    // we will generate 512 points each block
    int block_size = 512;

    // total threads
    int n_total_threads = grid_size * block_size;

    // 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));

    // actually setup each random state with different index
    setup_curandState<<<grid_size, block_size>>>(state_device);

    // wait until all threads are done
    cudaDeviceSynchronize();
}
"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

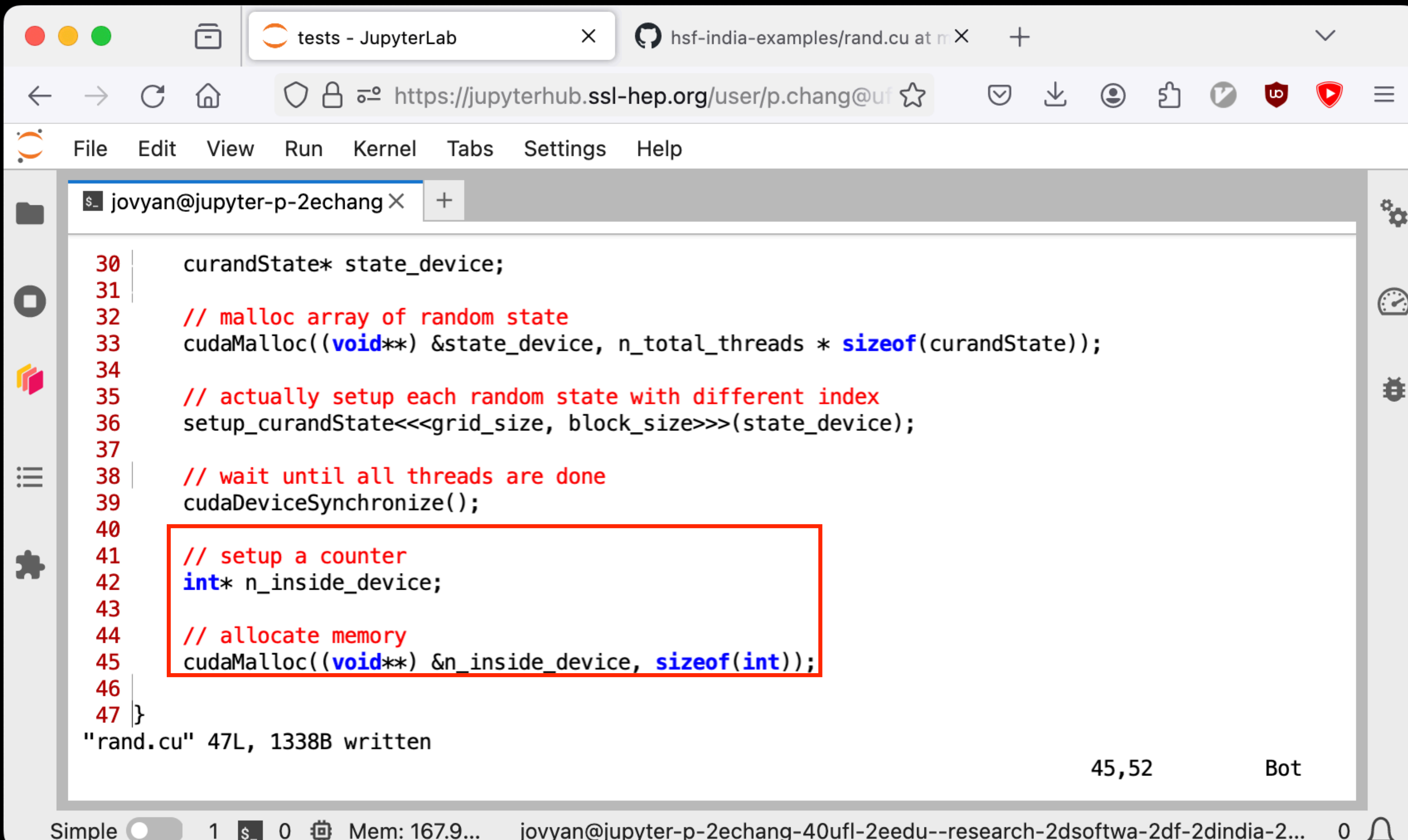
Title

Check it compiles

Now we setup a counter

UF Chang Florida

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



“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

UF Chang Florida

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);
    }
}
```

parse thread idx of this thread

get two random numbers

get distance

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

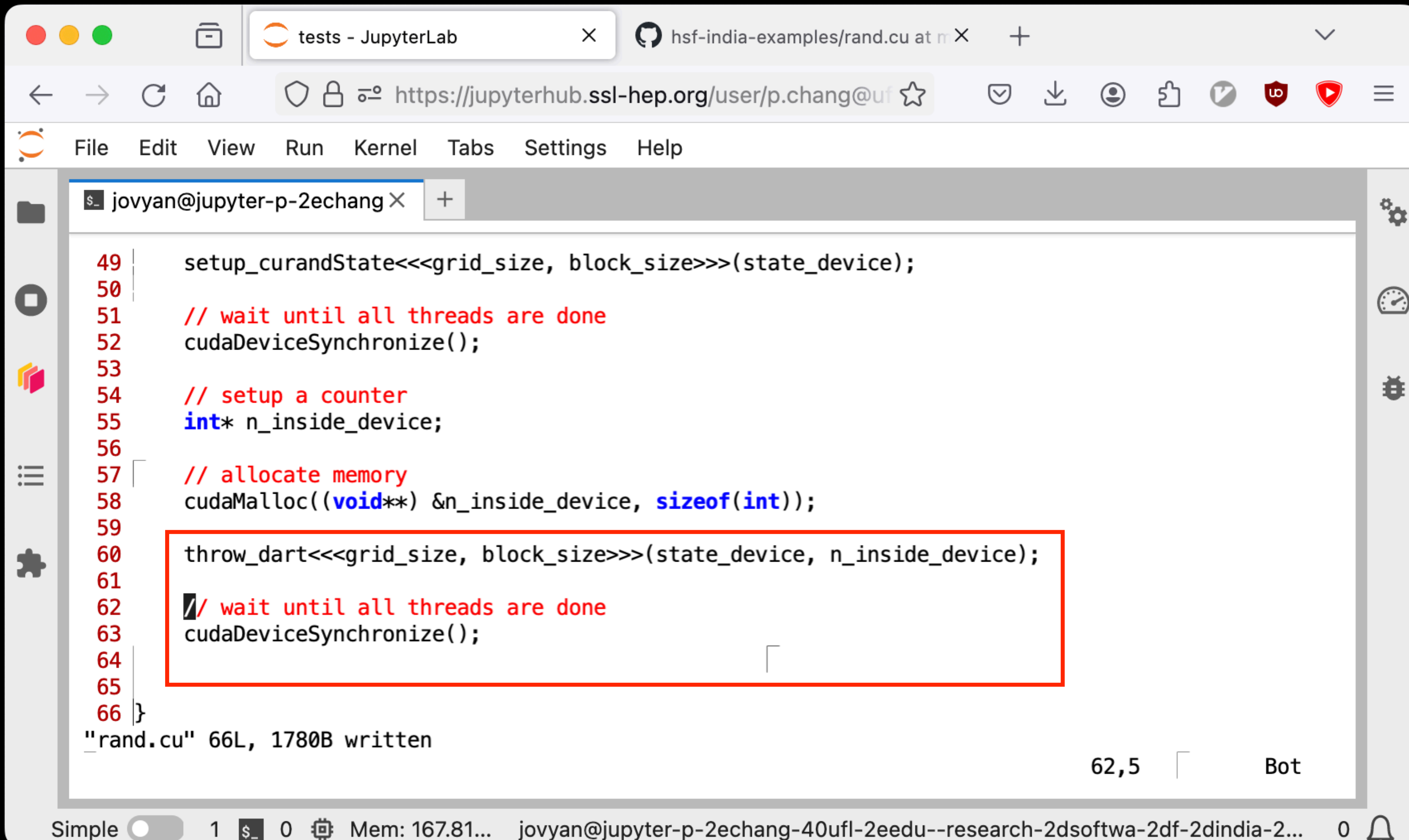
Title

UF Chang Florida

```

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

__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);
    }
}
```

Title

UF
Chang
Florida



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;
```

Title

UF Chang Florida

```
    cudaDeviceSynchronize();

    // 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);

    // 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;

    return 0;

}
```

Title

More refined version is here:
https://raw.githubusercontent.com/sgnoohc/hsf-india-examples/main/rand.cu

Result

UF
Chang
Florida

```
(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

Chang
Florida

Matrix Summation

This time we will try adding a matrix to another matrix

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

dim3

UF
Chang
Florida

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

UF
Chang
Florida

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

UF Chang Florida

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

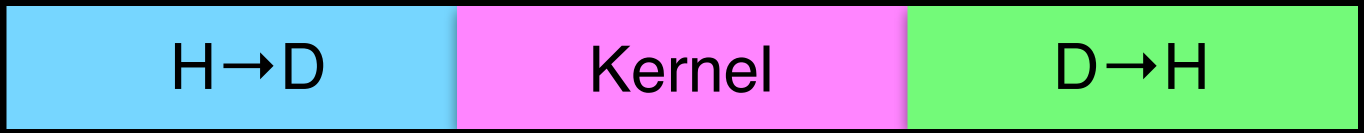
One of the biggest power of GPU is that it can process data while copying stuff in the background!

This can help eliminate or reduce overhead!

For example consider the normal case

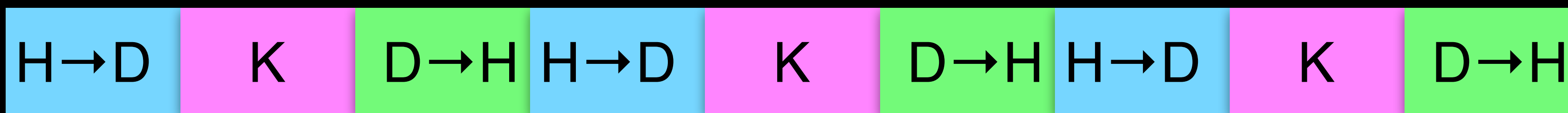
```
cudaMemcpy(…, cudaMemcpyHostToDevice);
kernel<<<…,…>>>(…);
cudaMemcpy(…, cudaMemcpyDeviceToHost);
```

This will process



If you repeatedly process this

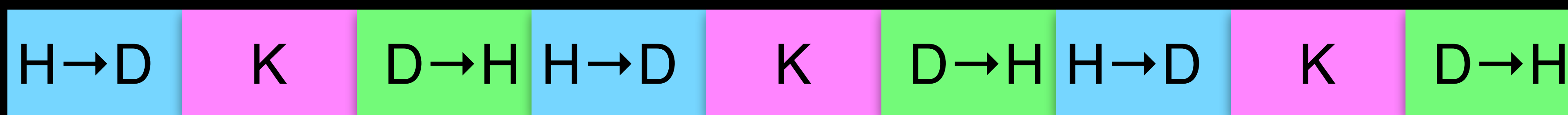
Things will all happen in sequence



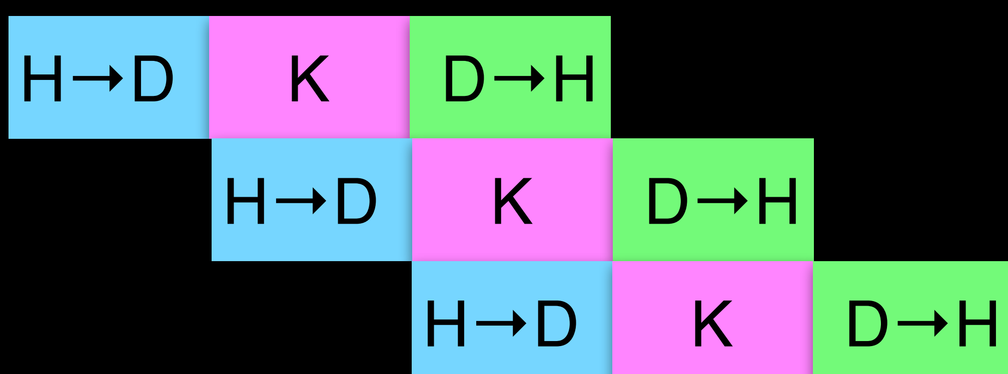
What if you could stagger?

If you repeatedly process this

Things will all happen in sequence



What if you could stagger?



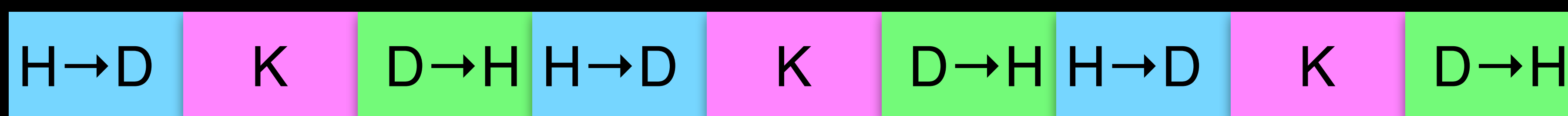
You would win!

cudaStream

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

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

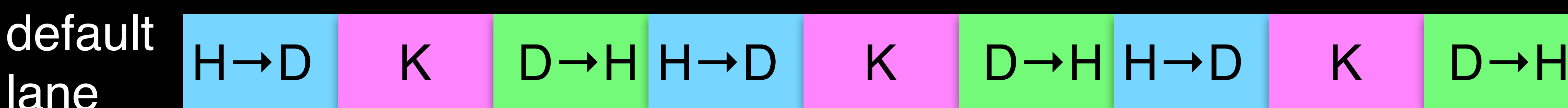
default lane



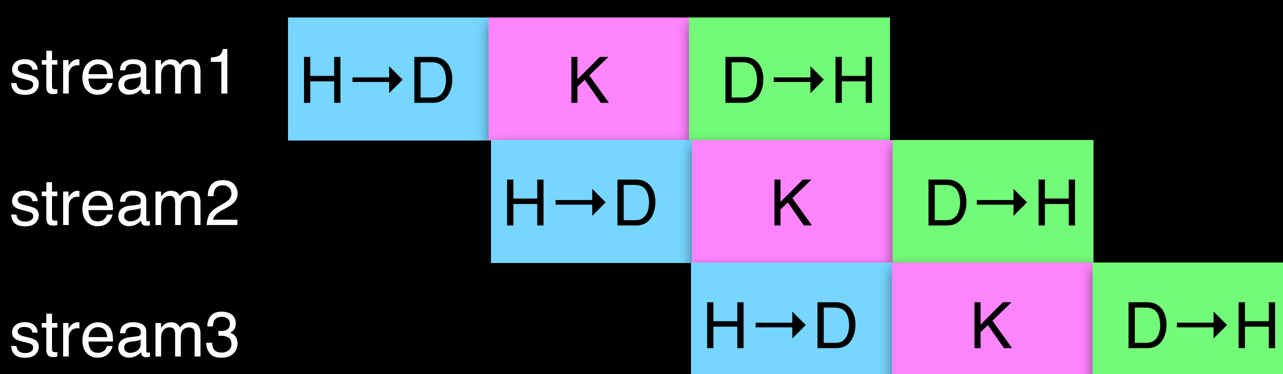
cudaStream

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

Previously when nothing was specificed 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

UF
Chang
Florida

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

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

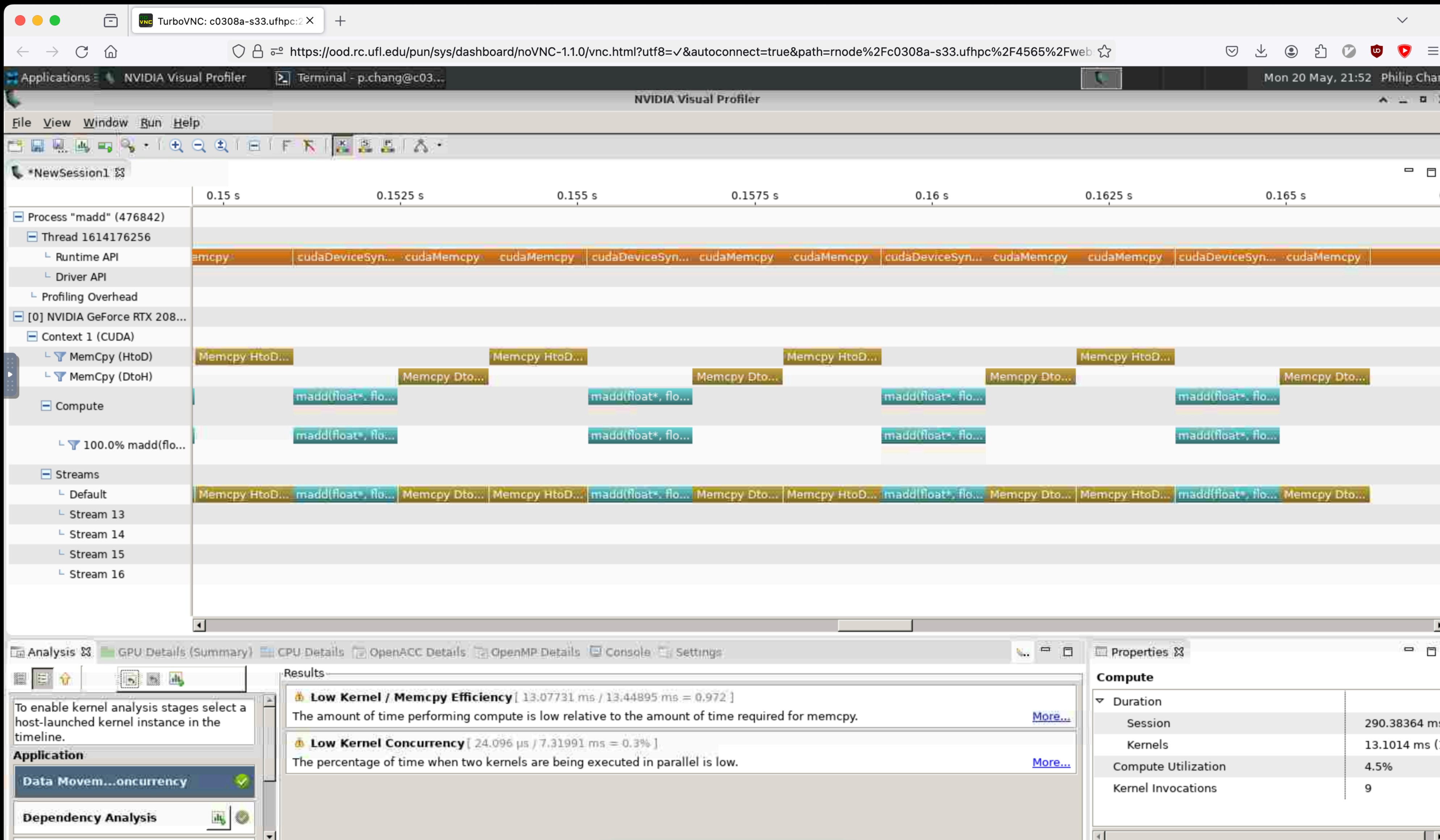
Once the program is compiled

UF
Chang
Florida

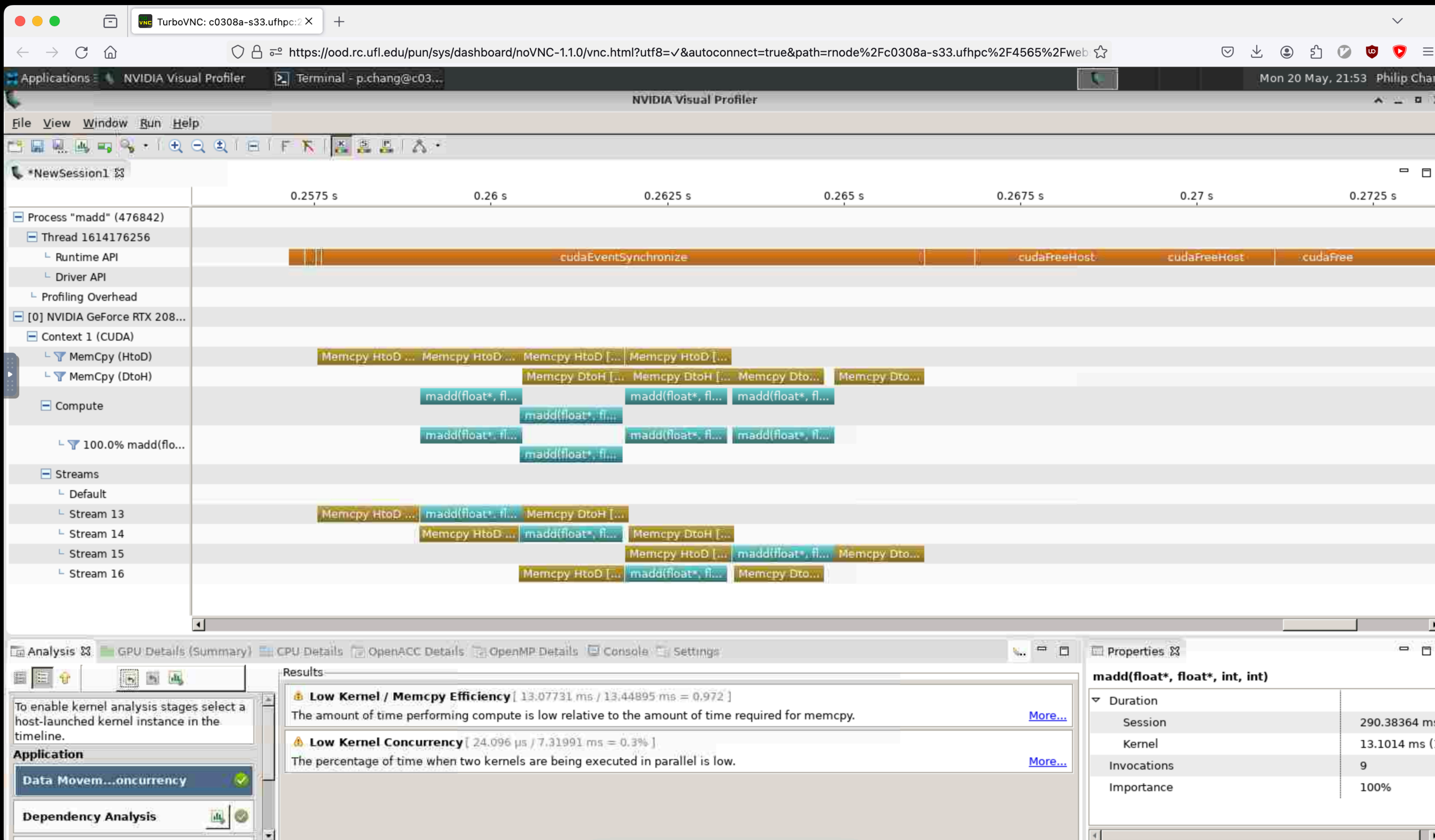
```
nvvp ./madd
```

Non-staggered example

UF Chang Florida



Staggered example



Title

Take away messages

Future includes many core approach ⇒ 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

UF
Chang
Florida

Parsing command line for large number

```cpp
#include <cstdlib>


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

Printing out information and putting requirements on input arguments

```cpp
#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;
}
```