CUDA Introduction

National Taiwan University by Haochen Chiu 2024-11-26

Agenda

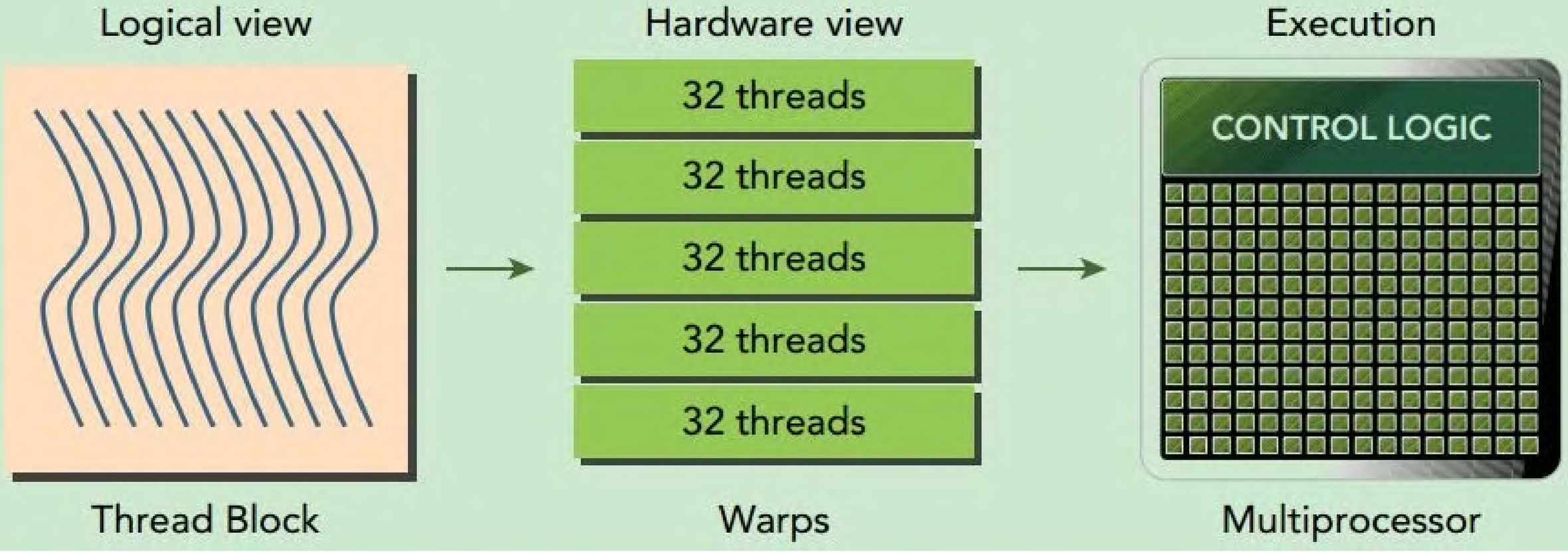
01 CUDA Basics

02 CUDA Program

03 Evaluation

CUDA Basics

Hardware, abstract, process flow



Physical HW

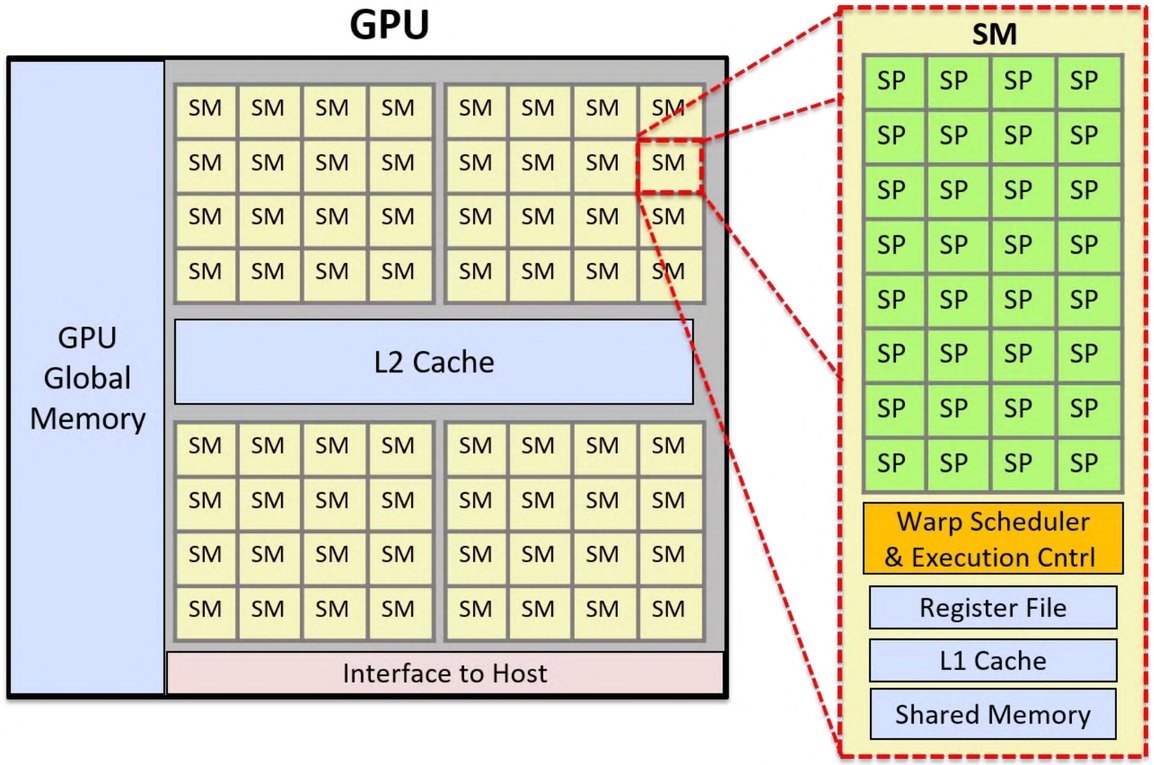
1. GPU = SMs (streaming multiprocessors)
2. SM = CUDA cores, e.g. Ampere arch SM = 128 cores

Abstract

1. 1 thread = 1 CUDA core
2. 32 threads = 1 warp (bc. memory fetch unit)

Physical hardware

overview



Processing unit

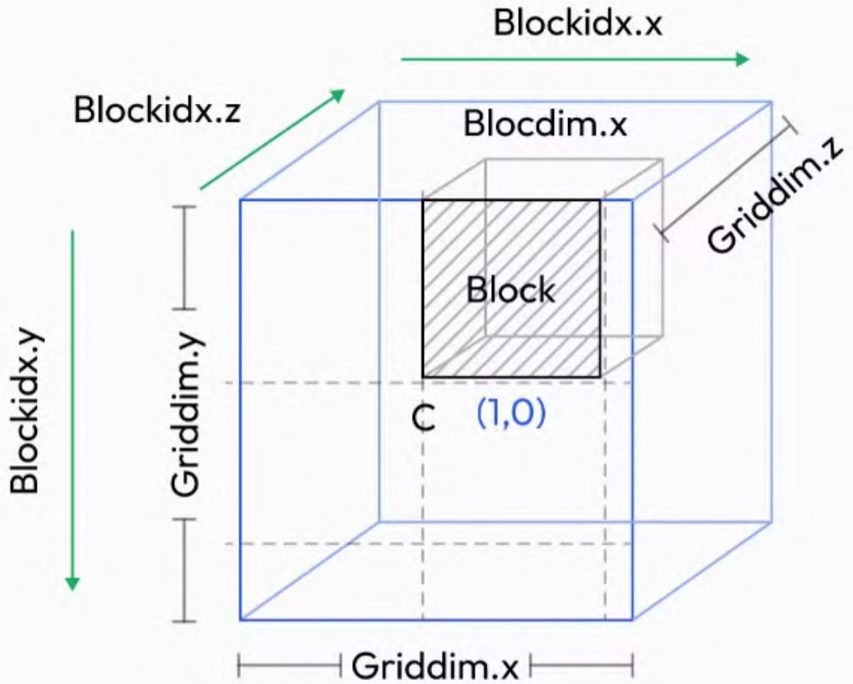
- SM → SPs
- SP (CUDA Core): Single-precision (FP32)

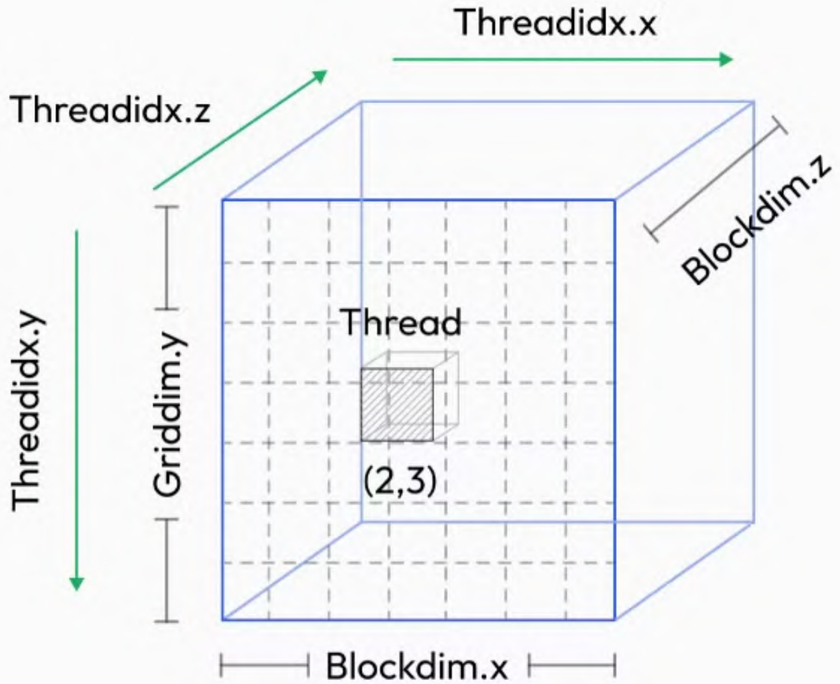
Memory

- Register：**Fastest**, private, small size.
- Shared Memory：**Fast**, shared across threads in SM.
- Global：**Slowest**, public to all, largest capacity

Cache Hierarchy

Global → L2 → L1 → Register/Share





A single thread of computation, minding its oun business

THREAD

Abstract view

Thread → A CUDA core.

Block → Threads(<1024) assigned to a SM. (share resource)

Grid → Blocks assigned to a function.

Abstract view

example code

Count required threads

Set thread shape

MatrixAdd

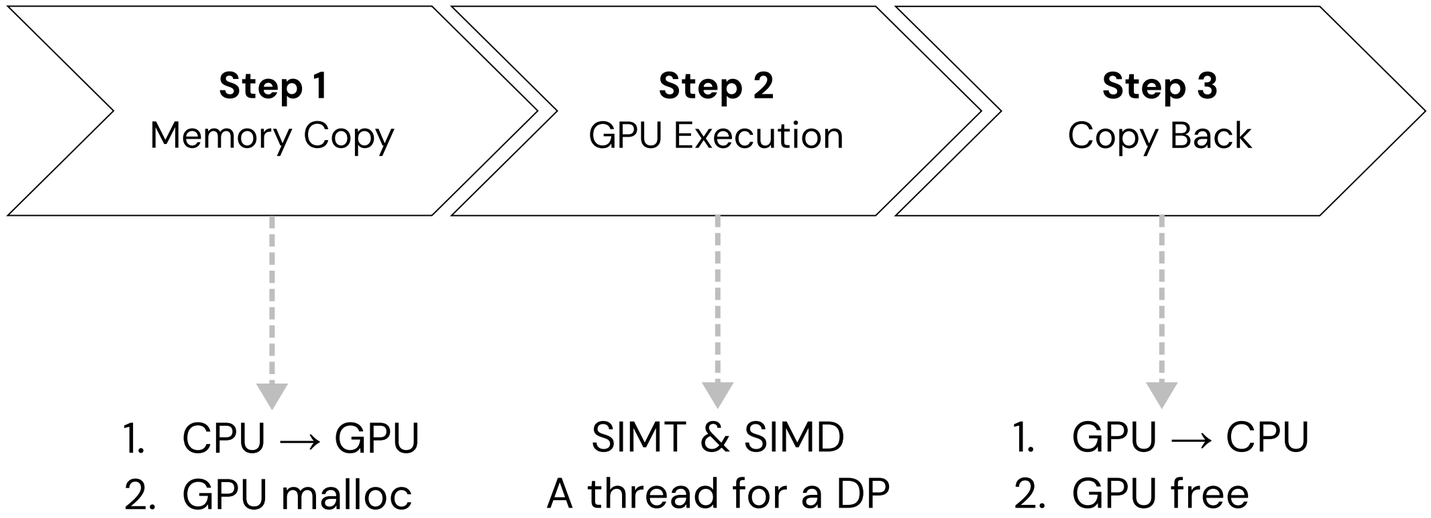
```
__global__ void matrixAdd(const float* A, const float* B,\
 float* C, int N, int M) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < M) {
        int idx = row * M + col;
        C[idx] = A[idx] + B[idx];
    }
}
int main() {
    .....
    dim3 blockDim(16, 16);
    dim3 gridDim((M + blockDim.x - 1) / blockDim.x,
                 (N + blockDim.y - 1) / blockDim.y);

    matrixAdd<<<gridDim, blockDim>>>(d_A, d_B, d_C, N, M);
    .....
}
```

- *__global__ → CPU, GPU visible*
- *dim3 → cuda data type for thread shape*

CUDA Process flow



CUDA Program

Thread indexing, kernel function

CUDA Program

design flow

Count thread index → Adjust for memory → Performance evaluate

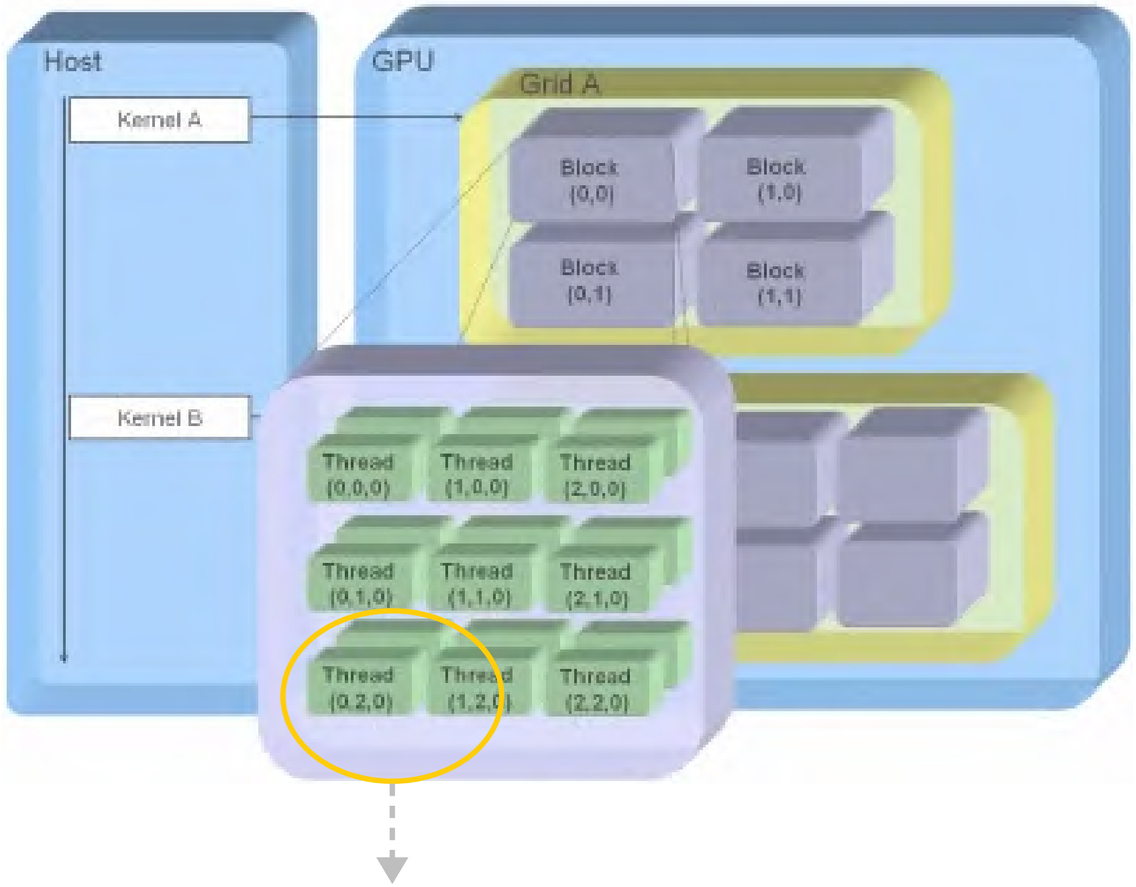
Global or Local
design according to problem

Optimize or Compromise
the problem may or may not fit coalesced style

Bottleneck
use tools to check API calling time

CUDA Program

Thread index



(0,2,0) → local index

blockdim.x * blockidx.x + (0,-,-) → Global index

Data size < 1024 → thread in a block

Data size > 1024 → thread across blocks

Global Indexing

- Thread IDs restart from 0 each block.
- Global thread index requires counting

example

Global thread counts

Nested for → 2D block
(16,16) or (256) or (16,4,4)

All data need 1 block

MatrixAdd

```
__global__ void matrixAdd(const float* A, const float* B,\
 float* C, int N, int M) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < M) {
        int idx = row * M + col;
        C[idx] = A[idx] + B[idx];
    }
}
int main() {
    .....
    dim3 blockDim(16, 16);
    dim3 gridDim((M + blockDim.x - 1) / blockDim.x,
                 (N + blockDim.y - 1) / blockDim.y);

    matrixAdd<<<gridDim, blockDim>>>(d_A, d_B, d_C, N, M);
    .....
}
```

Kernel function

Initialize

Kernel

```
int main(){
    ...
    dim3 blockDim(16,16);
    dim3 gridDim((M + blockDim.x -1) / blockDim.x,
                 (N + blockDim.y -1) / blockDim.y);

    //matrixAdd(d_A, d_B, d_C, M, N)
    matrixAdd<<<gridDim, blockDim>>>(d_A, d_B, d_C, M, N)
    ...
}
```

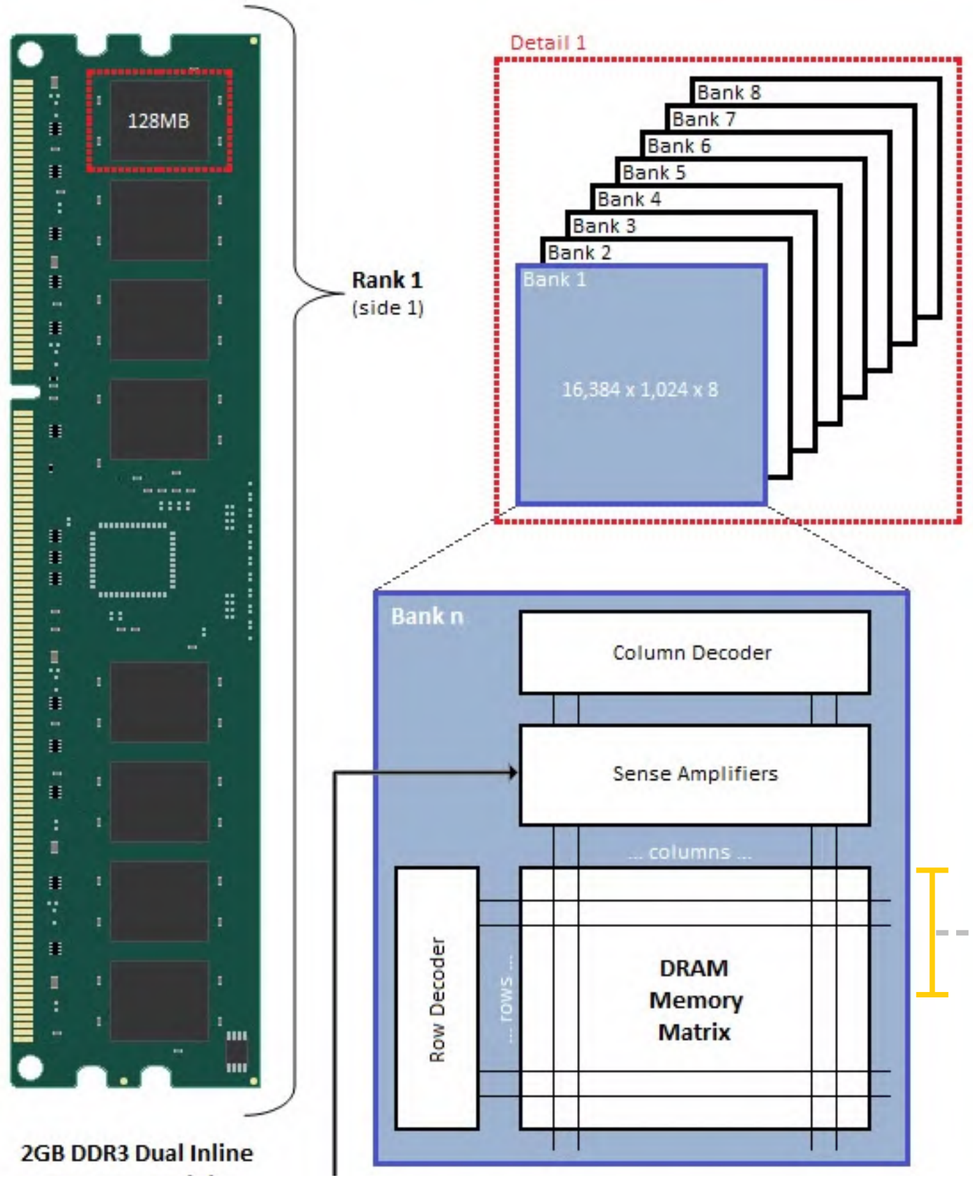
1. Copy parameters to thread registers
2. All thread has a copy

Allocate blocks to a function

Optimize memory

Access patterns, STFT Examples

Access pattern



Dram global memory

dram ic fetch 128 bytes/time

thread is FP32 → 32 threads each memory unit

(so warp = 32 threads, unit of each operation)

Access pattern

Coalesced access

32 threads match 32 address
need only 1 memory access

32 threads travel across 64 address
need 2 memory access

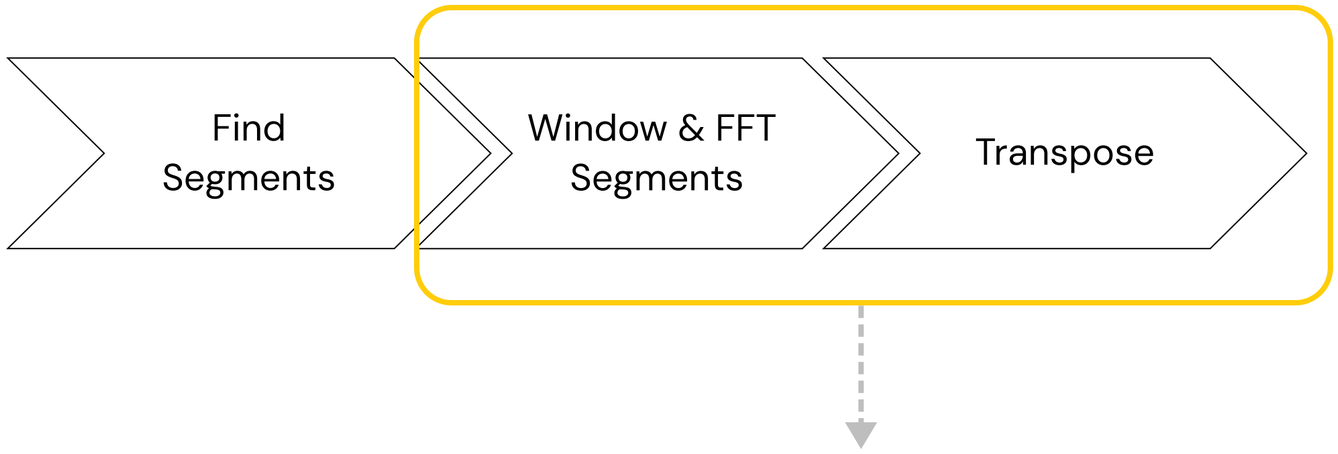
coalesced

```
__global__ void coalescedAccess(float* data, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        data[idx] = idx * 2.0f;
    }
}
__global__ void nonCoalescedAccess(float* data, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        data[idx * 2] = idx * 2.0f;
    }
}
```


index continuous → coalesced

Example

Optimize on STFT design



Inside for loop
Could accelerate with CUDA

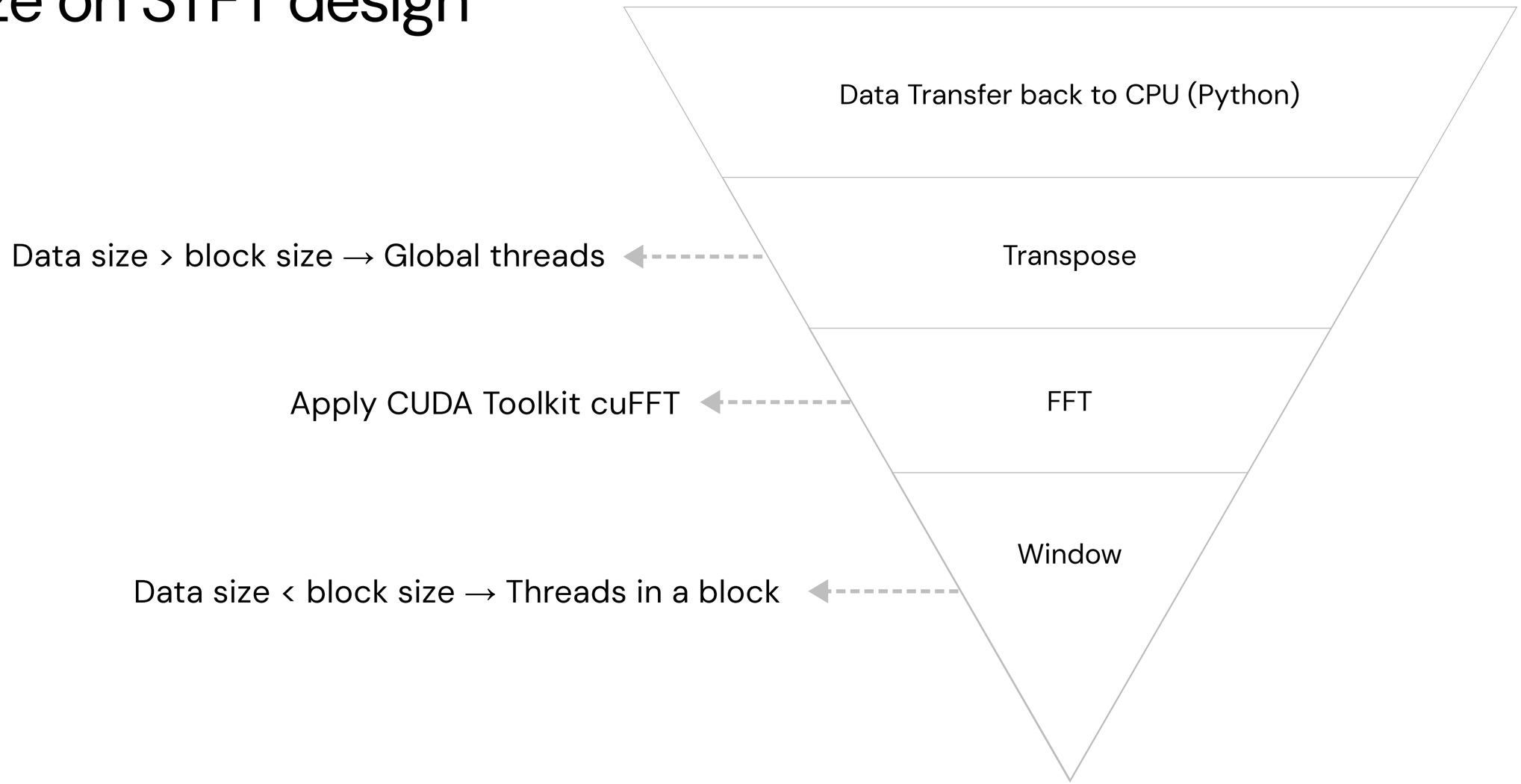
STFT python

```python
def stft_basic(Fs, x, w, H, only_positive=False):
    N = len(w)
    L = len(x)
    M = np.floor((L - N) / H).astype(int) + 1
    X = np.zeros((N, M), dtype='complex')
    for m in range(M):
        x_win = x[m * H:m * H + N] * w
        X_win = np.fft.fft(x_win) / Fs
        X[:, m] = X_win

    if only_positive:
        K = 1 + N // 2
        X = X[0:K, :]
    return X
```

Example

Optimize on STFT design



Example

Non-coalescing

Using local thread index
(bc fft_size = 256 < 1024)

Non-coalescing access for hop > 1

Continuous indexing → Coalescing access

process segments

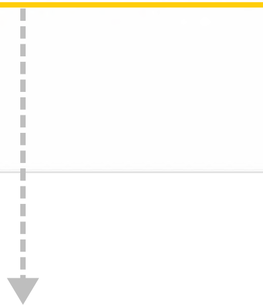
```
__global__ void process_segs(float* input, cuFloatComplex* output,
                             float* hanning_w,
                             int sig_len, int w_size,
                             int hop, int fft_size,
                             int segs) {
    int m = blockIdx.x;
    int t = threadIdx.x;

    if (m < segs && t < w_size) {
        int i = m * hop + t;
        if (i < sig_len) {
            float sample = input[i] * hanning_w[t];
            output[m * fft_size + t].x = sample;
            output[m * fft_size + t].y = 0.0f;
        }
    }
}
```

Transpose

```
__global__ void transpose(cuFloatComplex* d_output, cuFloatComplex* temp, int segs,
int fft_size) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i < segs && j < (fft_size/2 + 1)) {
        temp[j * segs + i] = d_output[i * (fft_size/2 + 1) + j];
    }
}
```



non-coalesced access

Evaluation

Runtime, nvprof

Runtime

Signal length	CUDA	Python
1000 (data points)	0.00041(s)	0.000027(s)
10000 (data points)	0.00046(s)	0.000342(s)
100000 (data points)	0.00076(s)	0.003372(s)
1000000 (data points)	0.00434(s)	0.034968(s)

Paralleled

- ☑ windowing
- ☑ transpose

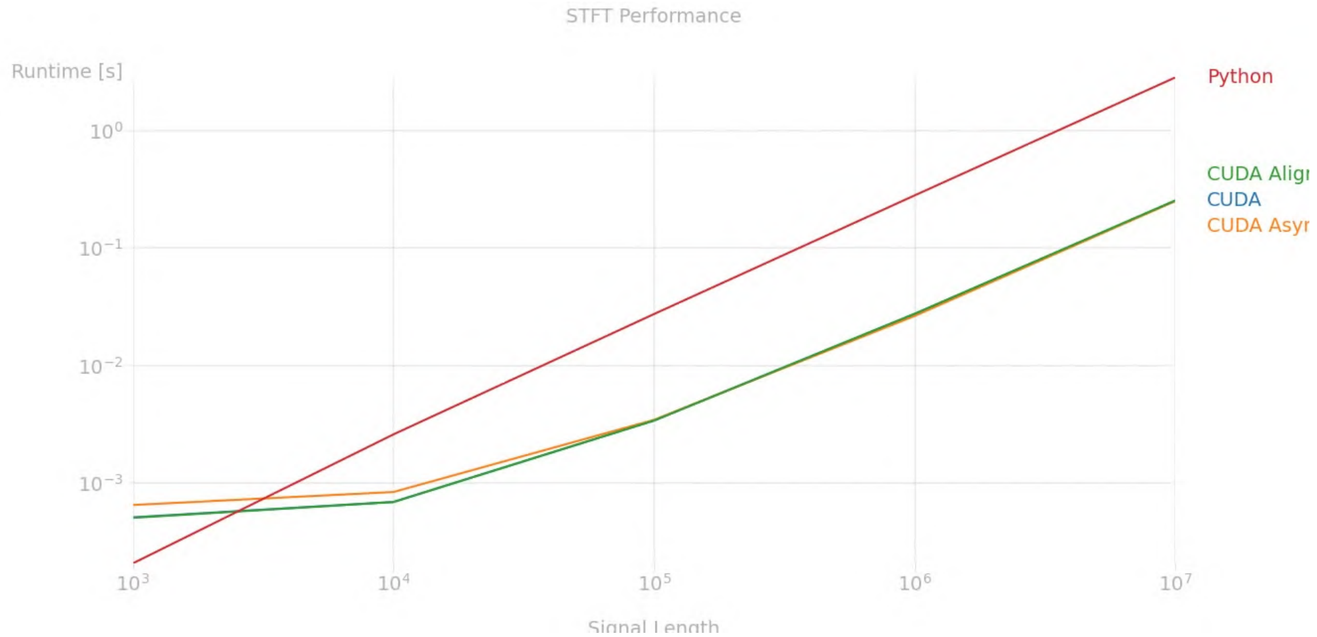
Parameters

- hop size = 256
- window length = 256

```
==2356191== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   88.60%  642.60ms      3852  166.82us  1.2150us  2.4800ms  [CUDA mem
                    3.49%  25.341ms     11556  2.1920us     703ns  18.400us  [CUDA mem
                    2.31%  16.757ms      3852  4.3500us  2.9110us  27.200us  void vect
padding_t=8, twiddle_t=0, loadstore_modifier_t=2, layout_t=0, unsigned int, float>(ker
                    1.91%  13.866ms      3852  3.5990us  2.2410us  23.712us  normalize
                    1.70%  12.359ms      3852  3.2080us  1.9830us  26.880us  transpose
                    1.02%  7.3806ms      3852  1.9160us  1.5040us  3.7120us  process_s
                    0.96%  6.9888ms      7706     906ns     287ns  2.2624ms  [CUDA mem
      API calls:   31.94%  795.91ms     11556  68.873us     680ns  2.9050ms  cudaMemcp
                   28.65%  713.85ms      3852  185.32us  173.07us  643.20us  cuModuleL
                   12.65%  315.18ms     26964  11.688us  1.2250us  80.543ms  cudaMallo
                   10.03%  249.97ms      3852  64.892us  56.587us  9.0683ms  cuModuleU
                    4.35%  108.38ms     19260  5.6270us     443ns  747.26us  cudaFree
                    2.44%  60.709ms     11556  5.2530us     980ns  1.0706ms  cudaDevic
                    2.03%  50.536ms     11556  4.3730us  2.8900us  189.91us  cudaLaunc
```

nvprof Output
Memory copy from CPU to GPU → 31.94%
Memory allocation on GPU → 12.65%

Runtime

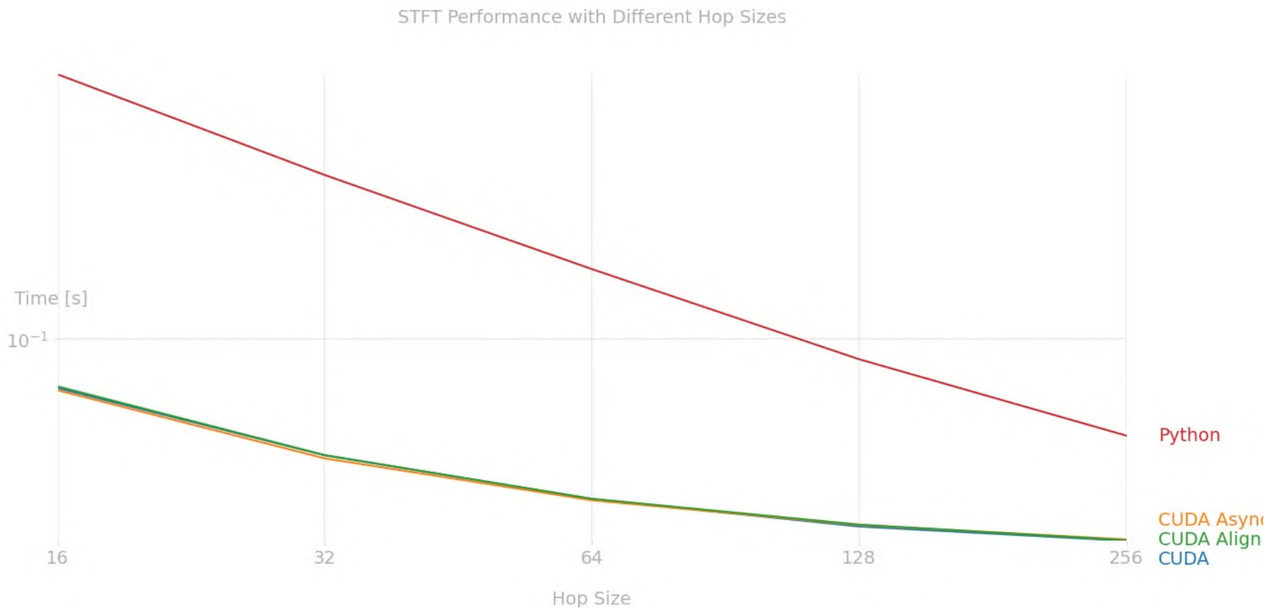


`more parallel, with smaller hop size`

- hop size = 32
- window size = 256

Signal length	CUDA	Python
1000 (data points)	0.000508(s)	0.000210(s)
10^4 (data points)	0.000697(s)	0.002519(s)
10^5 (data points)	0.003392(s)	0.027379(s)
10^6 (data points)	0.027439(s)	0.28058(s)
10^7 (data points)	0.251263(s)	2.819038(s)

Runtime

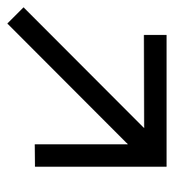


Hop Size	CUDA	Python
256 (data points)	0.02621(s)	0.05317(s)
128 (data points)	0.02872(s)	0.08897(s)
64 (data points)	0.03443(s)	0.16025(s)
32 (data points)	0.04601(s)	0.30234(s)
16 (data points)	0.07297(s)	0.58617(s)

`more parallel, compare different hop size`

- signal length = 10^6
- window size = 256

End



References

https://cuda-programming.blogspot.com/2013/02/bank-conflicts-in-shared-memory-in-cuda.html

https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

https://www.nurix.ai/blogs/gpus-part-2---understanding-the-gpu-programming-model

https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming)#:~:text=Blocks%20can%20be%20organized%20into,from%20the%20maximum%20grid%20dimensions.

https://alpaka.readthedocs.io/en/latest/basic/abstraction.html