GPU Programming Basics

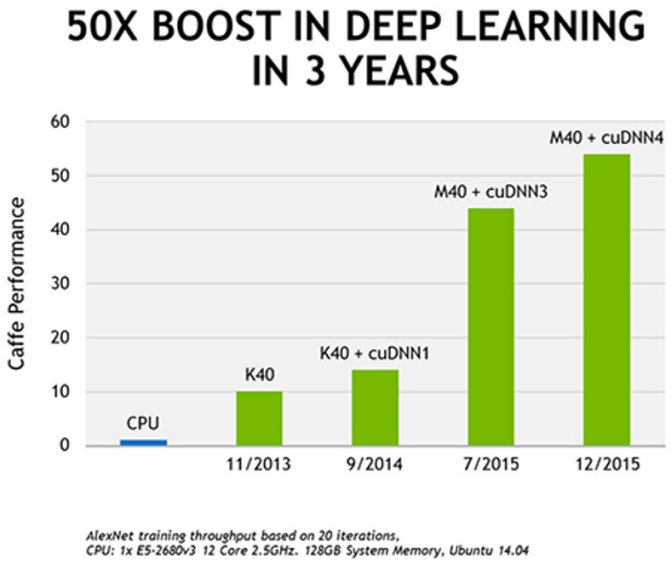
Zizheng Guo

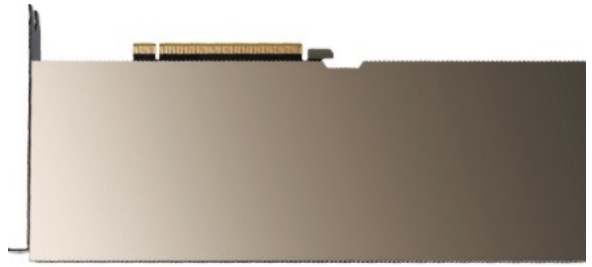
gzz@pku.edu.cn

2022.1

Why GPU?

- Fast
- Universal
- Easy



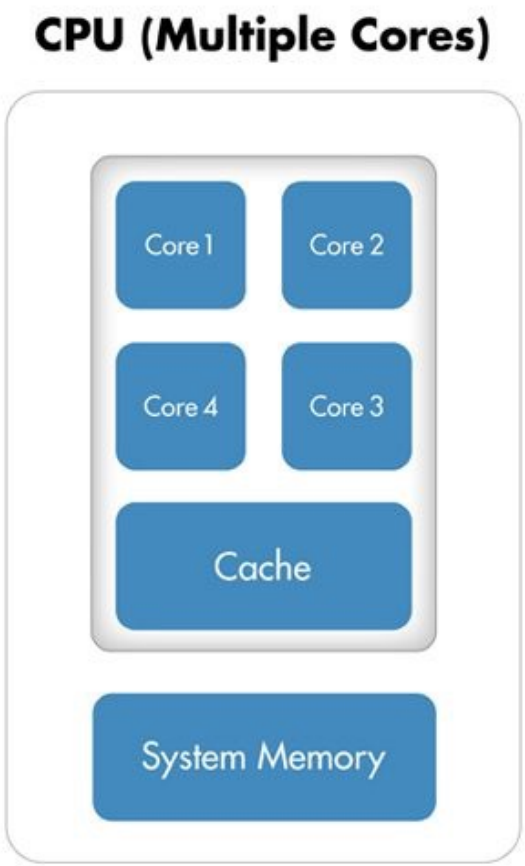


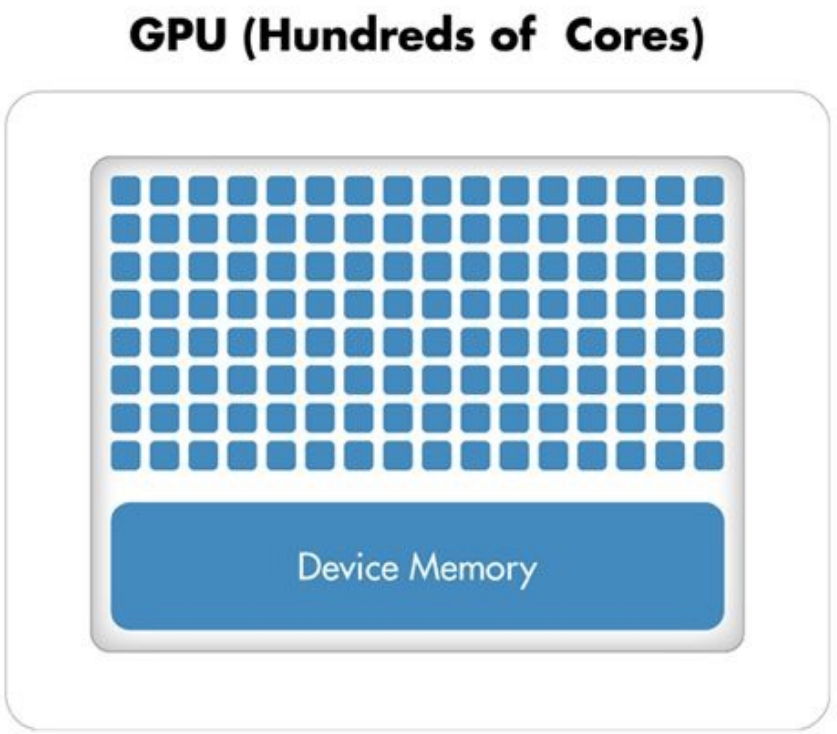
Jensen Huang: “NVIDIA是一家软件公司”

courtesy:
https://blogs.nvidia.com/blog/2016/01/12/accelerating-ai-artificial-intelligence-gpus/
https://www.google.com/search?q=nvidia+stock

What is GPU?

- **Easy tasks**
- **Massive parallelism**





How to write GPU code?

- For example: A[i] += B[i] * C[i]

```
__global__ void muladd(int size, float *a, float *b, float *c) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  a[i] += b[i] * c[i];
}
```

```
long int start = timestamp();
for(int i = 0; i < 100; ++i) {
  muladd<<<SIZE / 256, 256>>>(SIZE, gpu_a, gpu_b, gpu_c);
}
checkCUDA(cudaDeviceSynchronize());
long int end = timestamp();
printf("Time: %ld ms\n", end - start);
```

Compile and run it, so easy

- NVCC: cuda compiler

```
(base) [gzz2000@sccc example]$ nvcc muladd.cu -o muladd_cuda -O2
(base) [gzz2000@sccc example]$ ./muladd_cuda
Time: 78 ms
(base) [gzz2000@sccc example]$
```

- Compared to CPU version (40 threads):

```
(base) [gzz2000@sccc example]$ ./muladd_cpu
Time: 963 ms
(base) [gzz2000@sccc example]$
```

- 12X speed-up

Some details

```
#define checkCUDA(ret) assert((ret) == cudaSuccess)
```

- GPU memory allocation

```
checkCUDA(cudaMalloc(&gpu_a, sizeof(cpu_a)));
checkCUDA(cudaMalloc(&gpu_b, sizeof(cpu_b)));
checkCUDA(cudaMalloc(&gpu_c, sizeof(cpu_c)));
```

- Copy CPU->GPU

```
checkCUDA(cudaMemcpy(gpu_a, cpu_a, sizeof(cpu_a), cudaMemcpyHostToDevice));
checkCUDA(cudaMemcpy(gpu_a, cpu_a, sizeof(cpu_a), cudaMemcpyHostToDevice));
checkCUDA(cudaMemcpy(gpu_a, cpu_a, sizeof(cpu_a), cudaMemcpyHostToDevice));
```

- Copy back and synchronize

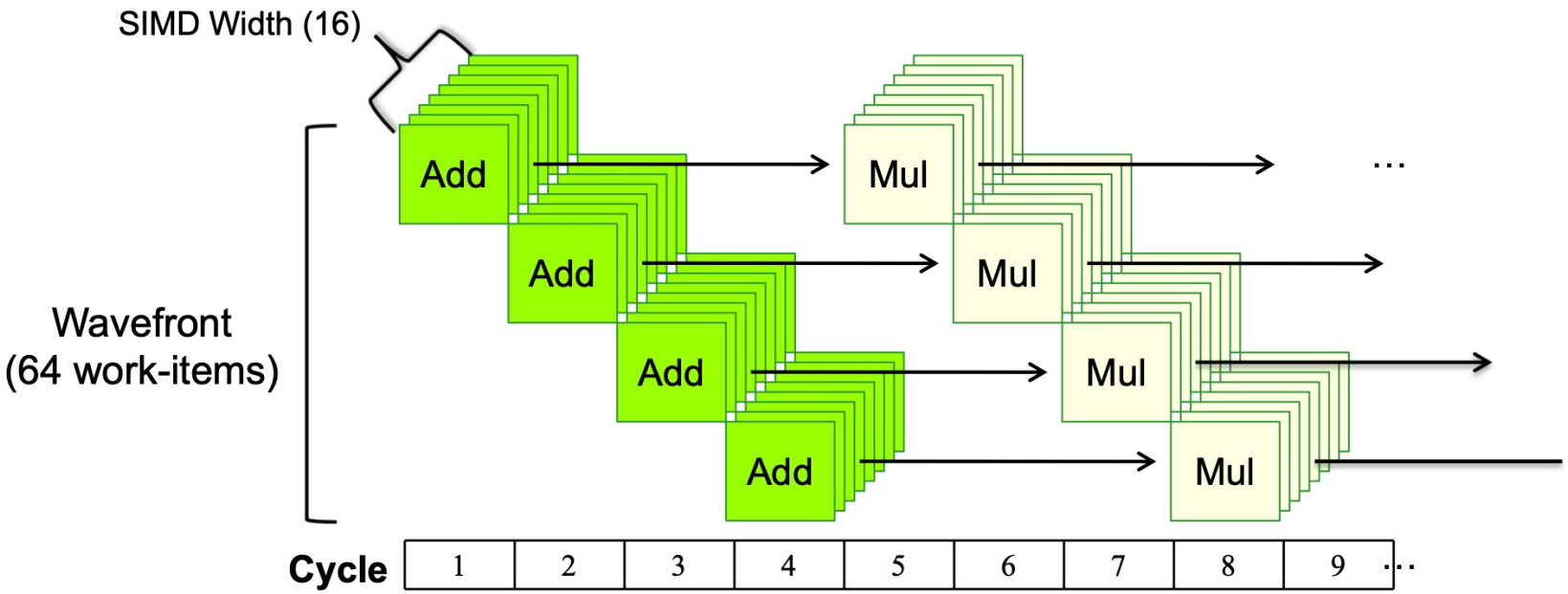
```
checkCUDA(cudaMemcpy(cpu_a, gpu_a, sizeof(cpu_a), cudaMemcpyDeviceToHost));
checkCUDA(cudaDeviceSynchronize());
```

- Free GPU memory

```
checkCUDA(cudaFree(gpu_a));
checkCUDA(cudaFree(gpu_b));
checkCUDA(cudaFree(gpu_c));
```

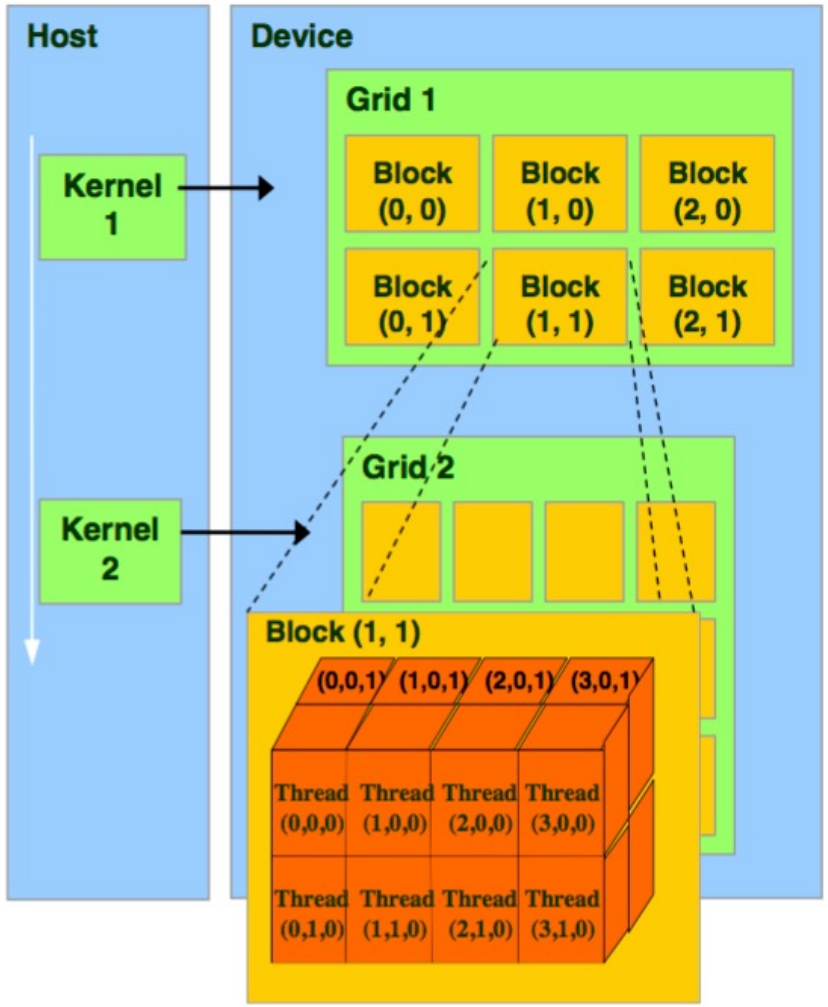
Why is GPU so fast?

- Can we use a 10000 cores CPU instead?
  - No, because memory latency dominates.
- Use parallelization to hide latency -- that is GPU.



Courtesy: Guojie Luo

CUDA programming language



- Blocks and threads

```
for(int i = 0; i < 100; ++i) {
  muladd<<<SIZE / 256, 256>>>(SIZE, gpu_a, gpu_b, gpu_c);
}

__global__ void muladd(int size, float *a, float *b, float *c) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  a[i] += b[i] * c[i];
}
```

- All **threads** in a block run in parallel
- All **blocks** run parallel in batches

Hardware behind the scene…

- Streaming multiprocessor <-> blocks
  - Multiple blocks on a single SM
- Streaming processor/cuda core <-> threads

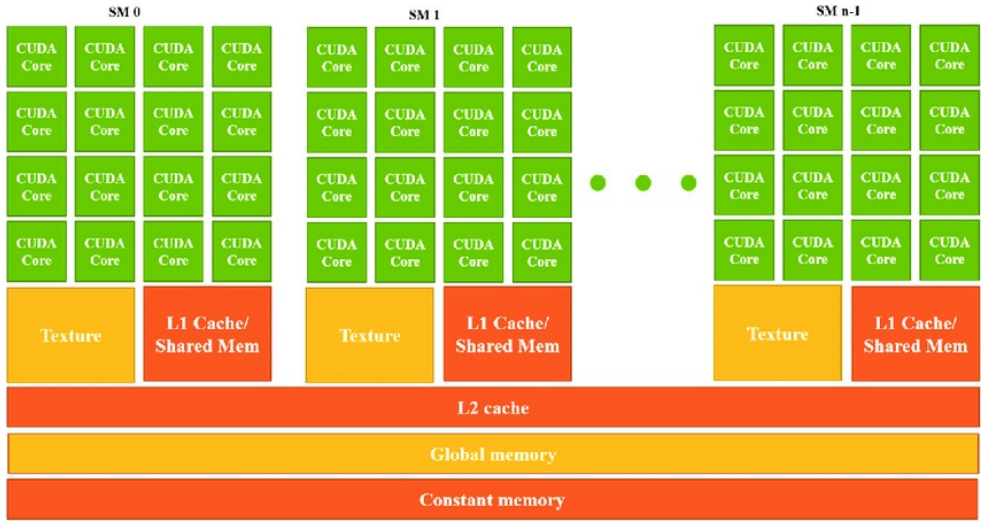
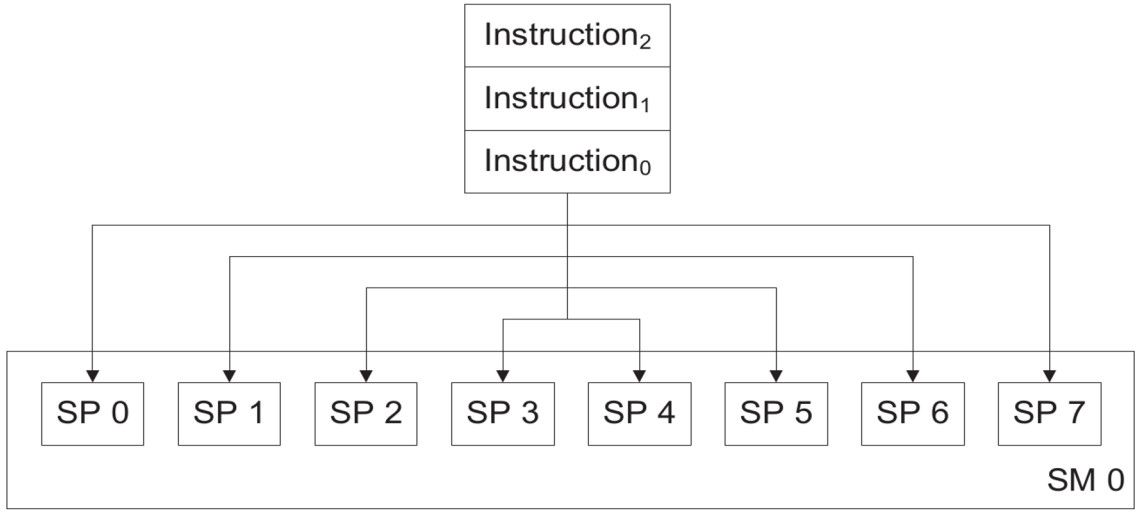


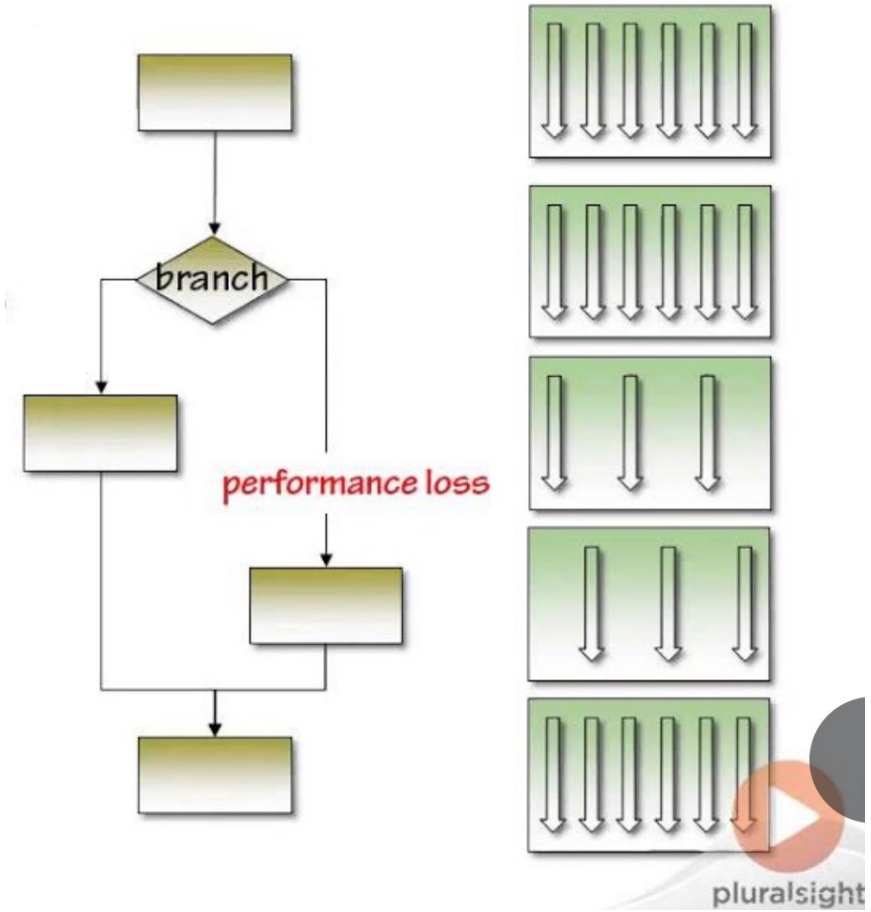
Table 5.1 Utilization %

Threads per Block/ Compute Capability	1.0	1.1	1.2	1.3	2.0	2.1	3.0
64	67	67	50	50	33	33	50
96	100	100	75	75	50	50	75
128	100	100	100	100	67	67	100
192	100	100	94	94	100	100	94
256	100	100	100	100	100	100	100
384	100	100	75	75	100	100	94
512	67	67	100	100	100	100	100
768	N/A	N/A	N/A	N/A	100	100	75
1024	N/A	N/A	N/A	N/A	67	67	100

Single instruction multiple threads (SIMT)

- Threads run in **lock-step**
- Branch divergence





CPU-GPU scheduling

- Launch
  - Does **not** block CPU execution

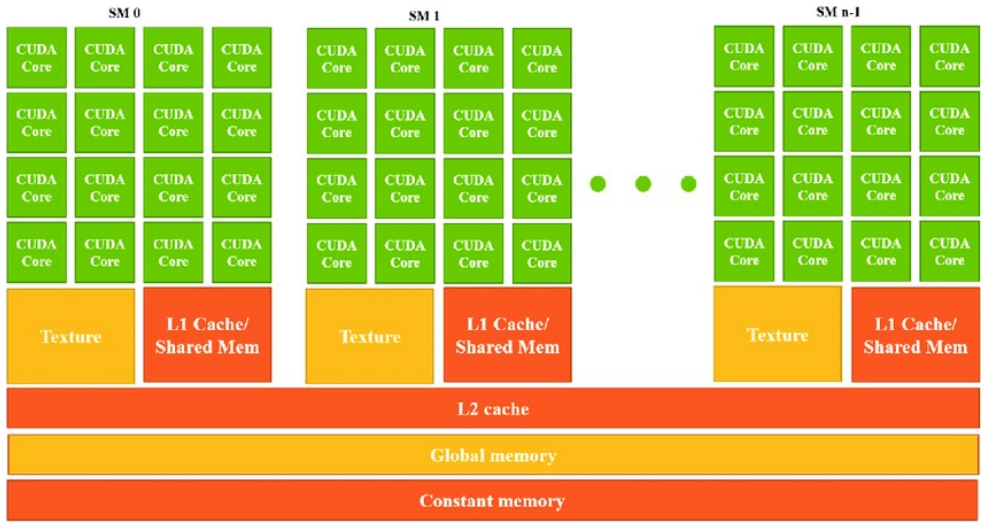
```
muladd<<<SIZE / 256, 256>>>(SIZE, gpu_a, gpu_b, gpu_c);
```

- Synchronize
  - Blocks CPU execution until all GPU activities finish

```
cudaDeviceSynchronize();
```

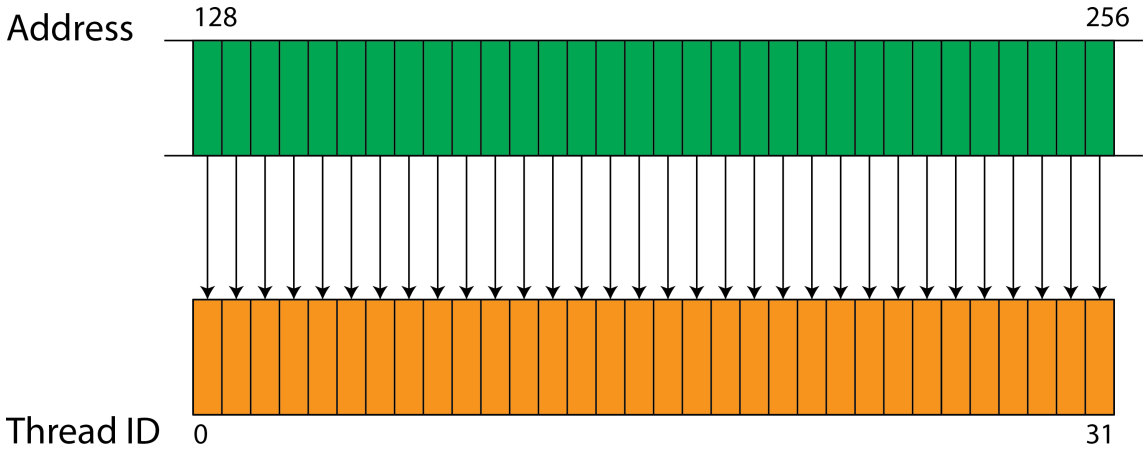
GPU memory model

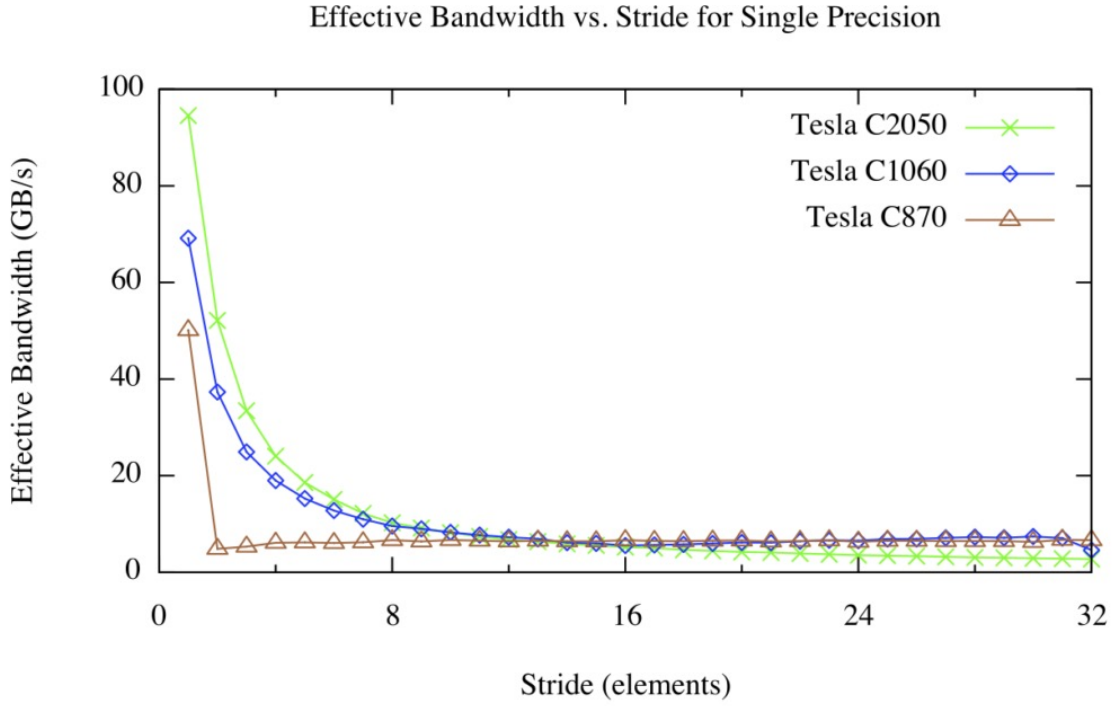
- *CPU memory: DDR*
  - Per Host
- Global memory: GDDR
  - Per GPU
- Shared memory: controllable L1 cache
  - Per block
- Local memory, registers
  - Per thread



Global memory coalescing

- Continuous (and aligned) accesses are faster.
  - Combined into a single query.





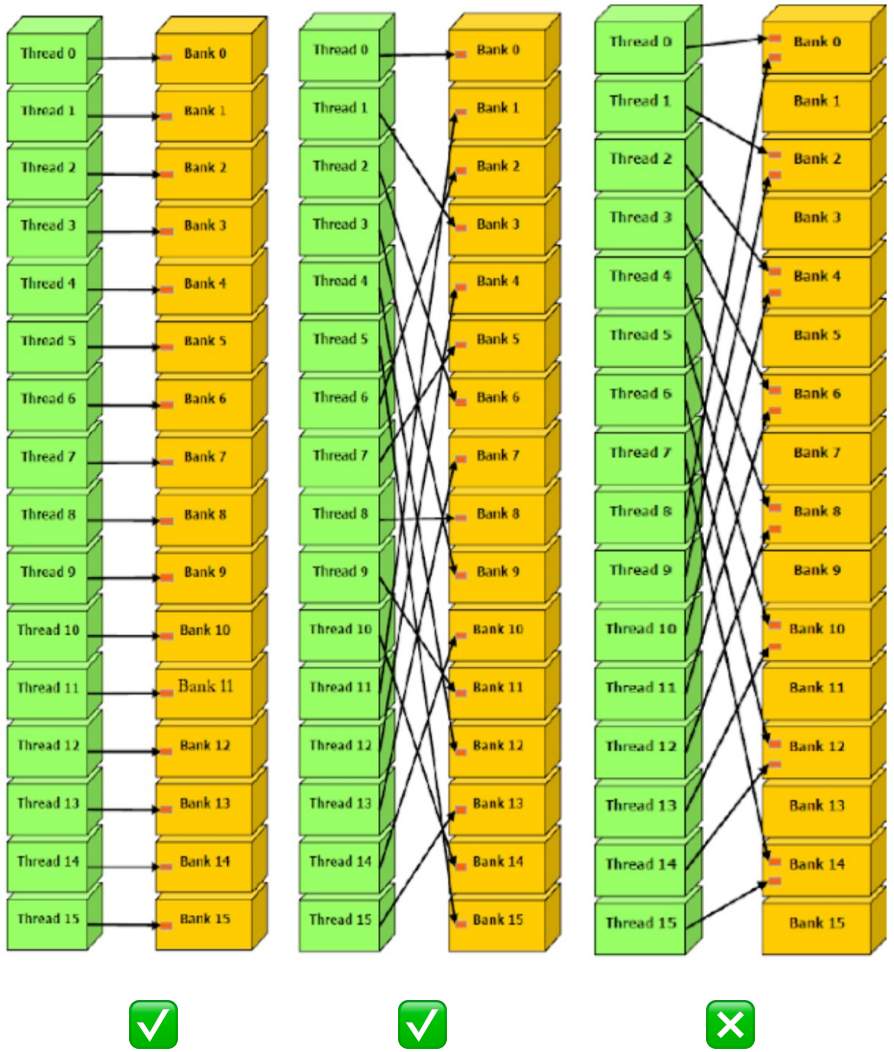
Courtesy https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/

Shared memory and bank access

- Per-block controllable L1 cache

```
__global__ void xxxxx(xxx) {
  __shared__ int data[128];
```

- High bandwidth, low latency
- Small
- Caution: bank conflict
  - Bank size: 4 bytes
  - #Banks: 32
- **Broadcast is also fast**

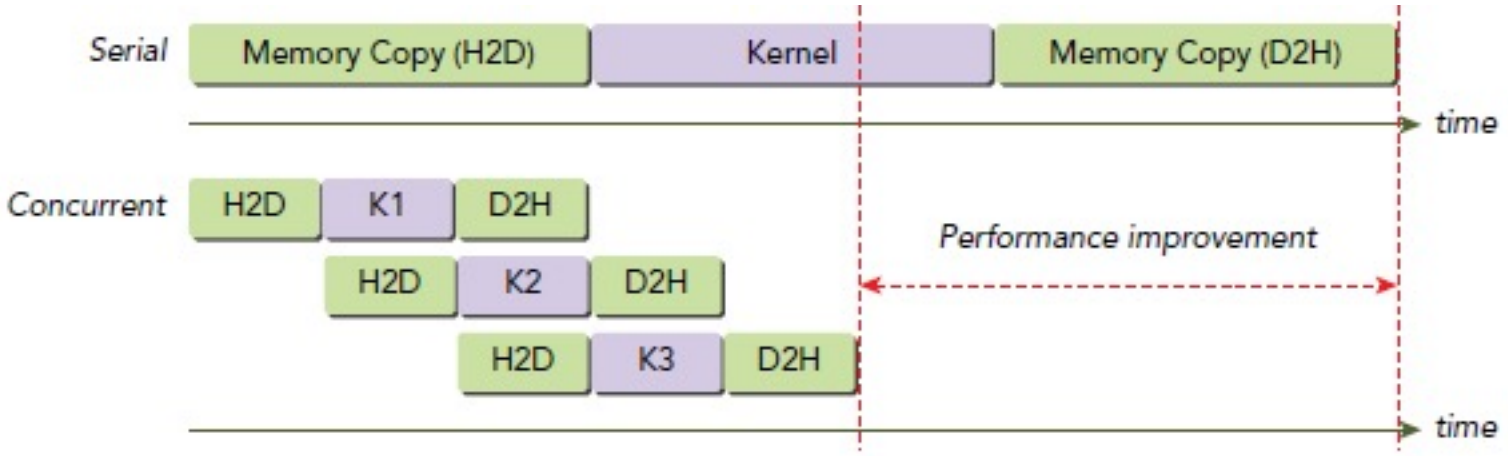


Write efficient GPU code

- Manage CPU-GPU data transfer
- Choose appropriate block size
- Loop unrolling
- Coalesce memory access
- Avoid atomic ops
- Use libraries like Thrust, CUB, …

Manage CPU-GPU data transfer

- These things can be done in parallel:
  - Computation on each single GPU
  - Computation on CPU
  - CPU-GPU data transfer



Manage CPU-GPU data transfer

- with pinned memory
  - **cudaMallocHost / cudaFreeHost**
  - Non-paged memory, slower allocation
  - faster and asynchronous transfer **cudaMemcpyAsync**
- and CUDA streams
  - cudaStreamCreate / cudaStreamDestroy
  - Kernel<<<block, thread, sharedmem, **stream**>>>(params)
  - cudaStreamSynchronize
- You can even use **multiple GPUs** with CUDA streams.

Atomic operations

- atomic{Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor}
- Returns old memory content, sometimes useful.
- Mostly, only int/uint supported.
- Large overhead
  - Even larger if same memory position is visited
- Clever ways to avoid (we'll see it at once)
  - Per-block temp array
  - 2-way merging of threads

Example: Histogram

- Problem size:
  - array: 512MB of u_char
  - bins count: 256

```
for(int i = 0; i < ARRSZ; ++i) {
  ++bin[arr[i]];
}
```

Histogram: cpu naive

```
void compute() {
  for(int i = 0; i < ARRSZ; ++i) {
    ++bin[arr[i]];
  }
}
```

Histogram: cpu parallel ??

```
void compute() {
#pragma omp parallel for num_threads(40)
  for(int i = 0; i < ARRSZ; ++i) {
    int c = arr[i];
#pragma omp atomic
    ++bin[c];
  }
}
```

Histogram: cpu parallel 2

```
void compute() {
#pragma omp parallel for num_threads(40)
  for(int t = 0; t < 40; ++t) {
    int tmp_bin[256] = {};
    for(int i = t; i < ARRSZ; i += 40) {
      int c = arr[i];
      ++tmp_bin[c];
    }
    for(int j = 0; j < 256; ++j) {
      int a = tmp_bin[j];
#pragma omp atomic
      bin[j] += a;
    }
  }
}
```

Each thread processes elements with a fixed step.

Histogram: cpu parallel 2 & 2.5

```
void compute() {
#pragma omp parallel for num_threads(40)
  for(int t = 0; t < 40; ++t) {
    int tmp_bin[256] = {};
    for(int i = t; i < ARRSZ; i += 40) {
      int c = arr[i];
      ++tmp_bin[c];
    }
    for(int j = 0; j < 256; ++j) {
      int a = tmp_bin[j];
#pragma omp atomic
      bin[j] += a;
    }
  }
}
```

Each thread processes elements with a fixed step.

```
void compute() {
  int blk = (ARRSZ + 39) / 40;
#pragma omp parallel for num_threads(40)
  for(int t = 0; t < 40; ++t) {
    int tmp_bin[256] = {};
    int l = blk * t, r = blk * (t + 1);
    if(r > ARRSZ) r = ARRSZ;
    for(int i = l; i < r; ++i) {
      int c = arr[i];
      ++tmp_bin[c];
    }
    for(int j = 0; j < 256; ++j) {
      int a = tmp_bin[j];
#pragma omp atomic
      bin[j] += a;
    }
  }
}
```

Each thread processes a contiguous range of elements. [Faster, why?]

Histogram: naive cuda

```
__global__ void histogram_kernel_v1(unsigned char *array, unsigned int *bins) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  // we do not need to check boundary here as ARRSZ is a power of 2.

  atomicAdd(&bins[array[tid]], 1u);
}

void compute() {
  const int block_size = 256;
  histogram_kernel_v1<<<ARRSZ / block_size, block_size>>>(arr_gpu, bin_gpu);
  assert(cudaSuccess == cudaDeviceSynchronize());
}
```

Histogram: naive cuda + unroll

No improvement, why?

```
__global__ void histogram_kernel_v2(unsigned int *array, unsigned int *bins) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;

  unsigned int value_u32 = array[tid];
  atomicAdd(&bins[value_u32 & 0x000000FF], 1u);
  atomicAdd(&bins[(value_u32 & 0x0000FF00) >> 8], 1u);
  atomicAdd(&bins[(value_u32 & 0x00FF0000) >> 16], 1u);
  atomicAdd(&bins[(value_u32 & 0xFF000000) >> 24], 1u);
}

void compute() {
  const int block_size = 256;
  histogram_kernel_v2<<<ARRSZ / block_size / 4, block_size>>>((unsigned int *)arr_
gpu, bin_gpu);
  assert(cudaSuccess == cudaDeviceSynchronize());
}
```

Histogram: shared memory to reduce global mem access

```
__shared__ unsigned int bins_shared[256];

__global__ void histogram_kernel_v3(unsigned int *array, unsigned int *bins) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  bins_shared[threadIdx.x] = 0; // block size assumed to be just 256
  __syncthreads();

  unsigned int value_u32 = array[tid];
  atomicAdd(&bins_shared[value_u32 & 0x000000FF], 1u);
  atomicAdd(&bins_shared[(value_u32 & 0x0000FF00) >> 8], 1u);
  atomicAdd(&bins_shared[(value_u32 & 0x00FF0000) >> 16], 1u);
  atomicAdd(&bins_shared[(value_u32 & 0xFF000000) >> 24], 1u);
  __syncthreads();

  atomicAdd(&bins[threadIdx.x], bins_shared[threadIdx.x]);
}

void compute() {
  const int block_size = 256; // do not change this
  histogram_kernel_v3<<<ARRSZ / block_size / 4, block_size>>>((unsigned int *)arr_
gpu, bin_gpu);
  assert(cudaSuccess == cudaDeviceSynchronize());
}
```

Histogram: shared memory, no unroll

```
__shared__ unsigned int bins_shared[256];

__global__ void histogram_kernel_v3_5(unsigned char *array, unsigned int *bins) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  bins_shared[threadIdx.x] = 0; // block size assumed to be just 256
  __syncthreads();

  atomicAdd(&bins_shared[array[tid]], 1u);
  __syncthreads();

  atomicAdd(&bins[threadIdx.x], bins_shared[threadIdx.x]);
}

void compute() {
  const int block_size = 256; // do not change this
  histogram_kernel_v3_5<<<ARRSZ / block_size, block_size>>>(arr_gpu, bin_gpu);
  assert(cudaSuccess == cudaDeviceSynchronize());
}
```

Histogram: shared memory, more unroll: N=32*4

```
__shared__ unsigned int bins_shared[256];
const int N = 32;

__global__ void histogram_kernel_v4(unsigned int *array, unsigned int *bins) {
  int toffset = blockIdx.x * blockDim.x + threadIdx.x;
  int sz = gridDim.x * blockDim.x;

  bins_shared[threadIdx.x] = 0; // block size assumed to be just 256
  __syncthreads();

  for(int i = 0, tid = toffset; i < N; ++i, tid += sz) {
    unsigned int value_u32 = array[tid];
    atomicAdd(&bins_shared[value_u32 & 0x000000FF], 1u);
    atomicAdd(&bins_shared[(value_u32 & 0x0000FF00) >> 8], 1u);
    atomicAdd(&bins_shared[(value_u32 & 0x00FF0000) >> 16], 1u);
    atomicAdd(&bins_shared[(value_u32 & 0xFF000000) >> 24], 1u);
  }
  __syncthreads();

  atomicAdd(&bins[threadIdx.x], bins_shared[threadIdx.x]);
}

void compute() {
  const int block_size = 256; // do not change this
  histogram_kernel_v4<<<ARRSZ / block_size / 4 / N, block_size>>>((unsigned int *)
arr_gpu, bin_gpu);
  assert(cudaSuccess == cudaDeviceSynchronize());
}
```

Each thread processes elements with a fixed step.

Histogram: shared memory, contiguous unroll 32*4

Each thread processes a contiguous range of elements. [SLOWER, why?]

```
__shared__ unsigned int bins_shared[256];
const int N = 32;

__global__ void histogram_kernel_v4_5(unsigned int *array, unsigned int *bins) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  bins_shared[threadIdx.x] = 0; // block size assumed to be just 256
  __syncthreads();

  int l = i * N, r = (i + 1) * N;
  for(int tid = l; tid < r; ++tid) {
    unsigned int value_u32 = array[tid];
    atomicAdd(&bins_shared[value_u32 & 0x000000FF], 1u);
    atomicAdd(&bins_shared[(value_u32 & 0x0000FF00) >> 8], 1u);
    atomicAdd(&bins_shared[(value_u32 & 0x00FF0000) >> 16], 1u);
    atomicAdd(&bins_shared[(value_u32 & 0xFF000000) >> 24], 1u);
  }
  __syncthreads();

  atomicAdd(&bins[threadIdx.x], bins_shared[threadIdx.x]);
}

void compute() {
  const int block_size = 256; // do not change this
  histogram_kernel_v4_5<<<ARRSZ / block_size / 4 / N, block_size>>>((unsigned int
*)arr_gpu, bin_gpu);
  assert(cudaSuccess == cudaDeviceSynchronize());
}
```

Histogram

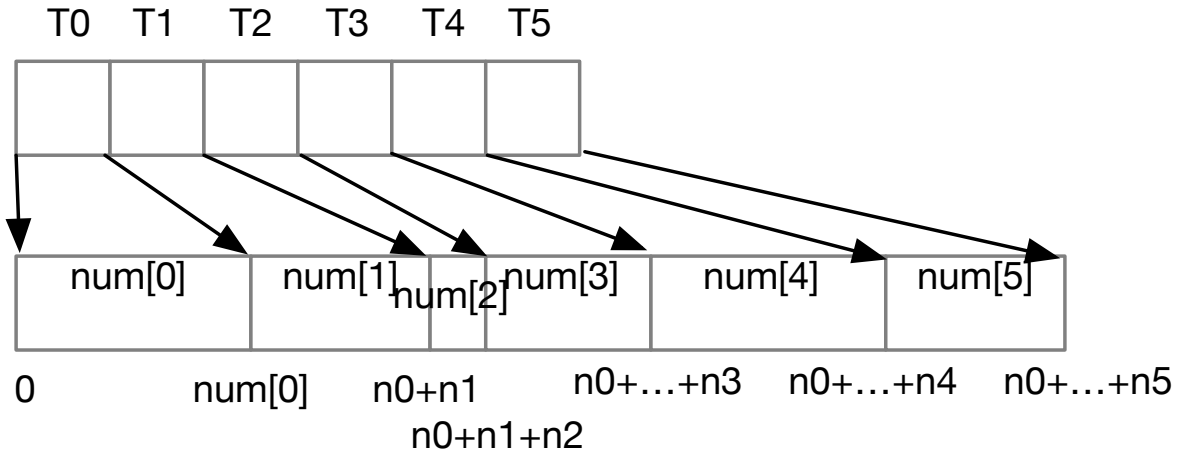
Reference code by GZZ is at /opt/share/cuda_training/histogram/

cpu.cpp: naive, 1.5s (-O2: 350ms)
cpu_parallel: naive parallel, worse, 6s (-O2: 6s)
cpu_parallel2: scatter worker thread, 200ms (-O2: 200ms)
cpu_parallel2.5: contiguous worker thread, 117ms (-O2: 45ms)

cuda: naive gpu, 213ms
cuda_v2: naive with batching, still 200ms
cuda_v3: shared memory with batching, 9ms
cuda_v3.5: shared memory without batching, 33ms
cuda_v4: shared memory with scatter worker thread, 0ms
cuda_v4.5: shared memory with contiguous worker thread: 2ms

GPU-friendly algorithm tricks

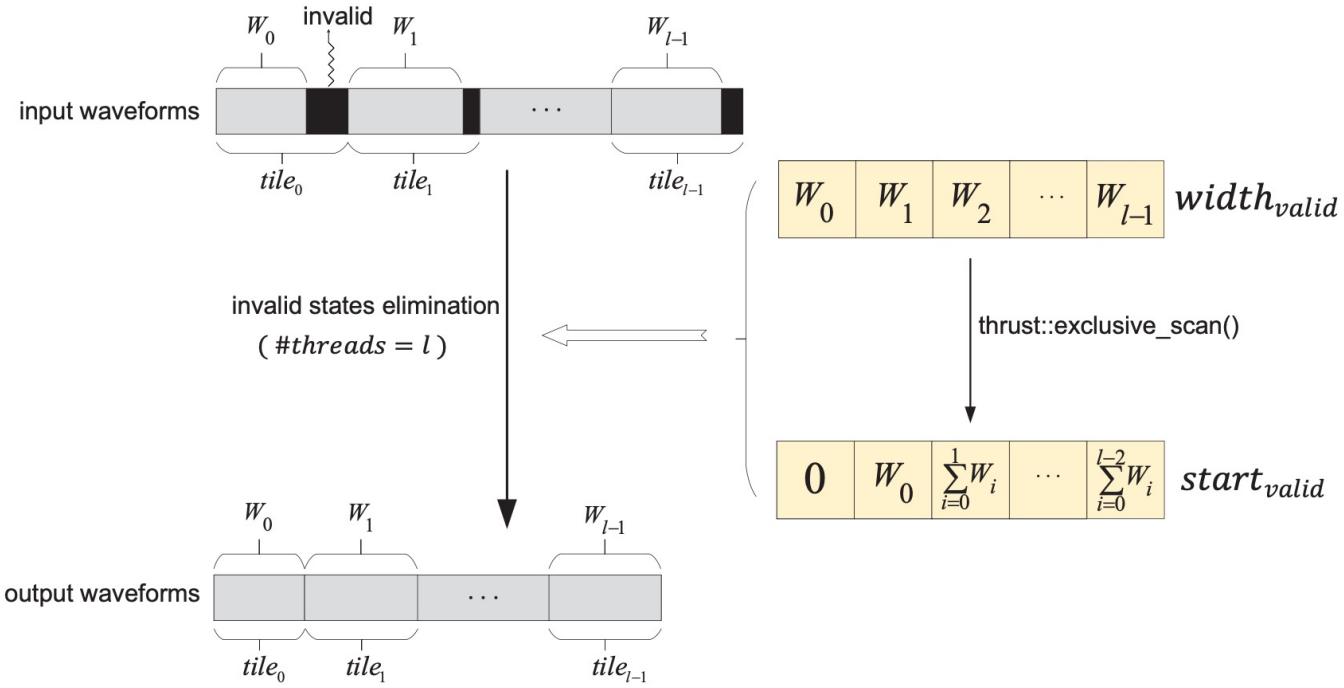
- Q(common): where is **std::vector** on GPU?
- A: Nowhere, because it's sequential.
  - But we can do it in parallel, very smartly, as follows:
- Suppose N threads generating new elements
  - (1) Estimate the number of new elements in advance: num[N]
  - (2) Compute the prefix sum of array num[N] -> start[N] (**Thrust**)
  - (3) Generate elements in parallel and use their own memory slots



Look-ahead level allocation
[Guannan Guo et al, DAC'21]

GPU-friendly algorithm tricks

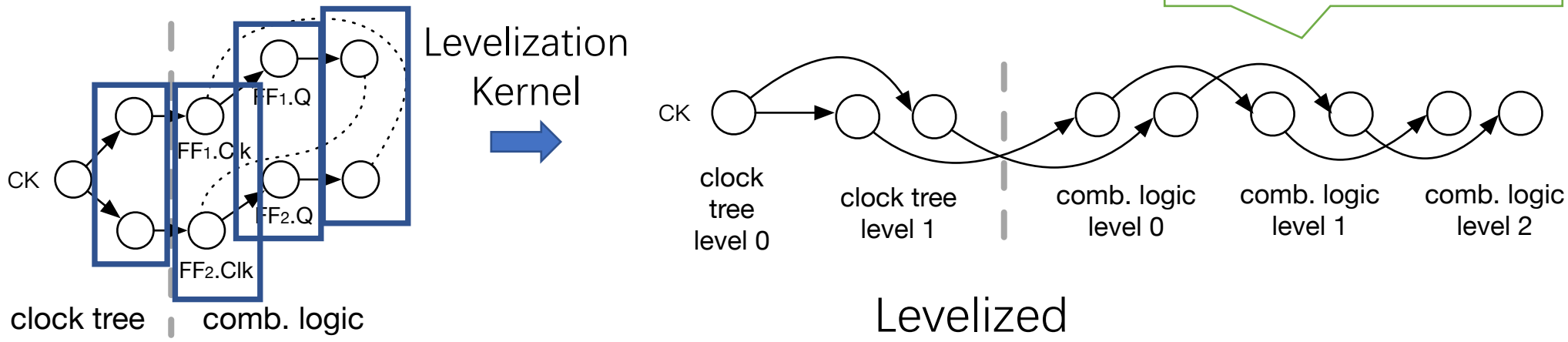
- Q+: what if estimating the number of new elements is hard?
- A: we only need an upper bound.
  - If some memory space is wasted, we can recycle them in parallel:



Courtesy Cheng Zheng et al ICCAD'21

GPU-friendly algorithm tricks

- Q: How to parallelize graph algorithms (like BFS, DFS, topological sorting, dynamic programming..)?
- A: try to **levelize** the graph first
  - (using my algorithm ☺) [ICCAD'20, DAC'21]



Graph Representation on CPU

Levelized Representation

Debug a GPU program

- cuda-gdb
- Compile with nvcc -g -G (would turn off all optimization)

```
(cuda-gdb) b muladd
Breakpoint 2 at 0x55555555c16b: file muladd.cu, line 20.
(cuda-gdb) c
Continuing.
[Detaching after fork from child process 24943]
[New Thread 0x7fffef8e5700 (LWP 24962)]
[New Thread 0x7fffef0e4700 (LWP 24963)]
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (0,0,0), device
 0, sm 0, warp 0, lane 0]

Thread 1 "muladd_cuda" hit Breakpoint 2, muladd<<<(262144,1,1),(256,1,1)>>> (
    size=67108864, a=0x7fffa6000000, b=0x7fff96000000, c=0x7fff86000000)
    at muladd.cu:21
21        int i = blockIdx.x * blockDim.x + threadIdx.x;
(cuda-gdb)
```

```
(cuda-gdb) cuda block 1 thread 3
[Switching focus to CUDA kernel 0, grid 1, block (1,0,0), thread (3,0,0), device
 0, sm 0, warp 9, lane 3]
21        int i = blockIdx.x * blockDim.x + threadIdx.x;
(cuda-gdb) n
22        if(i >= size) return;
(cuda-gdb)
23        a[i] += b[i] * c[i];
(cuda-gdb) p i
$12 = 259
(cuda-gdb)
```

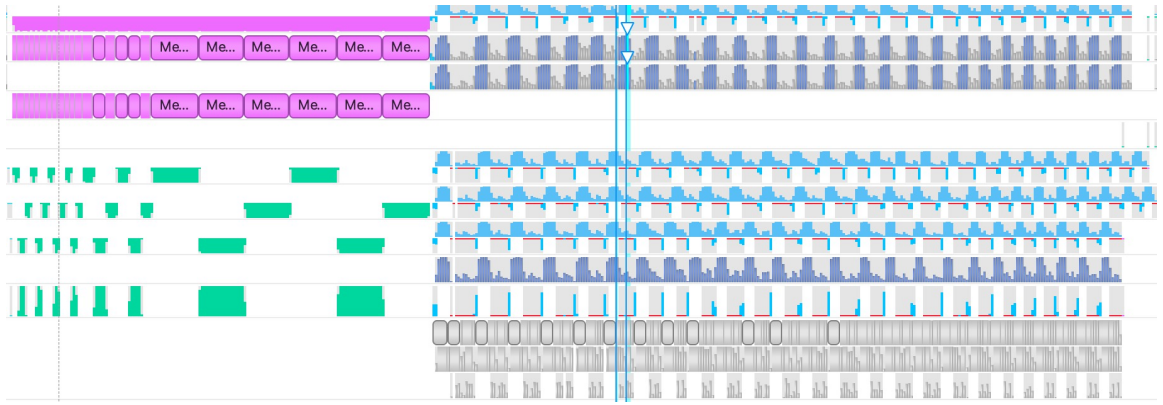
Profile a GPU program

- Nvidia **nsight** system/nvprof
- nsys profile -o <output> <command>
- Real-case demonstration

NVIDIA® Nsight™ Systems 是一个系统级性能分析工具，专用于实现应用算法的可视化，以帮助您发现诸多优化机会，以及进行调优以便跨任意数量或大小的 CPU 和 GPU（从大型服务器到较小的 SoC）进行高效扩展。

化，可帮助用户调查瓶颈，避免推断误报，并以更高的性能提升概率实现优化。用户将能够识别问题，例如 GPU 闲置、不必要的 GPU 同步、CPU 并行化不足，甚至其目标平台的 CPU 和 GPU 中意外昂贵的算法。它旨在跨各种 NVIDIA 平台进行扩展，例如：大型 Tesla 多

```
(base) zizhengguo@ceca2080x4:~/CPPR/code-gpu/build$ nsys profile -o combo6.100 .
/timer-gpu_shell ../benchmarks/Combo6v2/Combo6v2.* 100
Warning: LBR backtrace method is not supported on this platform. DWARF backtrace
 method will be used.
Collecting data...
[2022-01-11 13:57:20.202] [main] [info] [/home/zizhengguo/CPPR/code-gpu/timer.cp
```



Thanks

Questions are welcome

Zizheng Guo, gzz@pku.edu.cn