UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

HOW NOT TO

PROGRAM YOUR GPUS.

BASED ON MY DISSERTATION WORK “**LOAD BALANCING ON THE GPU**”

Muhammad Osama [mosama@ucdavis.edu]

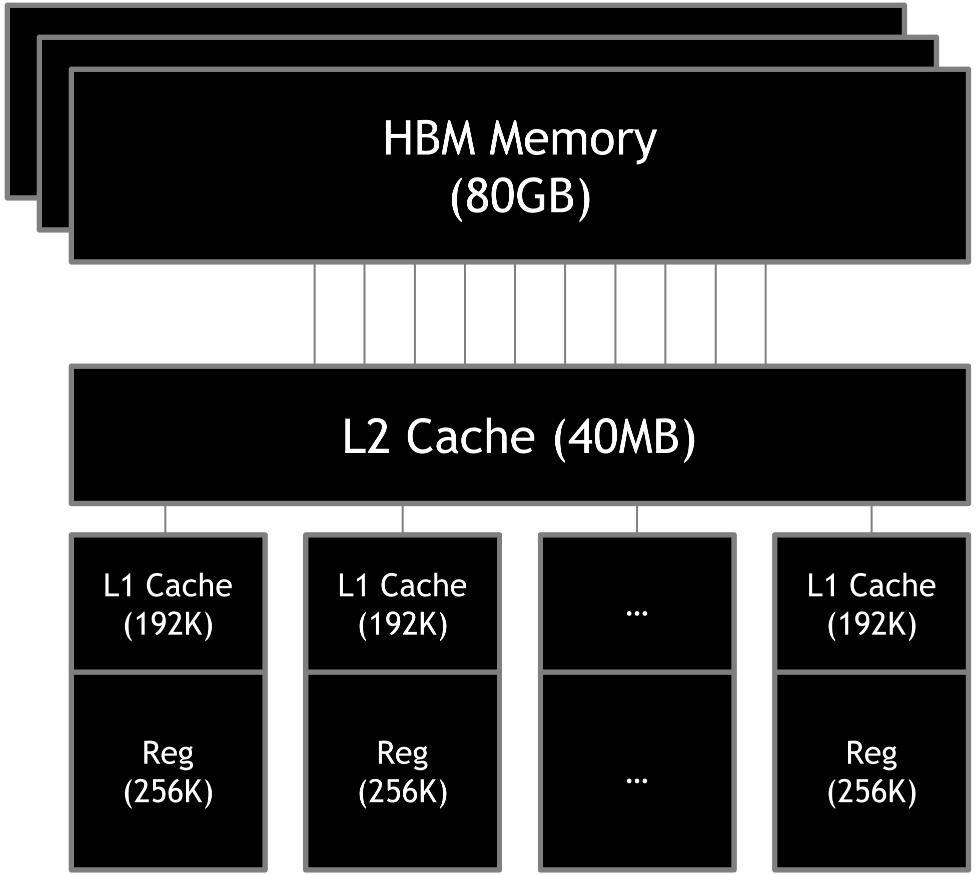
THE 3 PERSPECTIVES

- Programmability,
- Performance, and
- … everything else

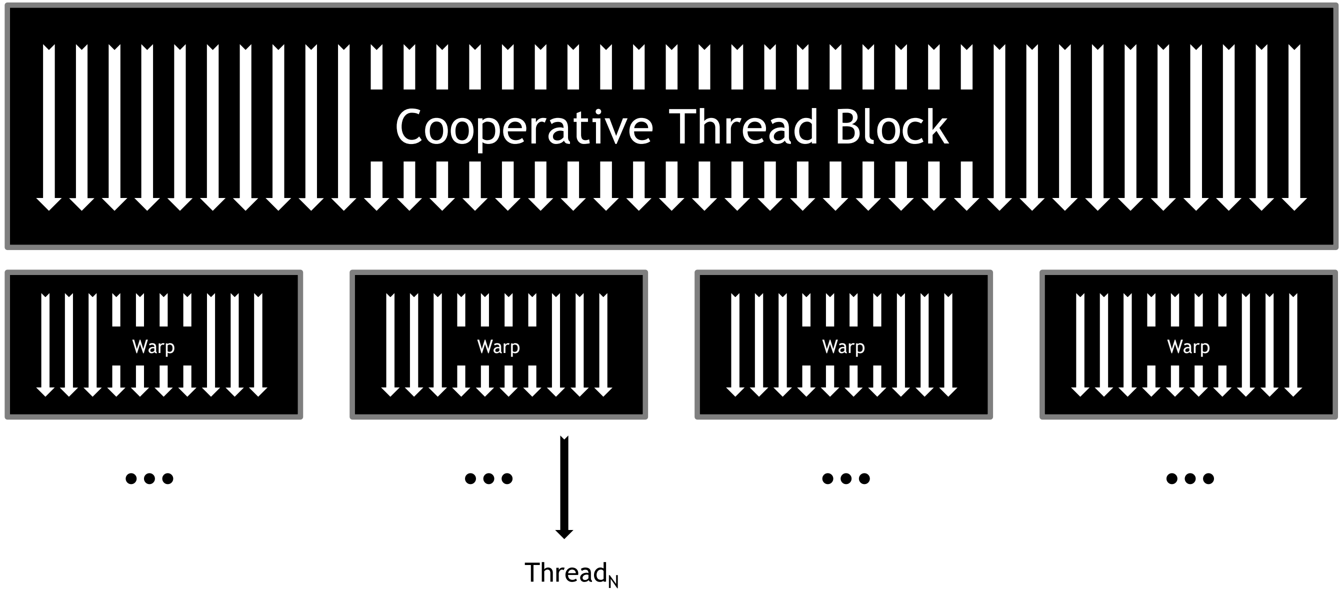
THE 3 PERSPECTIVES

- Programmability,
- Performance, and
- … everything else (Software Engineering)

MEMORY SYSTEM



COMPUTE HIERARCHY




```
int i = blockIdx.x * blockDim.x + threadIdx.x;
...
y[i] = a * x[i] + y[i];
```


UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

OCCUPANCY

Threads

Registers

Shared Memory

CONCURRENCY

“WHY CUDA PROGRAMMING IS THE WAY IT IS?”

Stephen Jones, CUDA Architect (NVIDIA)

It’s all physics.

Memory system is *everything*.

Occupancy is the next big thing.

Concurrency is saving grace.

… and oversubscription is how we achieve it.

What does that look like?

CUDA 101.

```
constexpr int N = 1<<20;
float* x; float* y;
cudaMalloc(&x, N * sizeof(float));
cudaMalloc(&y, N * sizeof(float));
cudaMemset(x, 1.0f, N * sizeof(float));
cudaMemset(y, 2.0f, N * sizeof(float));

constexpr int block_size = 128;
int grid_size = (N - block_size + 1) / block_size;
saxpy<<<grid_size, block_size>>>(N, 2.0f, x, y);

cudaFree(x); cudaFree(y);
```

```
constexpr int N = 1<<20;
float* x; float* y;
cudaMalloc(&x, N * sizeof(float));
cudaMalloc(&y, N * sizeof(float));
cudaMemset(x, 1.0f, N * sizeof(float));
cudaMemset(y, 2.0f, N * sizeof(float));

constexpr int block_size = 128;
int grid_size = (N – block_size + 1) / block_size;
saxpy<<<grid_size, block_size>>>(N, 2.0f, x, y);

cudaFree(x); cudaFree(y);
```

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N)
    y[i] = a * x[i] + y[i];
}
```

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N)
    y[i] = a * x[i] + y[i];
}
```

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N)
    y[i] = a * x[i] + y[i];
}
```

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N)
    y[i] = a * x[i] + y[i];
}
```

*This is **great!***

This is ***great!***

No. It's not.

PROBLEMS WITH THIS APPROACH.

Things crash when the `grid stride` exceeds the device limit.

Errors/Typos in calculating the `grid stride` (for example, ceil.)

There is no threads reuse, things are being created and destroyed.

You cannot debug by making the problem simpler: 1 thread, 1 block.

Readability (sequential saxpy is not written like the kernel code.)

https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/

```
constexpr int N = 1<<20;
float* x; float* y;
cudaMalloc(&x, N * sizeof(float));
cudaMalloc(&y, N * sizeof(float));
cudaMemset(x, 1.0f, N * sizeof(float));
cudaMemset(y, 2.0f, N * sizeof(float));

constexpr int block_size = 128;
int grid_size = (N - block_size + 1) / block_size;
saxpy<<<grid_size, block_size>>>(N, 2.0f, x, y);

cudaFree(x); cudaFree(y);
```

Exceeds device limit.

```
constexpr int N = 1<<20;
float* x; float* y;
cudaMalloc(&x, N * sizeof(float));
cudaMalloc(&y, N * sizeof(float));
cudaMemset(x, 1.0f, N * sizeof(float));
cudaMemset(y, 2.0f, N * sizeof(float));

constexpr int block_size = 128;
int grid_size = (N - block_size + 1) / block_size;
saxpy<<<grid_size, block_size>>>(N, 2.0f, x, y);

cudaFree(x); cudaFree(y);
```

Improper division.

```
constexpr int N = 1<<20;
float* x; float* y;
cudaMalloc(&x, N * sizeof(float));
cudaMalloc(&y, N * sizeof(float));
cudaMemset(x, 1.0f, N * sizeof(float));
cudaMemset(y, 2.0f, N * sizeof(float));

constexpr int block_size = 128;
int grid_size = ceil_div(N, block_size);
saxpy<<<grid_size, block_size>>>(N, 2.0f, x, y);

cudaFree(x); cudaFree(y);
```

Improper division.

```
constexpr int N = 1<<20;
float* x; float* y;
cudaMalloc(&x, N * sizeof(float));
cudaMalloc(&y, N * sizeof(float));
cudaMemset(x, 1.0f, N * sizeof(float));
cudaMemset(y, 2.0f, N * sizeof(float));

constexpr int block_size = 128;
int grid_size = ceil_div(N, block_size);
saxpy<<<grid_size, block_size>>>(N, 2.0f, x, y);
```

Correctness dependent on grid size.

```
cudaFree(x); cudaFree(y);
```

Correctness dependent on grid size.

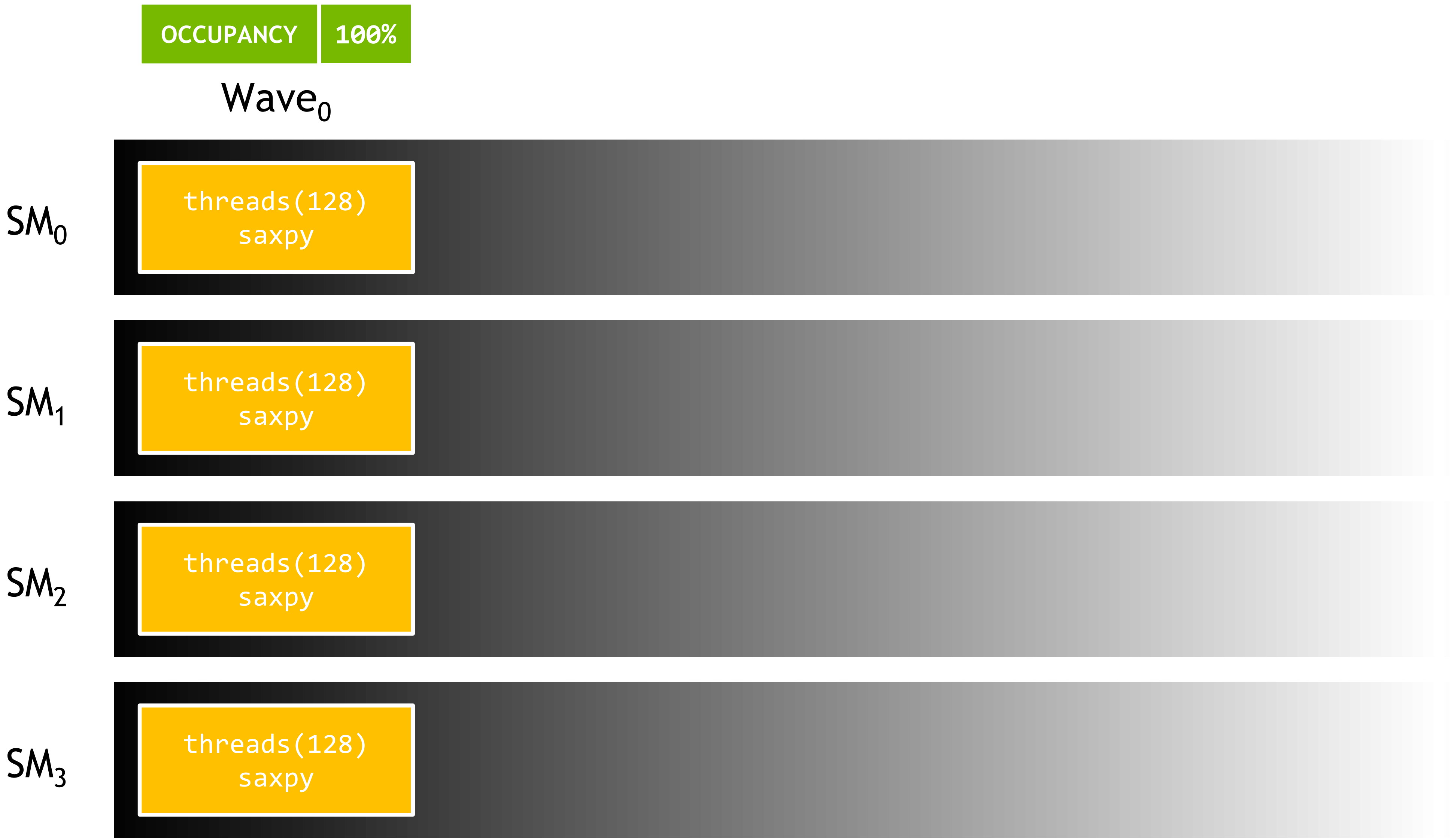
```
constexpr int N = 1<<20;
float* x; float* y;
cudaMalloc(&x, N * sizeof(float));
cudaMalloc(&y, N * sizeof(float));
cudaMemset(x, 1.0f, N * sizeof(float));
cudaMemset(y, 2.0f, N * sizeof(float));

saxpy<<<1, 1>>>(N, 2.0f, x, y);

cudaFree(x); cudaFree(y);
```

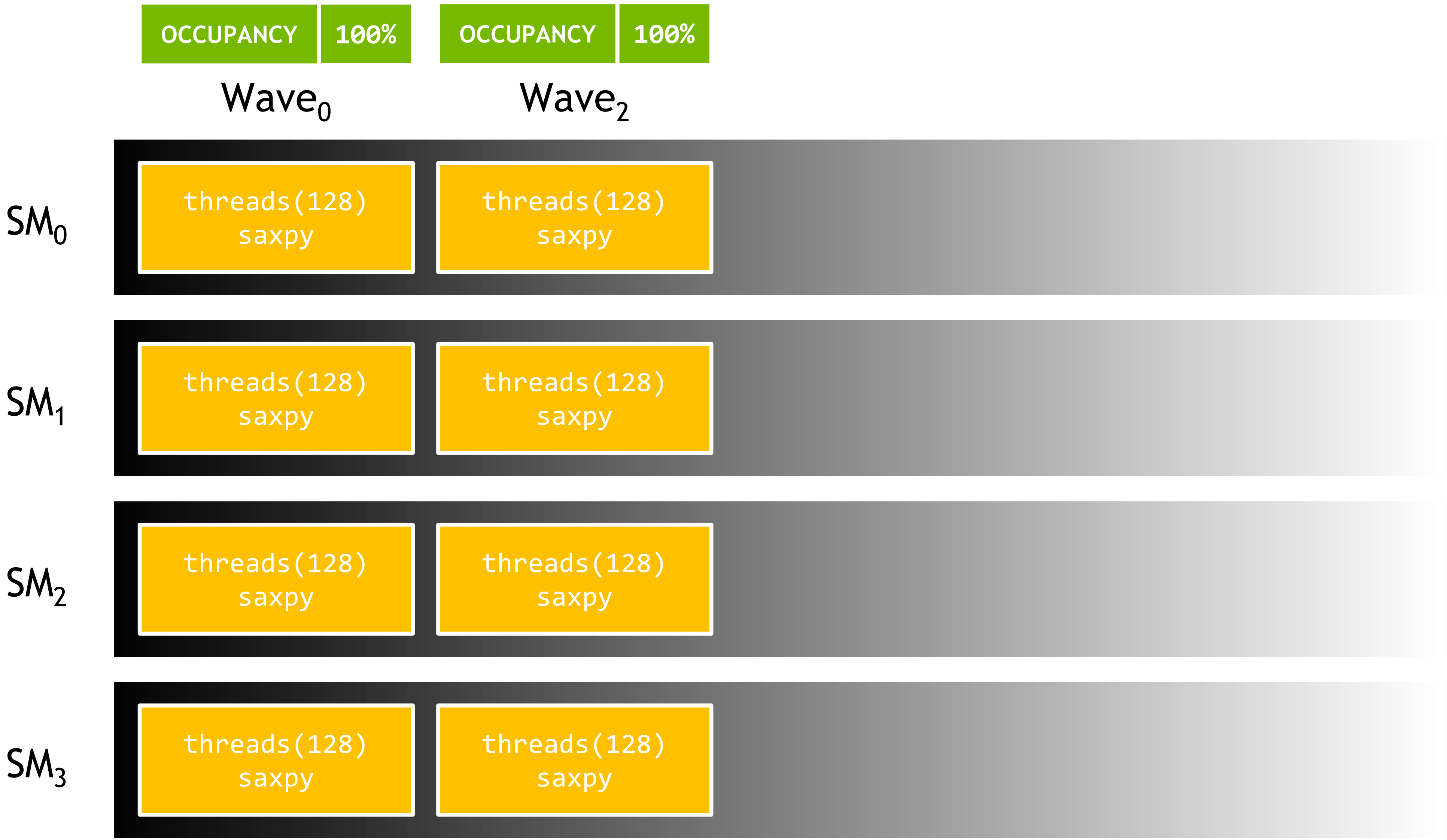
WHAT THE GPU SEES.

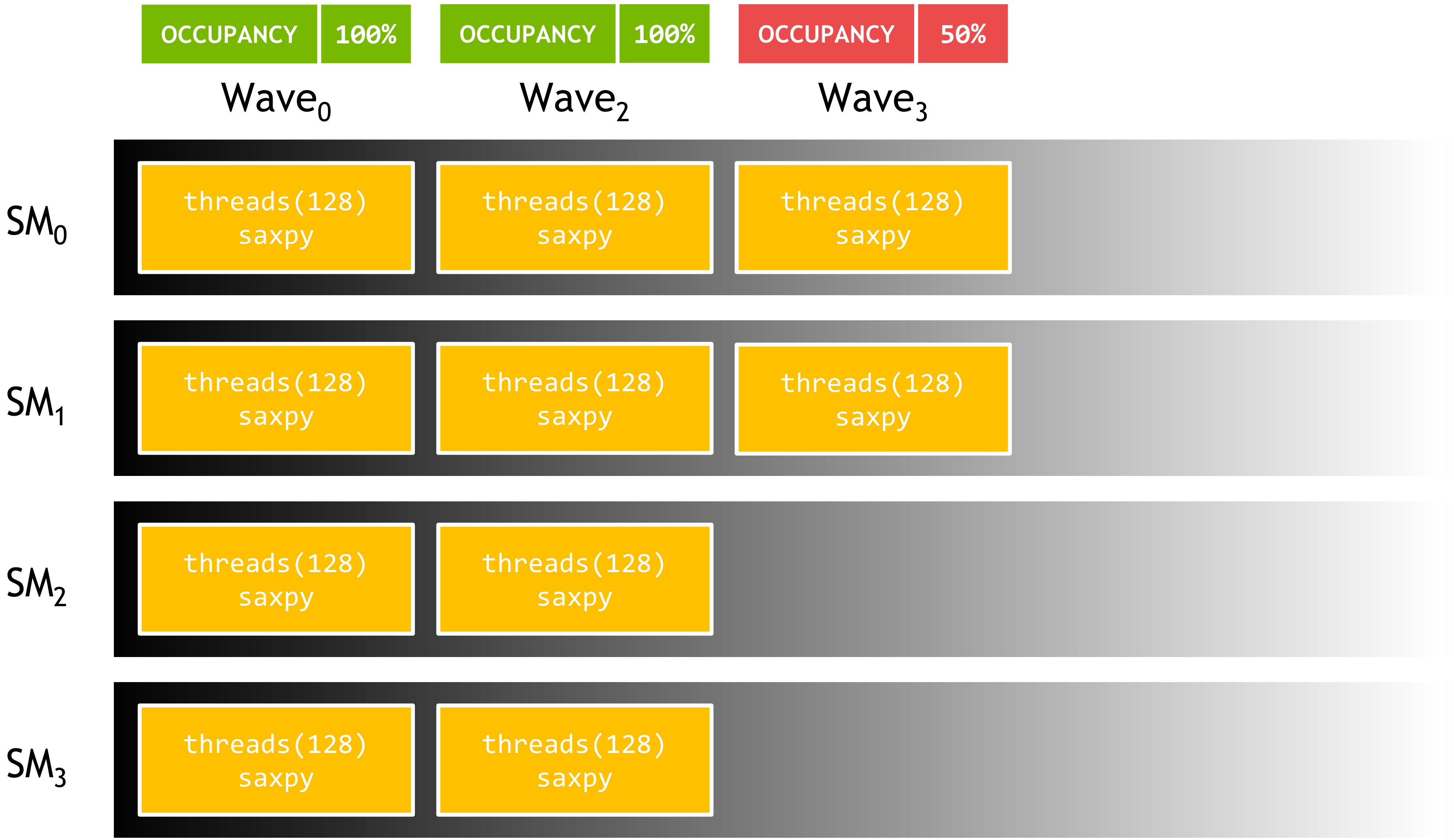
e.g.

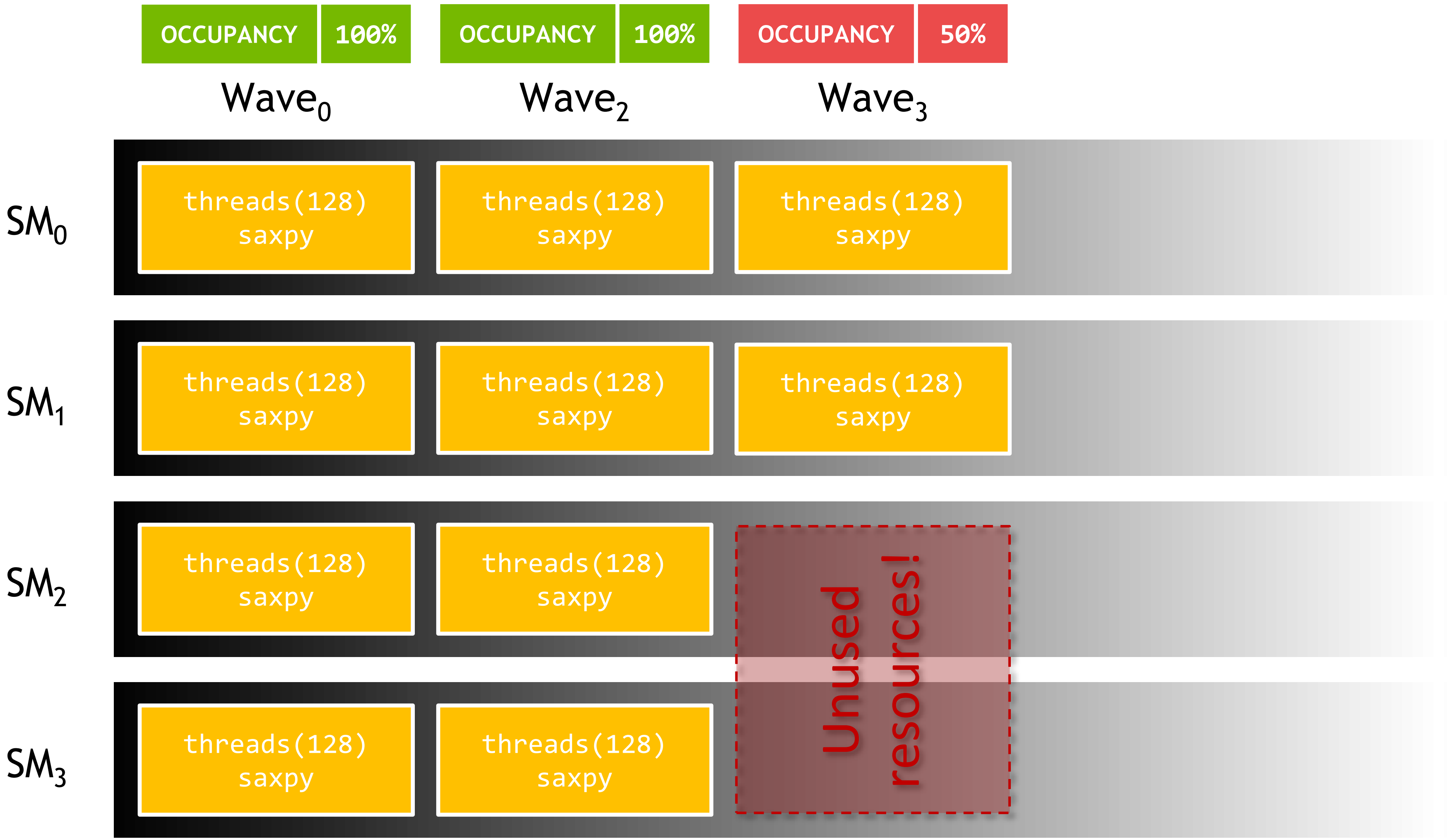
N = 4096 elements

Number of SMs = 4 (hypothetical GPU)


OCCUPANCY
100%
Wave0
SM0
threads(128)
saxpy
SM1
threads(128)
saxpy
SM2
threads(128)
saxpy
SM3
threads(128)
saxpy


OCCUPANCY
100%
OCCUPANCY
100%
Wave$_0$
Wave$_2$
SM$_0$
SM$_1$
SM$_2$
SM$_3$
threads(128)
saxpy
threads(128)
saxpy
threads(128)
saxpy
threads(128)
saxpy
threads(128)
saxpy
threads(128)
saxpy
threads(128)
saxpy
threads(128)
saxpy


OCCUPANCY
100%
OCCUPANCY
100%
OCCUPANCY
50%
Wave0
Wave2
Wave3
SM0
SM1
SM2
SM3
threads(128)
saxpy
threads(128)
saxpy
threads(128)
saxpy
threads(128)
saxpy
threads(128)
saxpy
threads(128)
saxpy
threads(128)
saxpy
threads(128)
saxpy
threads(128)
saxpy
threads(128)
saxpy


OCCUPANCY 100%
OCCUPANCY 100%
OCCUPANCY 50%
Wave$_0$
Wave$_2$
Wave$_3$
SM$_0$
SM$_1$
SM$_2$
SM$_3$
threads(128)
saxpy
Unused resources!

CUDA 201.

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N)
    y[i] = a * x[i] + y[i];
}
```

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
       i < N;
       i += blockDim.x * gridDim.x) {
    y[i] = a * x[i] + y[i];
  }
}
```

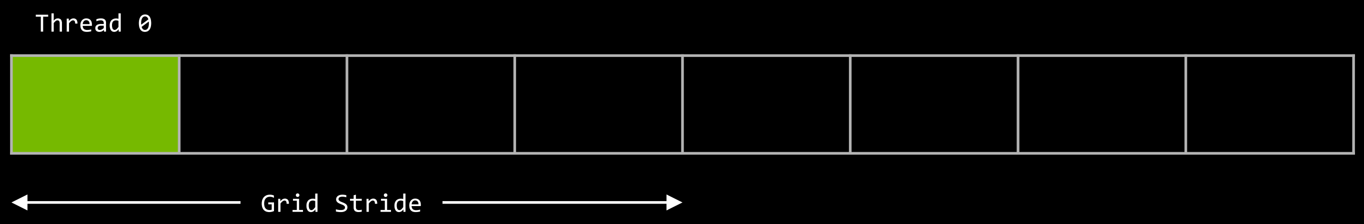
```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
       i < N;
       i += blockDim.x * gridDim.x) {
    y[i] = a * x[i] + y[i];
  }
}
```

THE “GRID STRIDE” LOOP

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
       i < N;
       i += blockDim.x * gridDim.x) {
    y[i] = a * x[i] + y[i];
  }
}
```

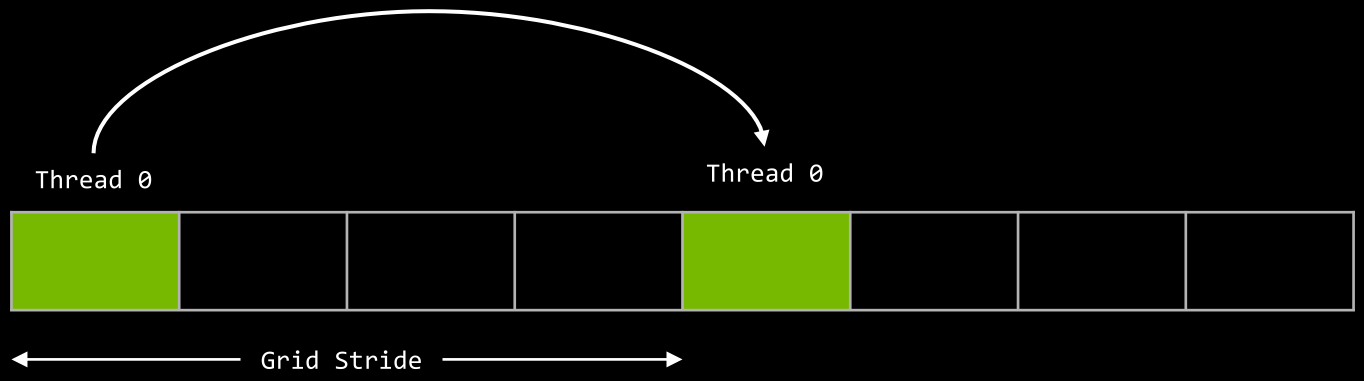
THE "GRID STRIDE" LOOP

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
       i < N;
       i += blockDim.x * gridDim.x) {
    y[i] = a * x[i] + y[i];
  }
}
```



THE “GRID STRIDE” LOOP

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
       i < N;
       i += blockDim.x * gridDim.x) {
    y[i] = a * x[i] + y[i];
  }
}
```



“DATA PARALLEL” MODE

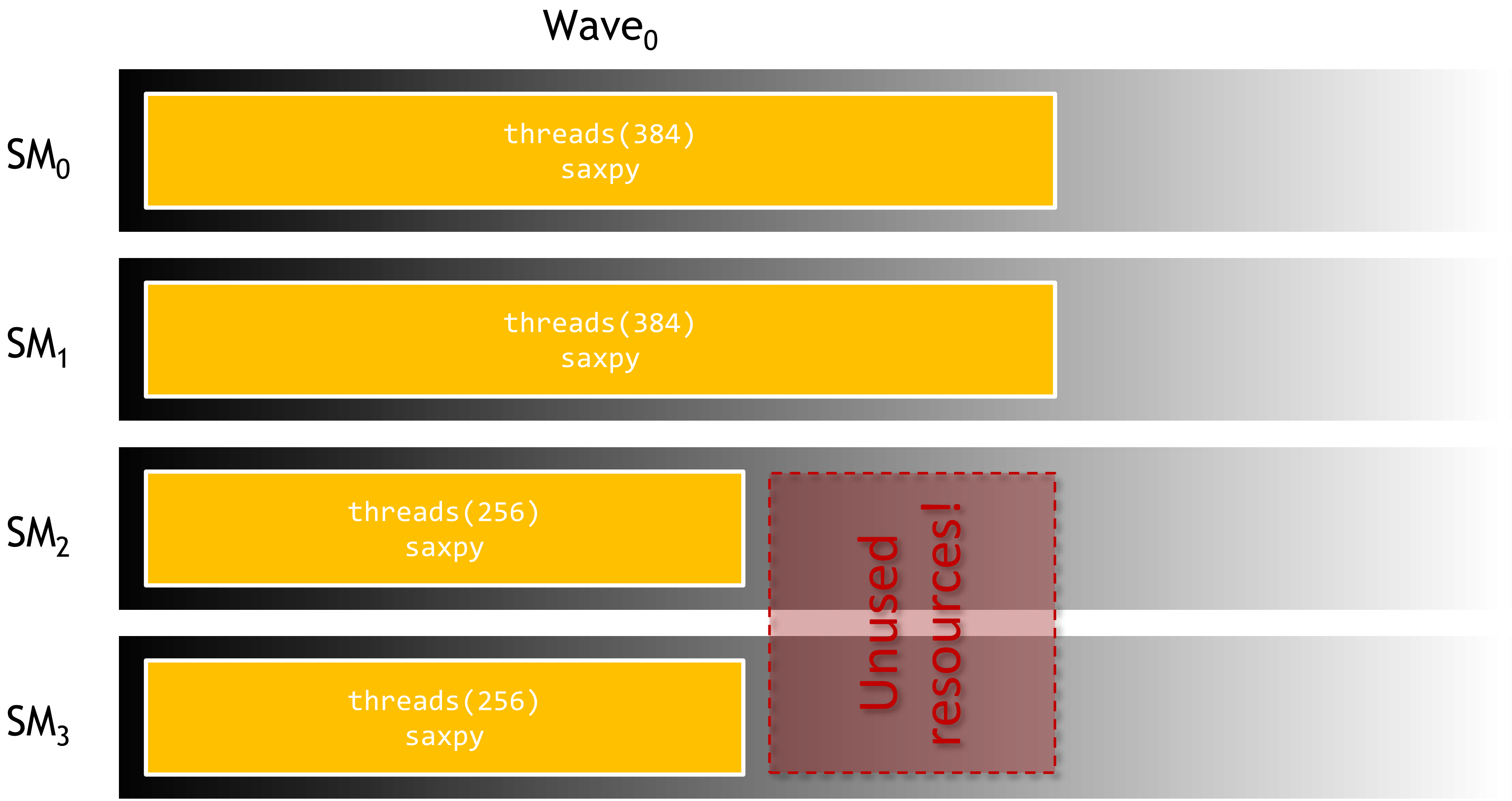
```
...

constexpr int block_size = 128;
int grid_size = ceil_div(N, block_size);
saxpy<<<grid_size, block_size>>>(N, 2.0f, x, y);

...
```

"DEBUG" MODE

```
...

constexpr int block_size = 128;
int grid_size = 1;
saxpy<<<grid_size, block_size>>>(N, 2.0f, x, y);

...
```

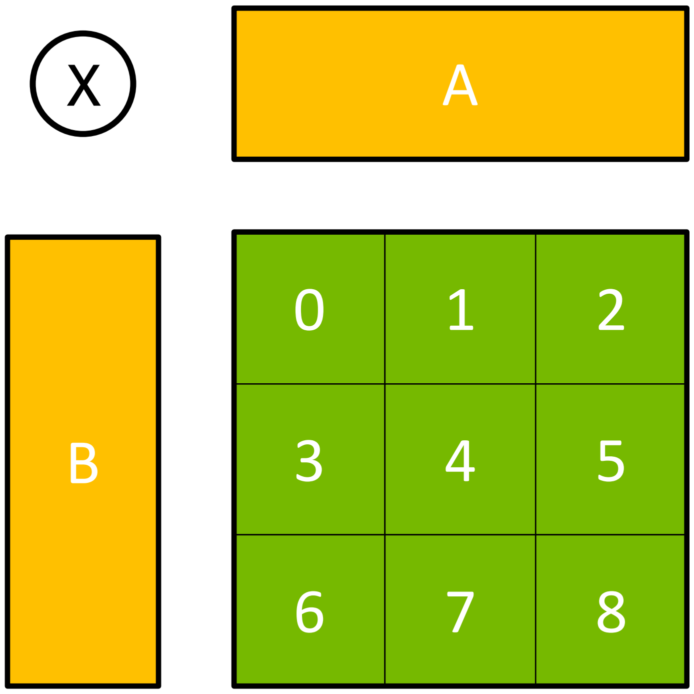
"..." MODE

```
...

constexpr int block_size = 128;
int grid_size = 2 * SMs;
saxpy<<<grid_size, block_size>>>(N, 2.0f, x, y);

...
```


Wave$_0$
SM$_0$
threads(384)
saxpy
SM$_1$
threads(384)
saxpy
SM$_2$
threads(256)
saxpy
SM$_3$
threads(256)
saxpy
Unused resources!

LET'S ILLUSTRATE USING A REAL PROBLEM.

General Matrix-Multiplication (GEMM)



$$C = A \times B$$

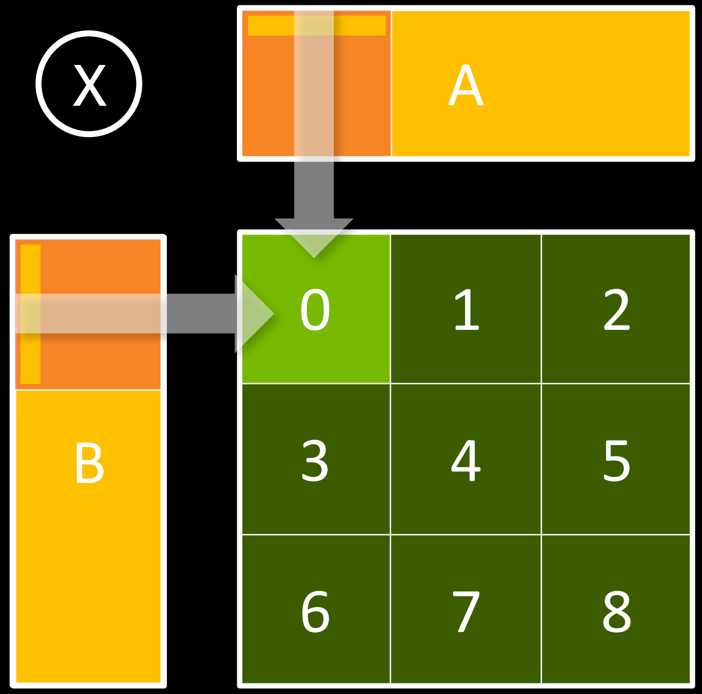
Fine-grained parallelism

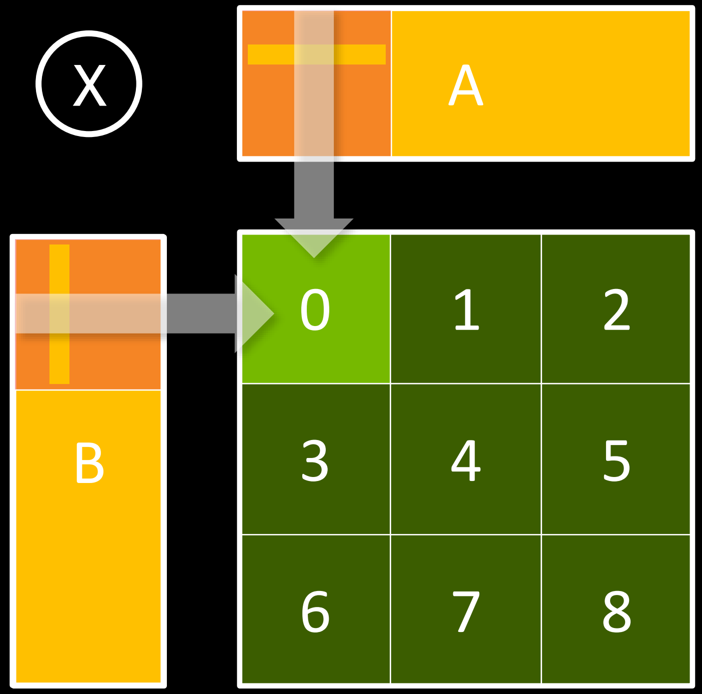
Regular workload

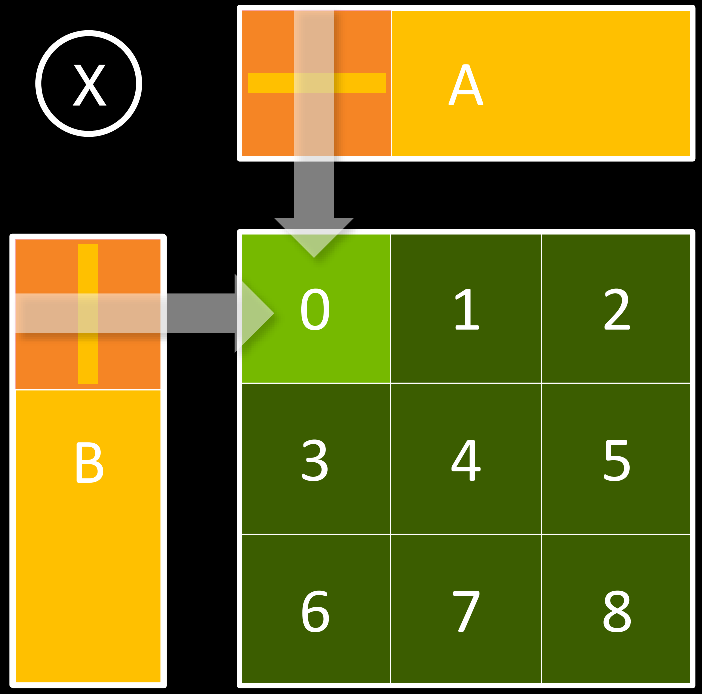
Perfect for our memory systems

… but, how do we program it?

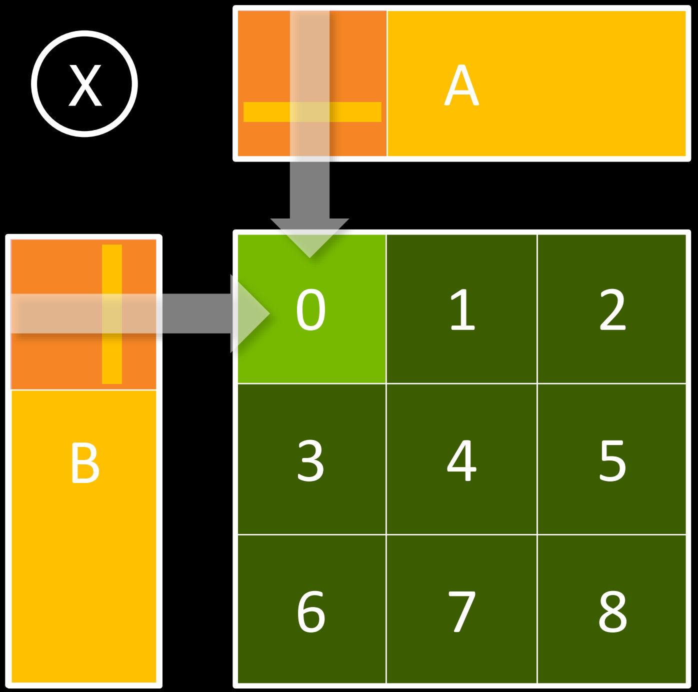
```
accumulator[BLK_M, BLK_K]
iters_per_tile = ⌈k/BLK_K⌉;

// One output tile per CTA.
fork CTA[x] in [⌈m/BLK_M⌉ x ⌈n/BLK_N⌉]
  // Perform MAC iterations for this tile.
  accumulator = mac_loop(x, 0, iters_per_tile);
  // Store accumulators to output tile.
  store_tile(C, x, accumulator);
join
```


X
A
B
0
1
2
3
4
5
6
7
8

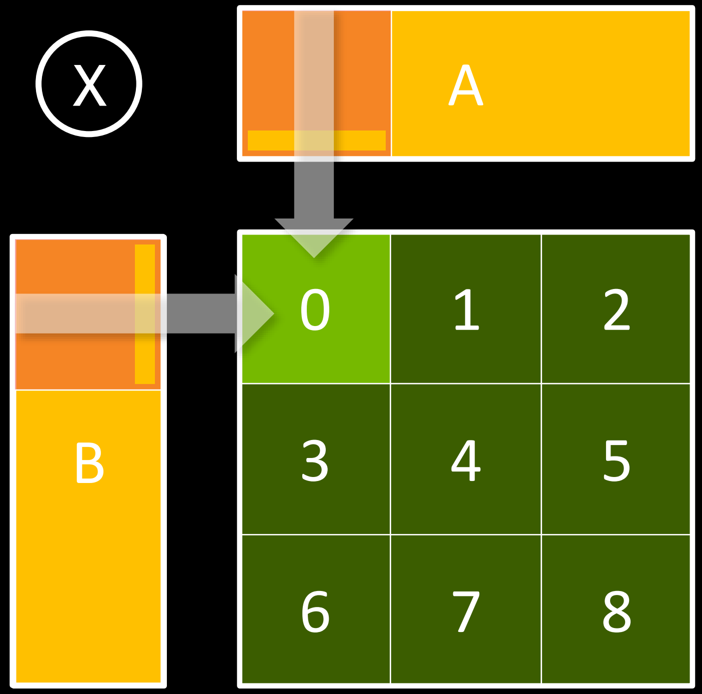

X
A
B
0
1
2
3
4
5
6
7
8



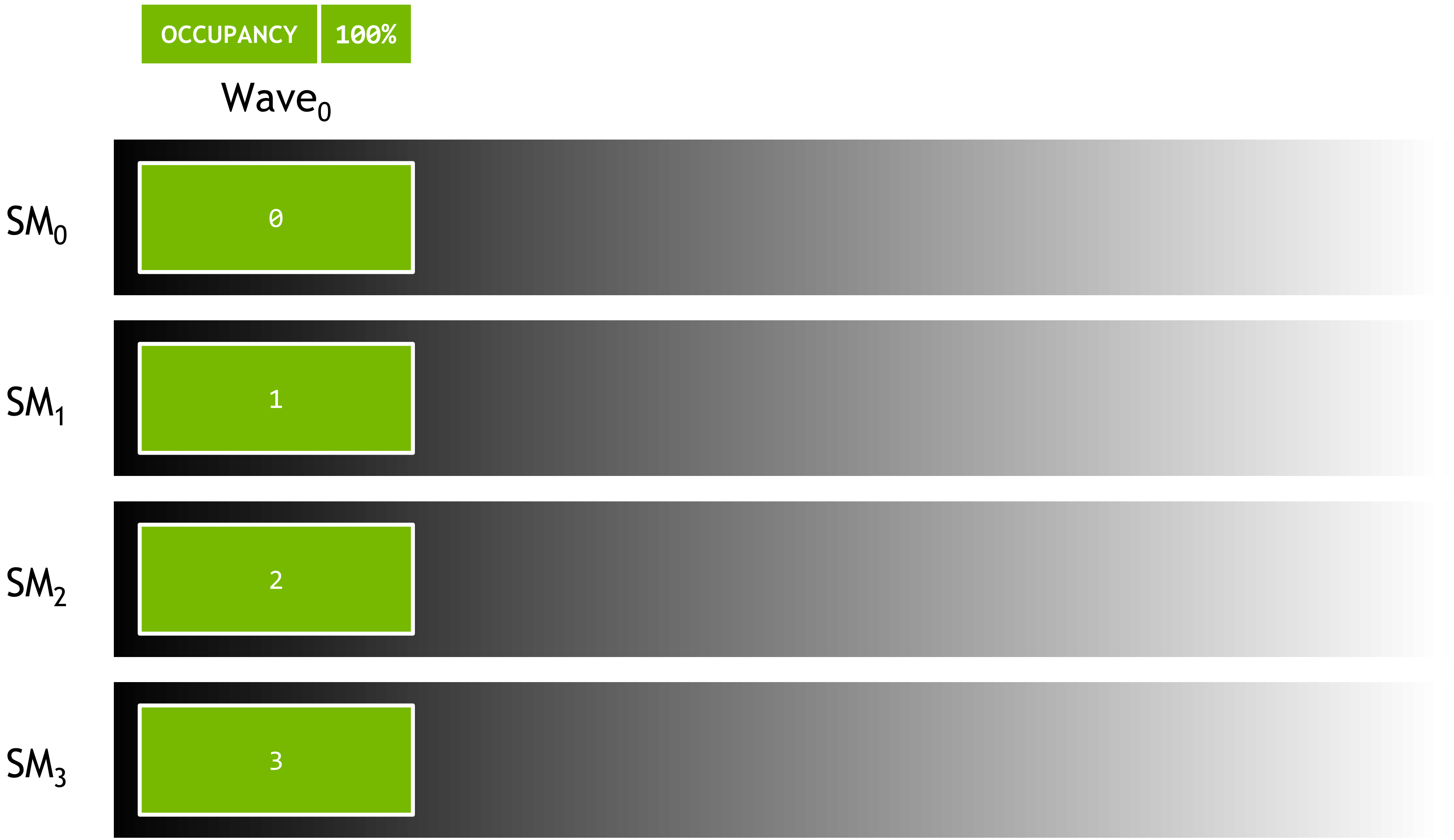
X
A
B
0
1
2
3
4
5
6
7
8

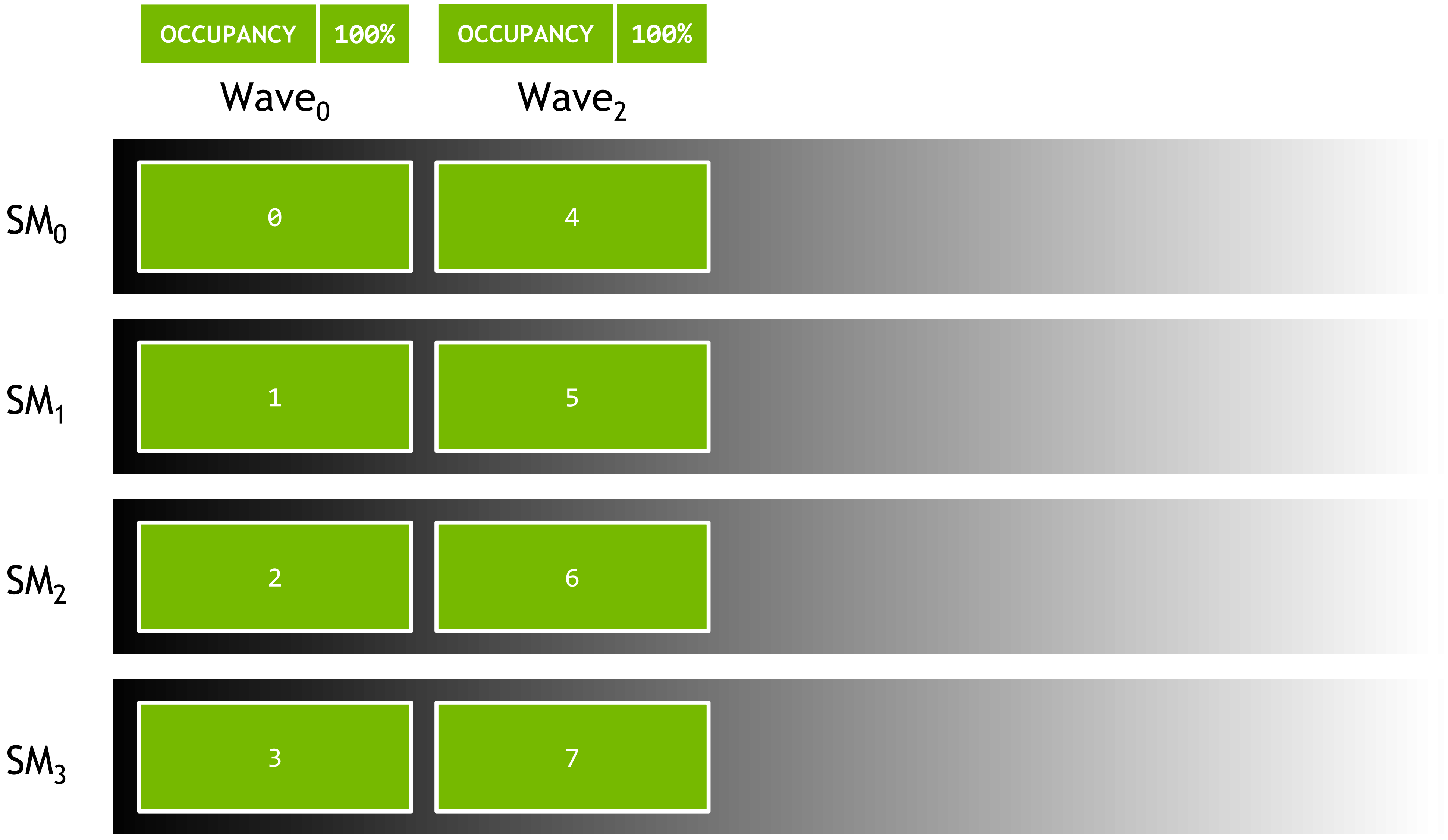


X
A
B
0
1
2
3
4
5
6
7
8

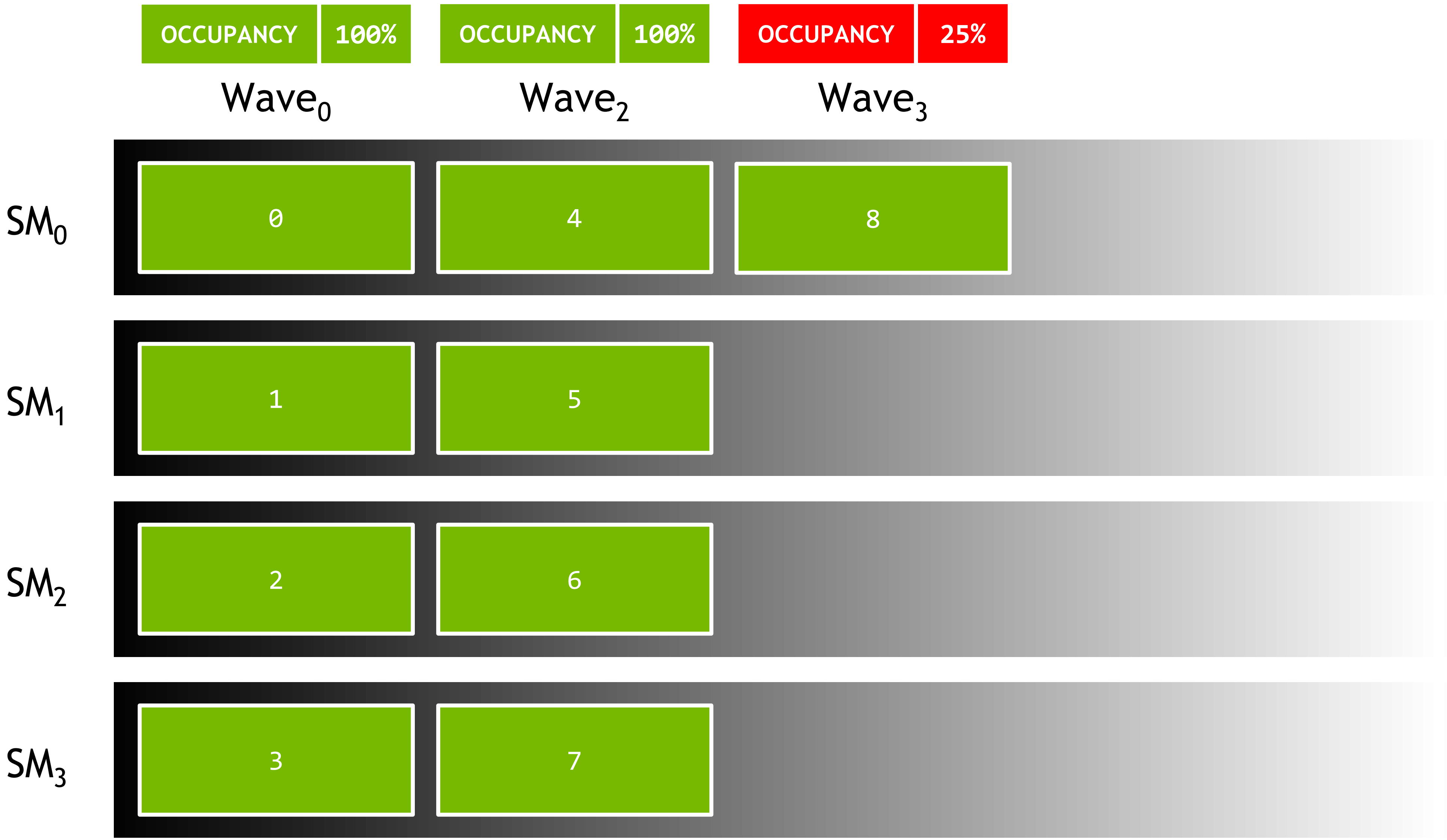


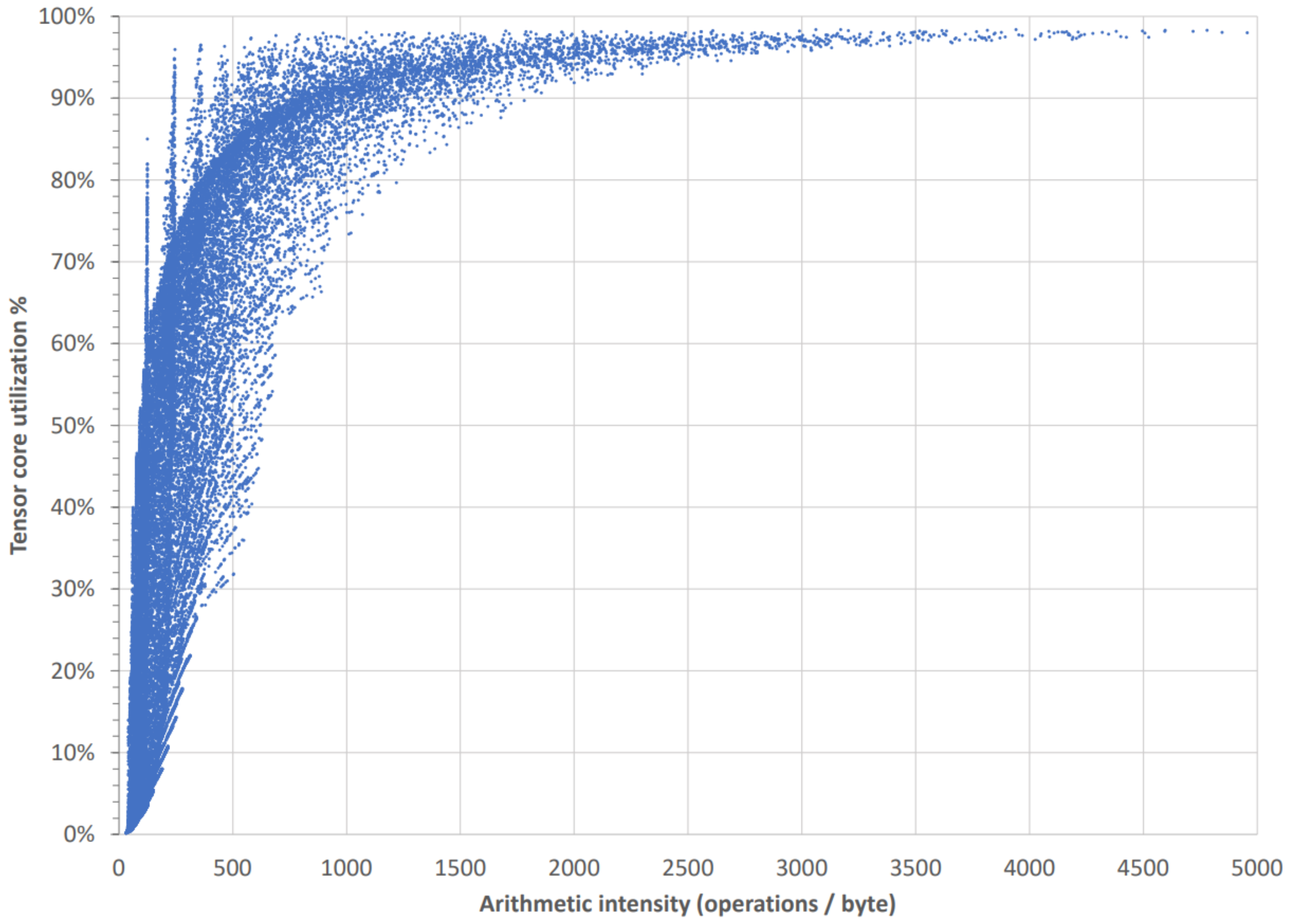
X
A
B
0
1
2
3
4
5
6
7
8


OCCUPANCY
100%
Wave$_0$
SM$_0$
0
SM$_1$
1
SM$_2$
2
SM$_3$
3

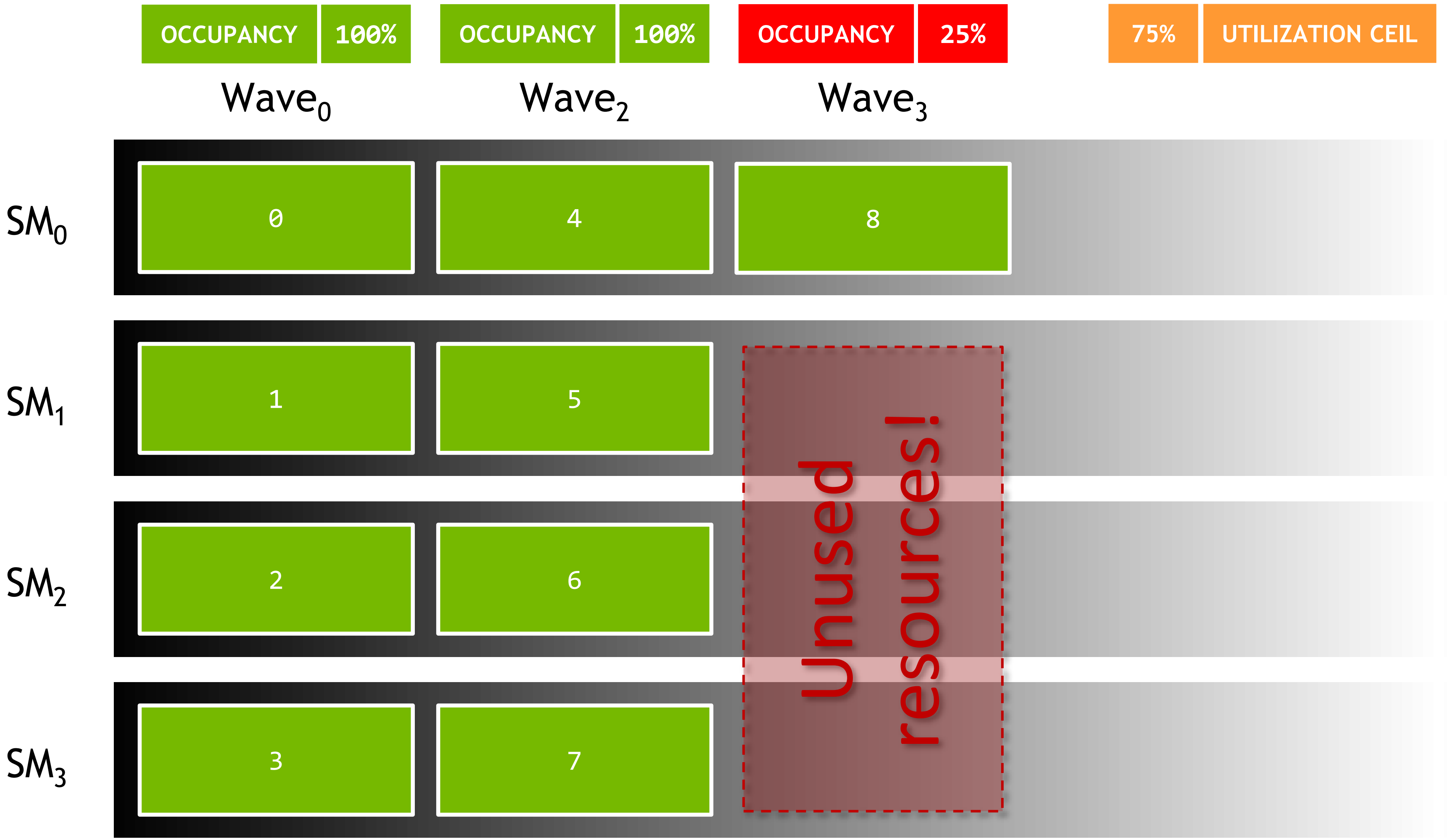


OCCUPANCY
100%
OCCUPANCY
100%
Wave0
Wave2
SM0
SM1
SM2
SM3
0
4
1
5
2
6
3
7

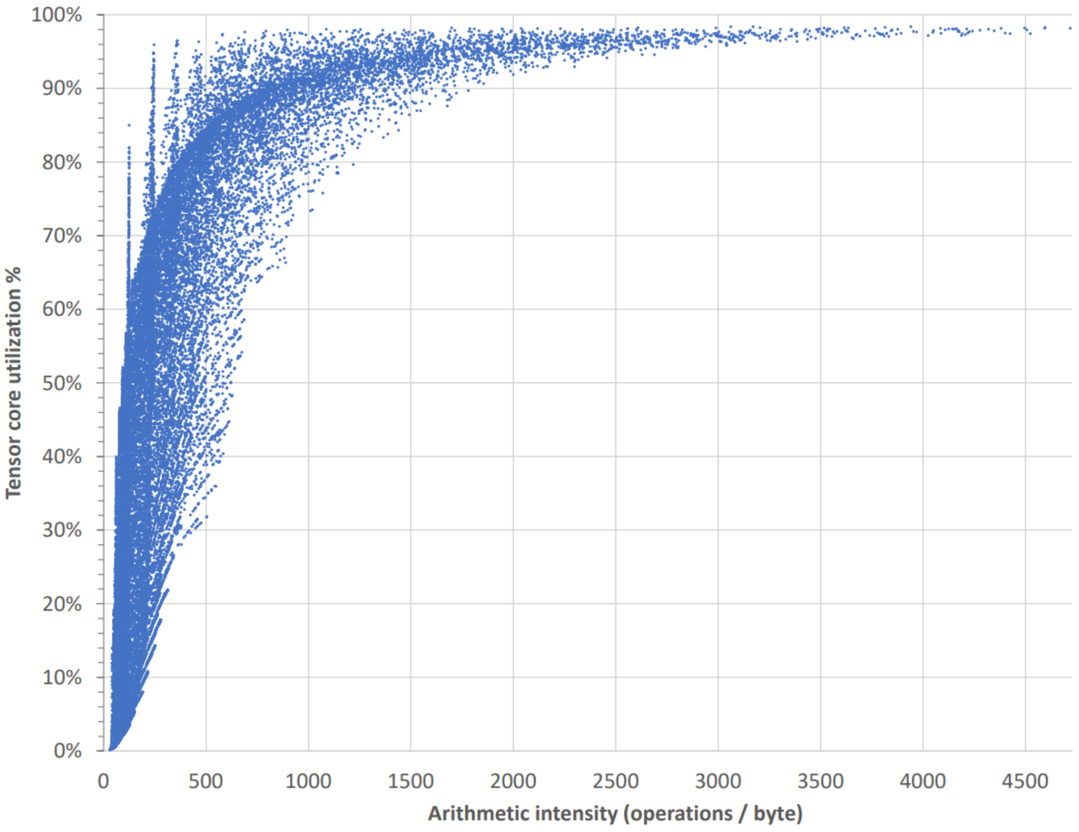

OCCUPANCY
100%
OCCUPANCY
100%
OCCUPANCY
25%
Wave0
Wave2
Wave3
SM0
0
4
8
SM1
1
5
SM2
2
6
SM3
3
7

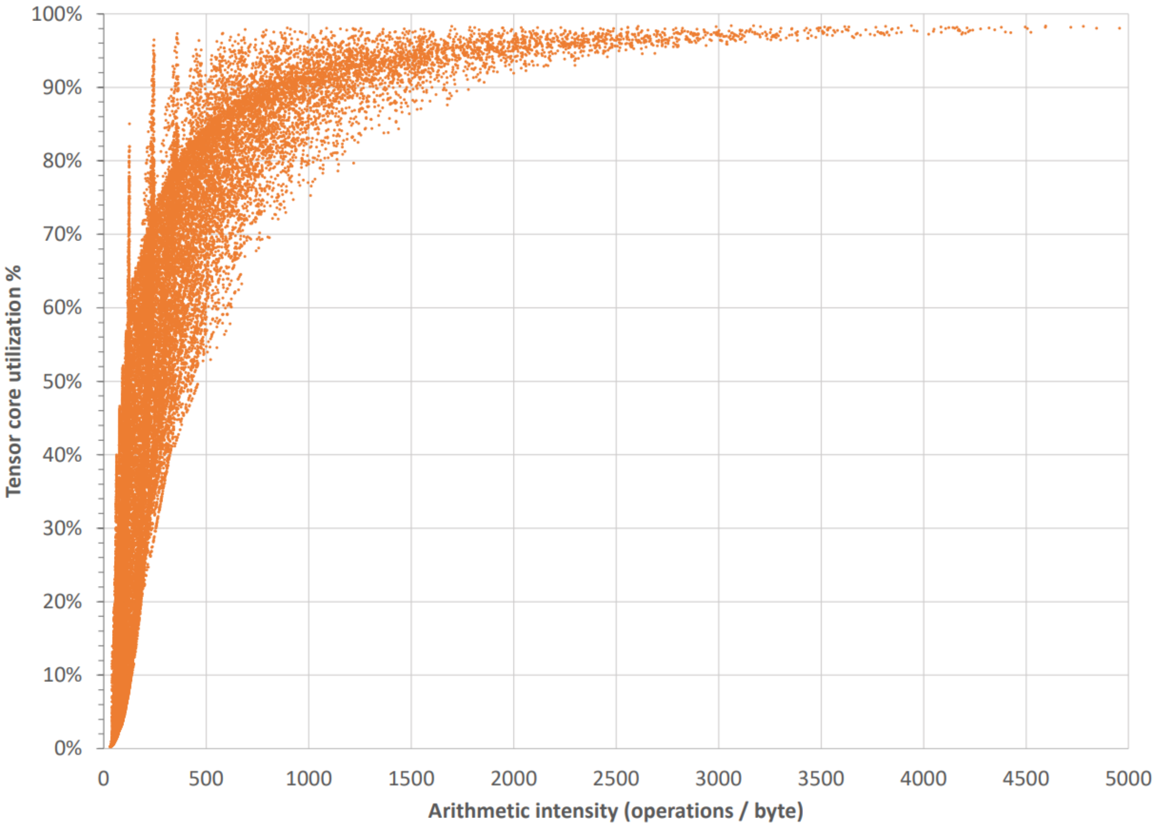


DATA-PARALLEL HGEMM.

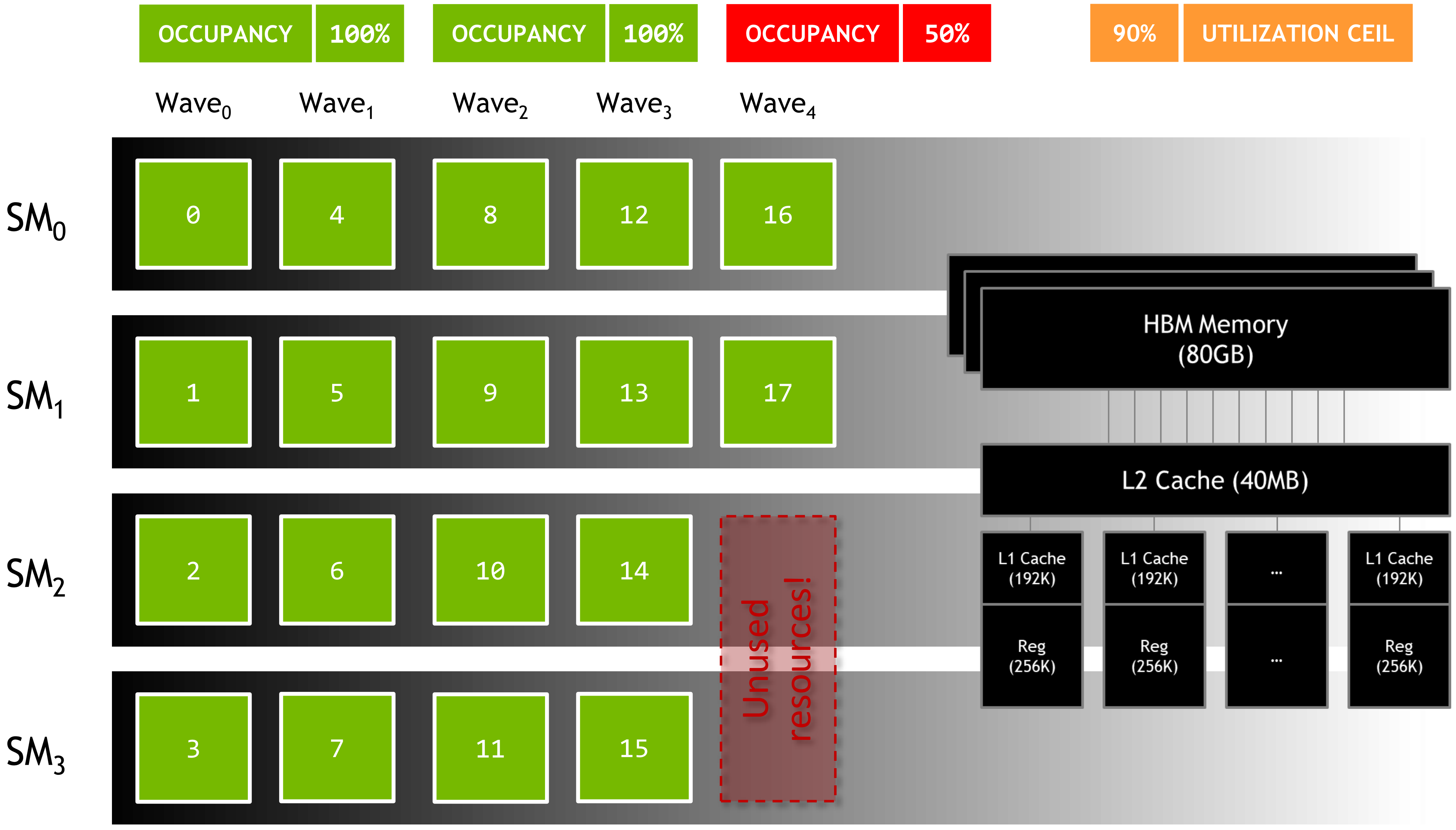

OCCUPANCY 100%
OCCUPANCY 100%
OCCUPANCY 25%
75% UTILIZATION CEIL
Wave0
Wave2
Wave3
SM0
SM1
SM2
SM3
0
4
8
1
5
2
6
3
7
Unused resources!

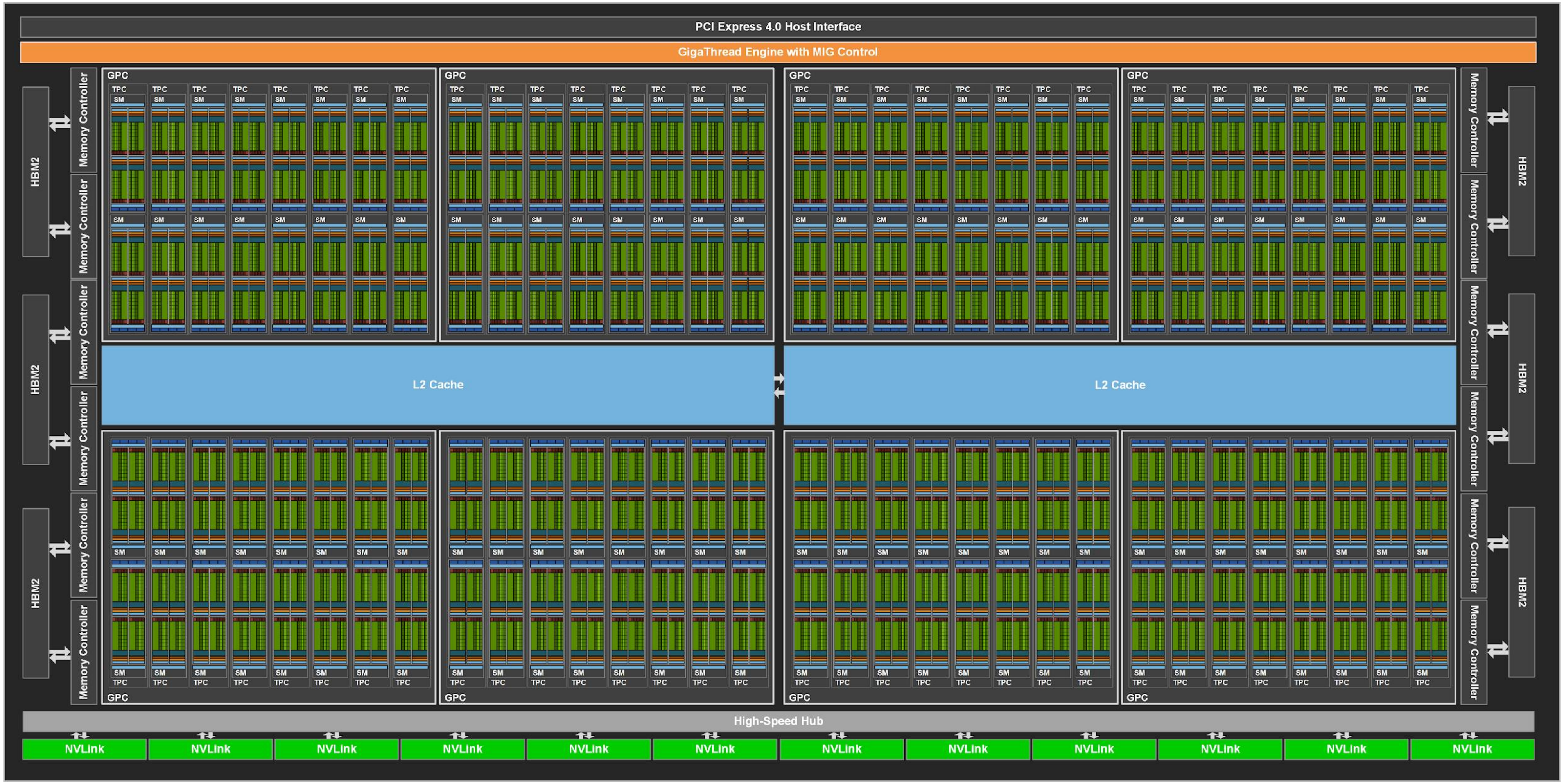

OCCUPANCY
100%
OCCUPANCY
100%
OCCUPANCY
50%
90%
UTILIZATION CEIL
$Wave_0$
$Wave_1$
$Wave_2$
$Wave_3$
$Wave_4$
SM_0
0
4
8
12
16
SM_1
1
5
9
13
17
SM_2
2
6
10
14
SM_3
3
7
11
15
Unused resources!





DATA-PARALLEL HGEMM W/ MORE TILE SIZES.


OCCUPANCY 100%
OCCUPANCY 100%
OCCUPANCY 50%
90% UTILIZATION CEIL
$Wave_0$
$Wave_1$
$Wave_2$
$Wave_3$
$Wave_4$
SM_0
SM_1
SM_2
SM_3
0
4
8
12
16
1
5
9
13
17
2
6
10
14
Unused resources!
3
7
11
15
HBM Memory (80GB)
L2 Cache (40MB)
L1 Cache (192K)
Reg (256K)
L1 Cache (192K)
Reg (256K)
...
...
L1 Cache (192K)
Reg (256K)



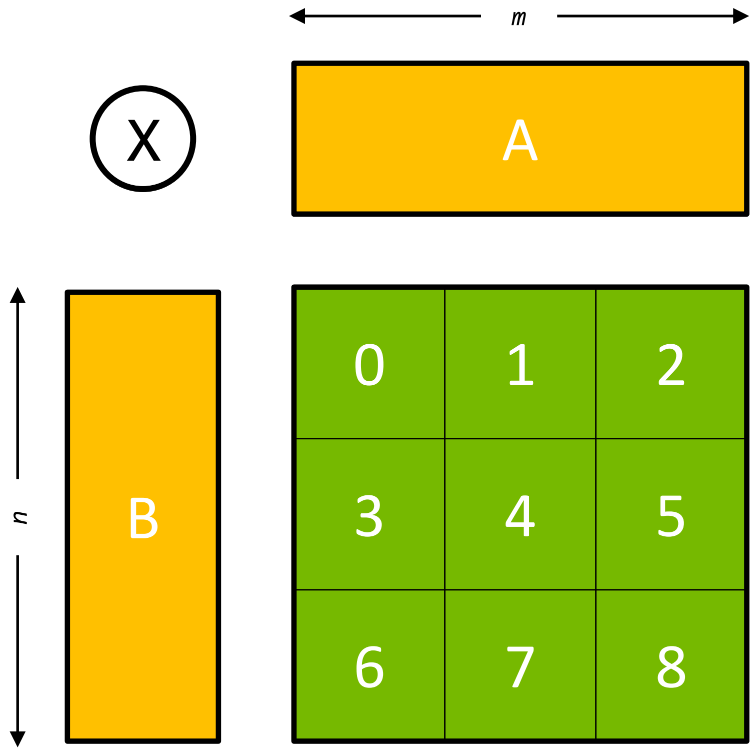
NVIDIA GA100 (128 SMs)

CUDA 201.5

"Work-centric approach" & Programmability

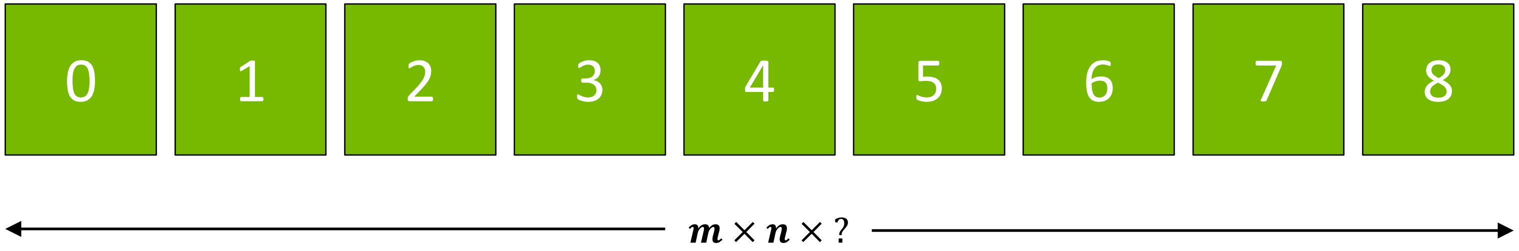
Consider scheduling as a first-class citizen when programming your GPUs.

THE WORK-CENTRIC GEMM.



Data-parallel approach, work scheduled: $m \times n$

THE WORK-CENTRIC GEMM.

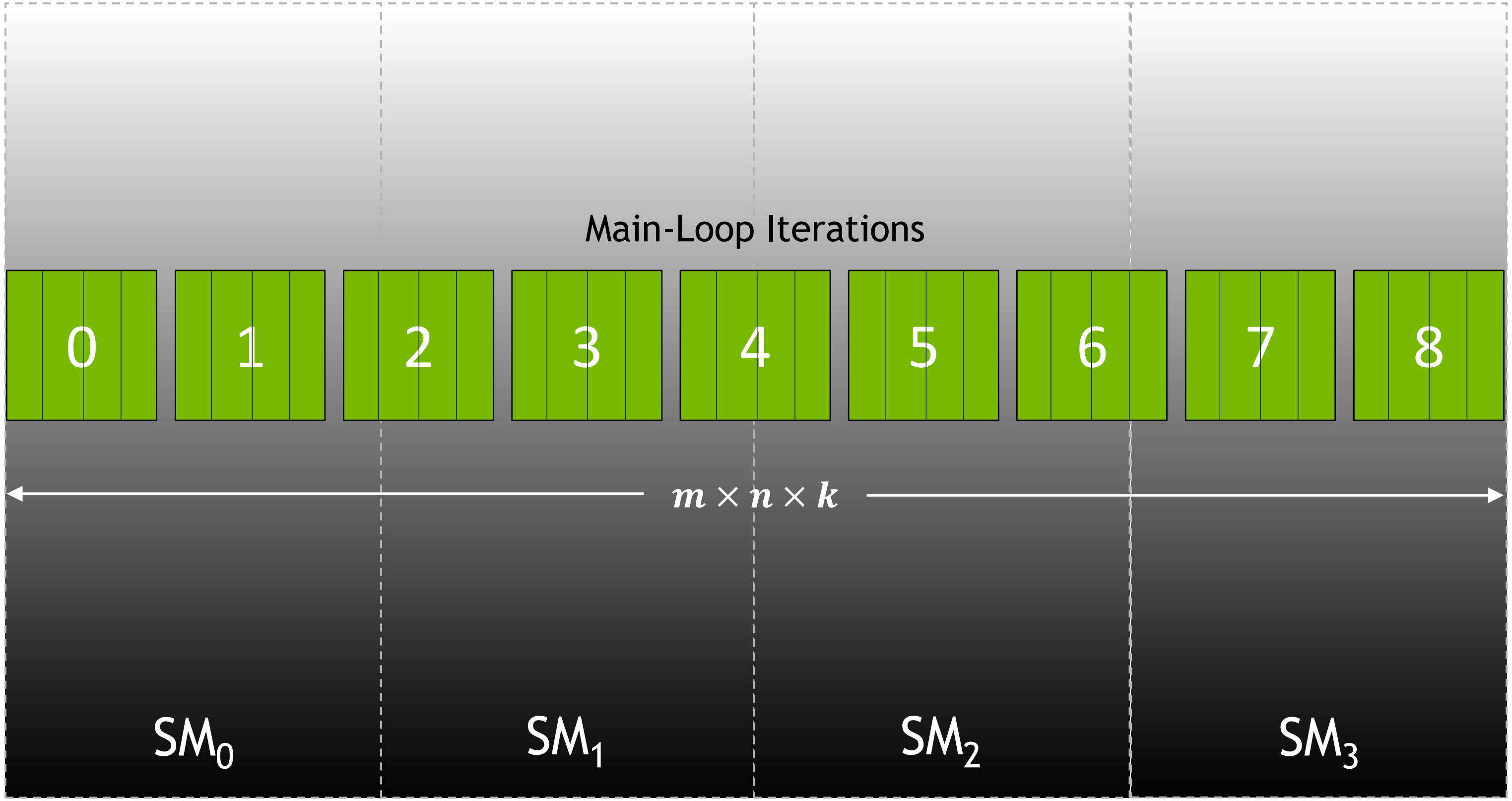


THE WORK-CENTRIC GEMM.

$\boldsymbol{m} \times \boldsymbol{n} \times \boldsymbol{k}$

Main-Loop Iterations

The Work-Centric GEMM.




OCCUPANCY
100%
Wave0
SM0
0
1
2
fixup
SM1
2
3
4
fixup
fixup
SM2
4
5
6
fixup
fixup
SM3
6
7
8
fixup

```
__global__ gemm(...) {
    // The Stream-K work-processing loop
    while (cta_itr < cta_last_itr) {
        ...
        // The standard tile-based GEMM main loop
        accum = gemm_tile(tile_id, cta_itr - tile_first_itr, tile_stop_itr - tile_first_itr);
        if (!started_tile && !finished_tile) {
            // Carry-out partial sums
        } else {
            if (started_tile && !finished_tile) {
               // Carry-in sets of partial sums
            }
            // Produce final output tile
            eplilogue(accum, tile_id);
        }
    }
}
```

```
__global__ gemm(...) {
    // The Stream-K work-processing loop
    while (cta_itr < cta_last_itr) {
        ...
        // The standard tile-based GEMM main loop
        accum = gemm_tile(tile_id, cta_itr - tile_first_itr, tile_stop_itr - tile_first_itr);
        if (!started_tile && !finished_tile) {
            // Carry-out partial sums
        } else {
            if (started_tile && !finished_tile) {
               // Carry-in sets of partial sums
            }
            // Produce final output tile
            eplilogue(accum, tile_id);
        }
    }
}
```

“DEBUG” MODE

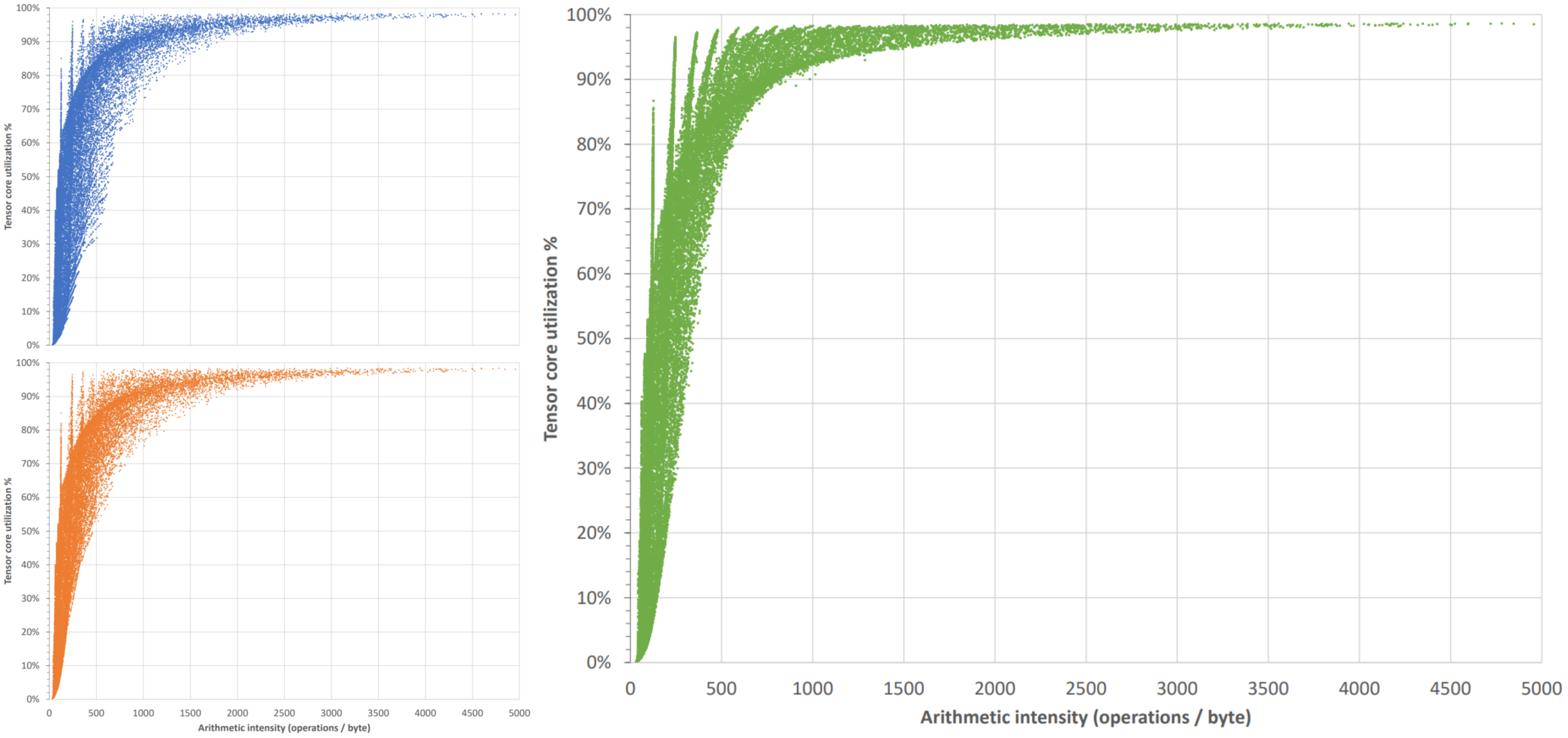
```
...

constexpr int block_size = 128;
int grid_size = 1;
gemm<<<grid_size, block_size>>>(...);

...
```

“100% UTILIZATION” MODE

```
...

constexpr int block_size = 128;
int grid_size = 2 * SMs;
gemm<<<grid_size, block_size>>>(...);

...
```

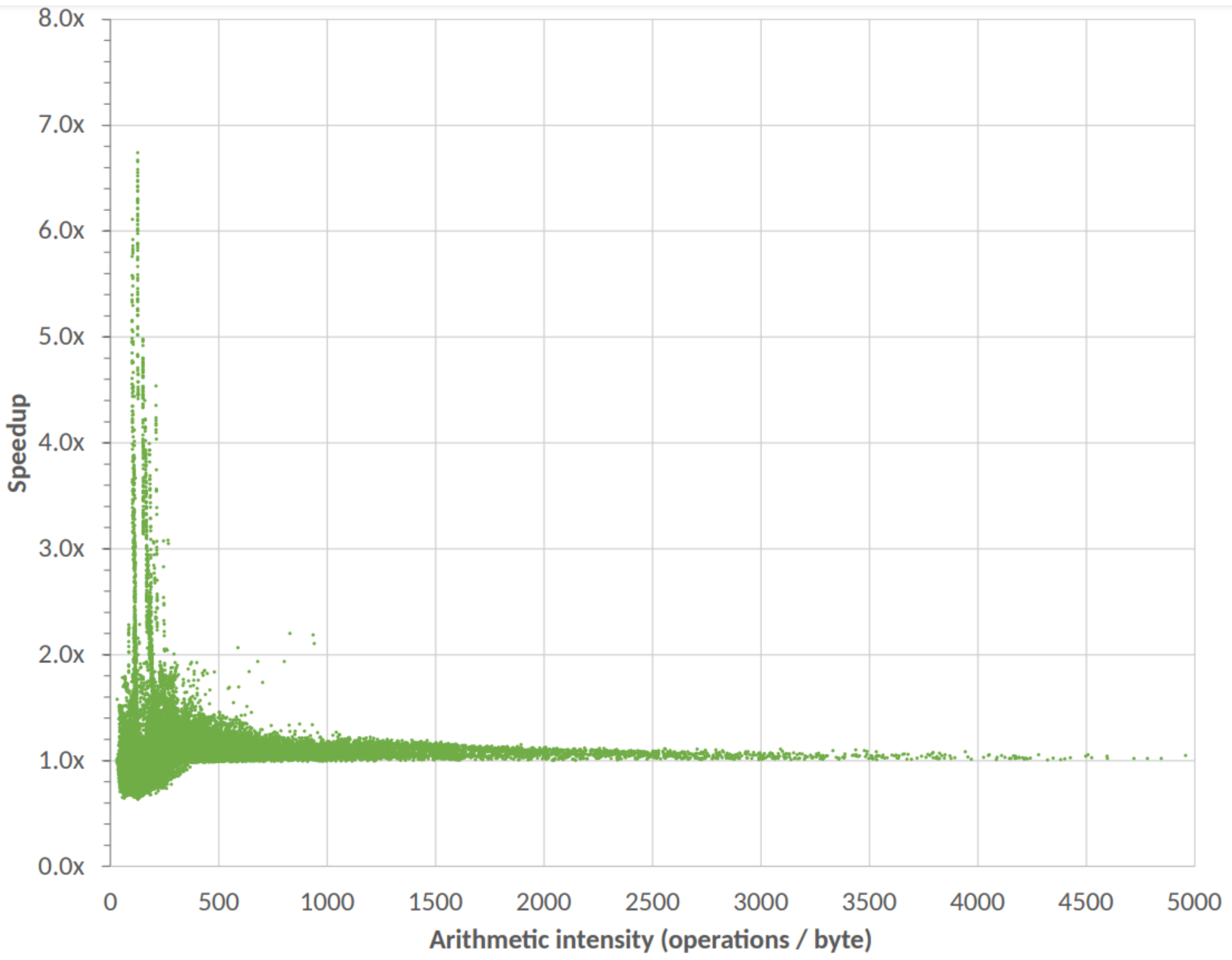
“DATA PARALLEL” MODE

```
...

constexpr int block_size = 128;
int grid_size = num_output_tiles;
gemm<<<grid_size, block_size>>>(...);

...
```



WORK-CENTRIC HGEMM.

UC DAVIS ELECTRICAL AND COMPUTER ENGINEERING



SPEED-UP COMPARED TO CUBLAS.

Generic Programming In CUDA/C++

(1) Lifting

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
       i < N;
       i += blockDim.x * gridDim.x) {
    y[i] = a * x[i] + y[i];
  }
}
```

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
       i < N;
       i += blockDim.x * gridDim.x) {
    y[i] = a * x[i] + y[i];
  }
}
```

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
       i < N;
       i += blockDim.x * gridDim.x) {
    y[i] = a * x[i] + y[i];
  }
}
```

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
       i < N;
       i += blockDim.x * gridDim.x) {
    y[i] = a * x[i] + y[i];
  }
}
```

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  for (auto i : ranges::grid_stride_range(0, N)) {
    y[i] = a * x[i] + y[i];
  }
}
```

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  for (auto i : ranges::grid_stride_range(0, N)) {
    y[i] = a * x[i] + y[i];
  }
}
```

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  for (auto i : ranges::grid_stride_range(0, N)) {
    y[i] = a * x[i] + y[i];
  }
}
```

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  for (auto i : ranges::grid_stride_range(0, N)) {
    y[i] = a * x[i] + y[i];
  }
}
```

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  for (auto i : ranges::grid_stride_range(0, N)) {
    y[i] = a * x[i] + y[i];
  }
}
```

```
template <typename T>
__device__ __forceinline__ step_range_t<T>
  grid_stride_range(T begin, T end) {
  begin += blockIdx.x * blockDim.x + threadIdx.x;
  return range(begin, end).step(blockDim.x * gridDim.x);
}
```

```
template <typename T>
__device__ __forceinline__ step_range_t<T>
  grid_stride_range(T begin, T end) {
  begin += blockIdx.x * blockDim.x + threadIdx.x;
  return range(begin, end).step(blockDim.x * gridDim.x);
}
```

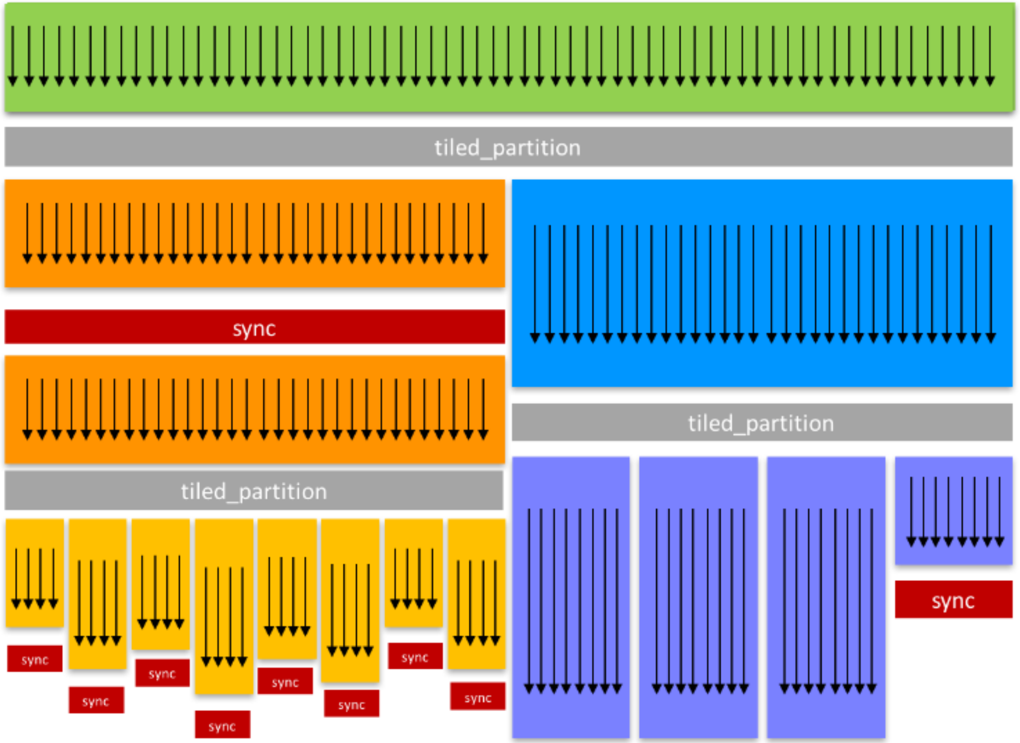
```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  for (auto i : ranges::grid_stride_range(0, N)) {
    y[i] = a * x[i] + y[i];
  }
}
```

Cooperative Groups: Flexible CUDA Thread Programming

By Mark Harris and Kyrylo Perelygin

Discuss (31) +1 Like

Tags: Algorithms, Cooperative Groups, CUDA, Parallel Programming




```
/// Loading an integer from global into shared memory
__global__ void kernel(int *globalInput) {
    __shared__ int x;
    thread_block g = this_thread_block();
    // Choose a leader in the thread block
    if (g.thread_rank() == 0) {
        // load from global into shared for
        // all threads to work with
        x = (*globalInput);
    }
    // After loading data into shared memory,
    // you want to synchronize if all threads
    // in your thread block need to see it
    g.sync(); // equivalent to __syncthreads();
}
```

Generic Programming In CUDA/C++

(2) Specialization

```
__global__
template<typename type_t, typename size_t>
void saxpy(const size_t N, const type_t a, const type_t *x, type_t *y) {
  for (auto i : ranges::grid_stride_range(0, N)) {
    y[i] = a * x[i] + y[i];
  }
}
```

```
...

saxpy<float, int><<<grid_size, block_size>>>(N, 2.0f, x, y);

...
```

```
__global__
void saxpy(const int N, const float a, const float *x, float *y) {
  for (auto i : ranges::grid_stride_range(0, N)) {
    y[i] = a * x[i] + y[i];
  }
}
```

Generic Programming In **C++20**

(3) Concepts

```
__global__
template<typename type_t, typename size_t>
void saxpy(const size_t N, const type_t a, const type_t *x, type_t *y) {
  for (auto i : ranges::grid_stride_range(0, N)) {
    y[i] = a * x[i] + y[i];
  }
}
```

Valid?

Valid?

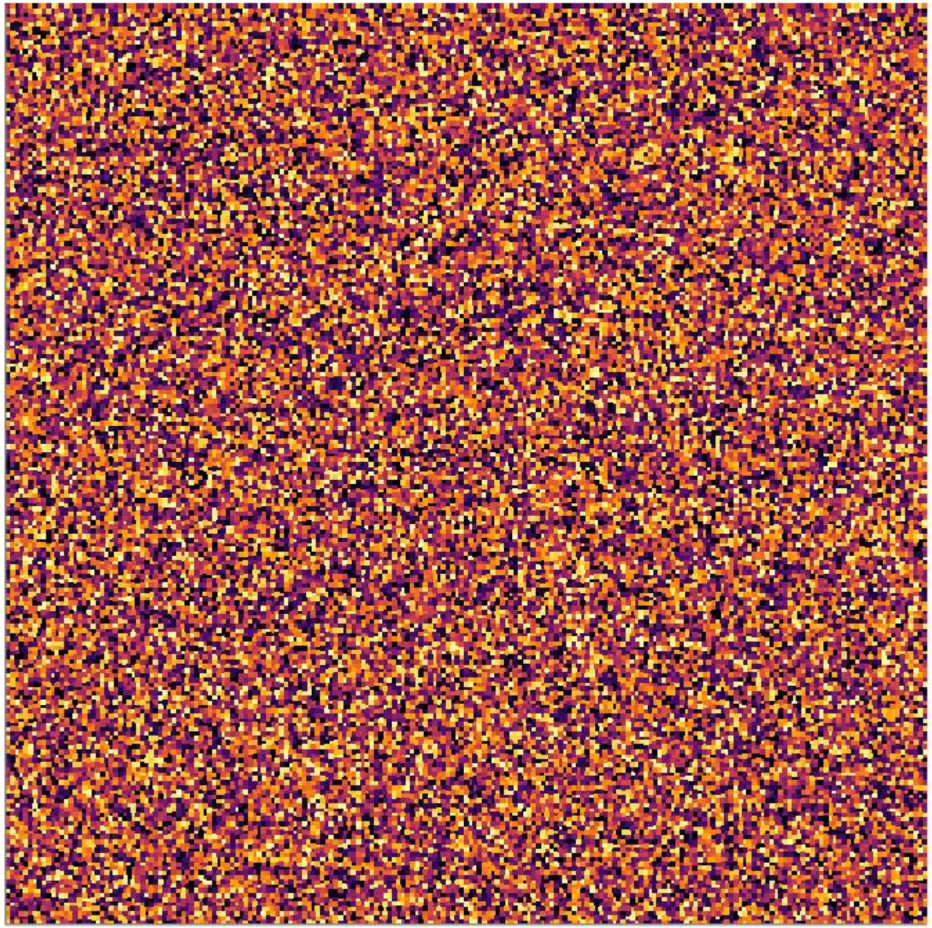
https://en.cppreference.com/w/cpp/concepts

WHY DOES IT MATTER?

- Study existing works.
- “Lift” away unnecessary requirements.
- Repeat the process until we have found generic algorithm.
- Express specializations (types, processes, etc.)
- Catalog requirements (concepts.)

Generic programming is an instrumental process in developing abstractions.

WHAT WE HAVE LOOKED AT SO FAR.



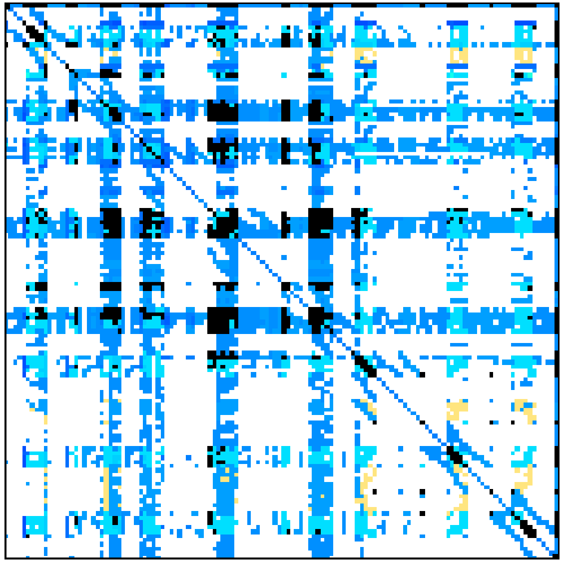
Regular problem.

Memory system's heaven.

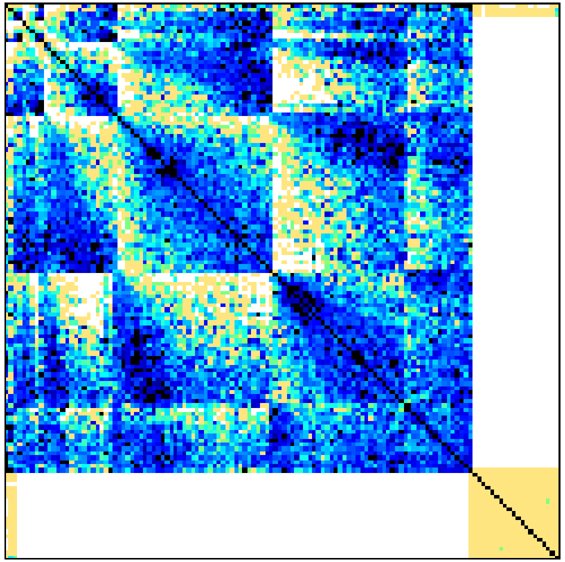
We solved the problem

(100% utilization)

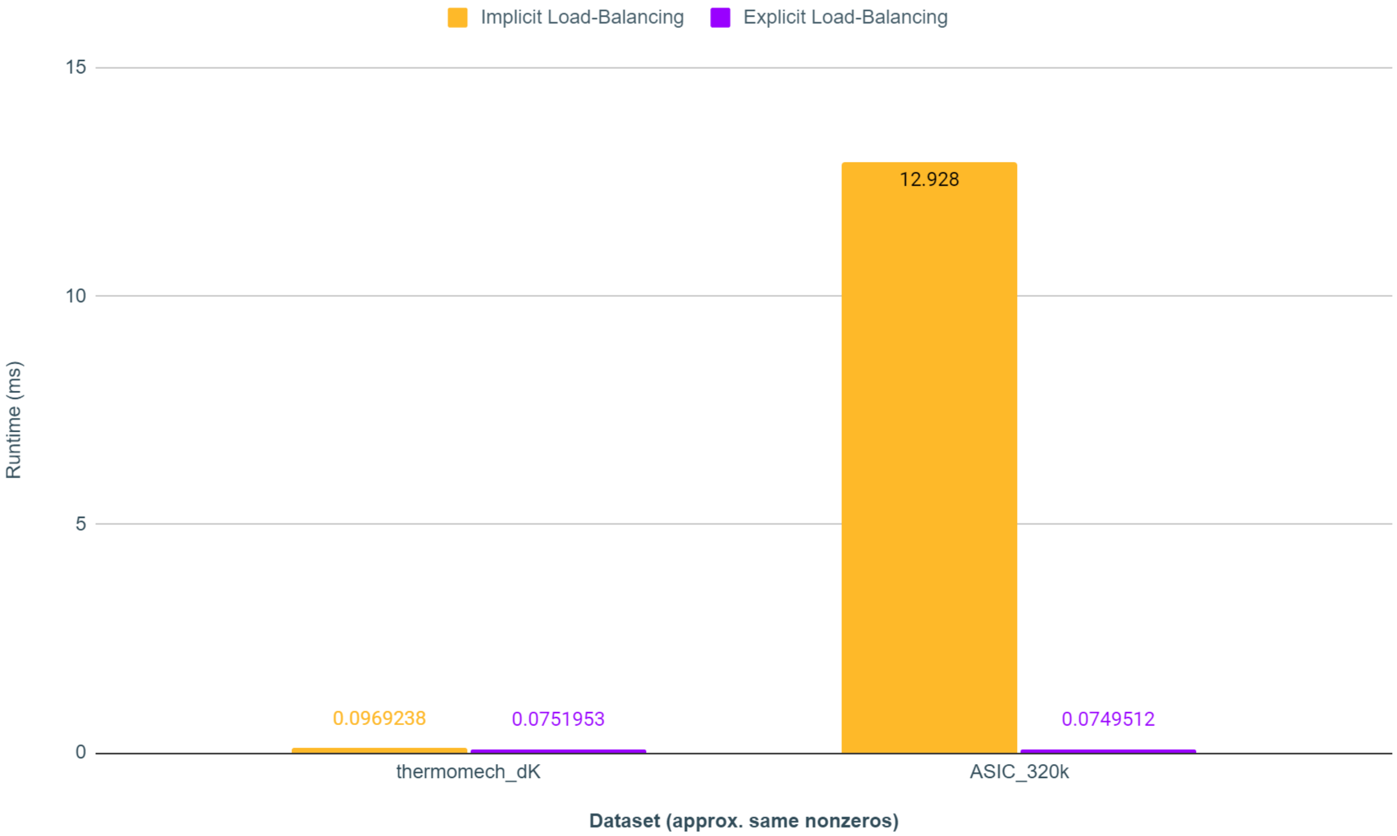
BUT WHAT IF… WORLD ISN'T ALL NICE.

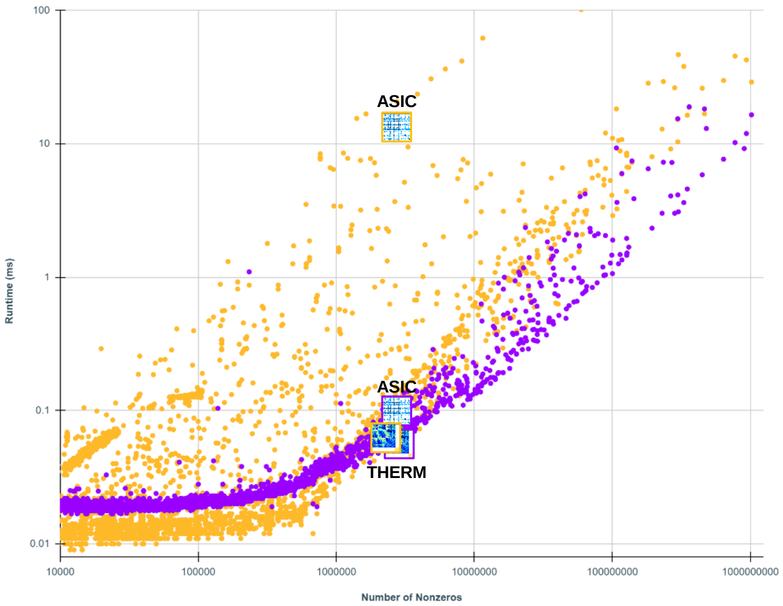


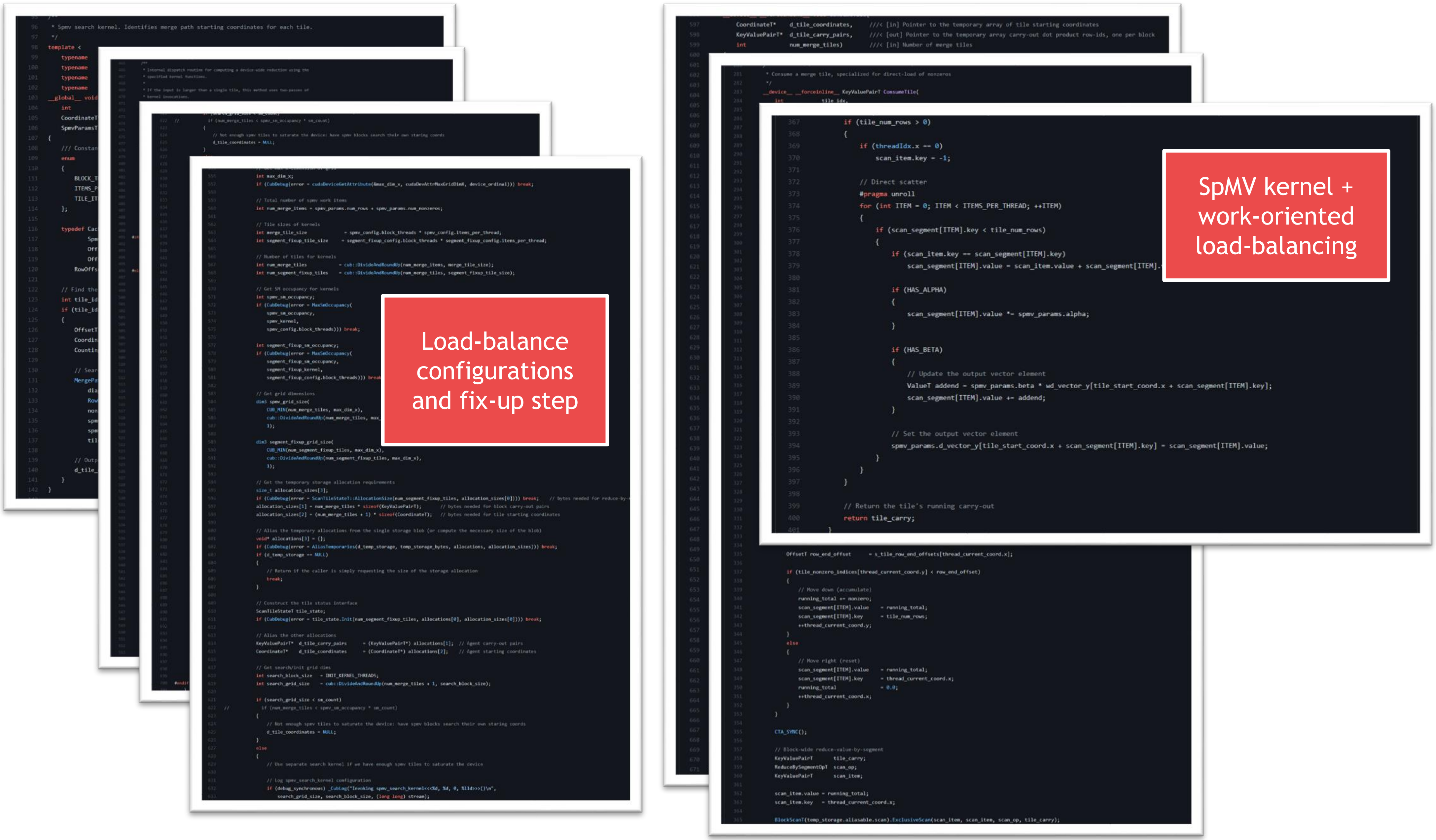
ASIC_320K
Circuit Simulation
2,635,364 nonzeros



Thermomech_dK
Temperature & Deformation
2,846,228 nonzeros


Implicit Load-Balancing
Explicit Load-Balancing
15
10
5
0
Runtime (ms)
12.928
0.0969238
0.0751953
0.0749512
thermomech_dK
ASIC_320k
Dataset (approx. same nonzeros)


100
10
1
0.1
0.01
Runtime (ms)
ASIC
ASIC
THERM
10000
100000
1000000
10000000
100000000
1000000000
Number of Nonzeros


Load-balance configurations and fix-up step
SpMV kernel + work-oriented load-balancing
if (tile_num_rows > 0)
{
if (threadIdx.x == 0)
scan_item.key = -1;
// Direct scatter
#pragma unroll
for (int ITEM = 0; ITEM < ITEMS_PER_THREAD; ++ITEM)
{
if (scan_segment[ITEM].key < tile_num_rows)
{
if (scan_item.key == scan_segment[ITEM].key)
scan_segment[ITEM].value = scan_item.value + scan_segment[ITEM].
if (HAS_ALPHA)
{
scan_segment[ITEM].value *= spmv_params.alpha;
}
if (HAS_BETA)
{
// Update the output vector element
ValueT addend = spmv_params.beta * wd_vector_y[tile_start_coord.x + scan_segment[ITEM].key];
scan_segment[ITEM].value += addend;
}
// Set the output vector element
spmv_params.d_vector_y[tile_start_coord.x + scan_segment[ITEM].key] = scan_segment[ITEM].value;
}
}
}
// Return the tile's running carry-out
return tile_carry;
}

```
#pragma unroll
for (int ITEM = 0; ITEM < ITEMS_PER_THREAD; ++ITEM)
{
    OffsetT nonzero_idx         = CUB_MIN(tile_nonzero_indices[thread_current_coord.y], spmv_params.num_nonzeros - 1);
    OffsetT column_idx          = wd_column_indices[nonzero_idx];
    ValueT  value               = wd_values[nonzero_idx];

    ValueT  vector_value        = wd_vector_x[column_idx];

    ValueT  nonzero             = value * vector_value;

    OffsetT row_end_offset      = s_tile_row_end_offsets[thread_current_coord.x];

    if (tile_nonzero_indices[thread_current_coord.y] < row_end_offset)
    {
        // Move down (accumulate)
        running_total += nonzero;
        scan_segment[ITEM].value    = running_total;
        scan_segment[ITEM].key      = tile_num_rows;
        ++thread_current_coord.y;
    }
    else
    {
        // Move right (reset)
        scan_segment[ITEM].value    = running_total;
        scan_segment[ITEM].key      = thread_current_coord.x;
        running_total               = 0.0;
        ++thread_current_coord.x;
    }
}
```

Equates to the actual computation the user wants to do:
$y = Ax$

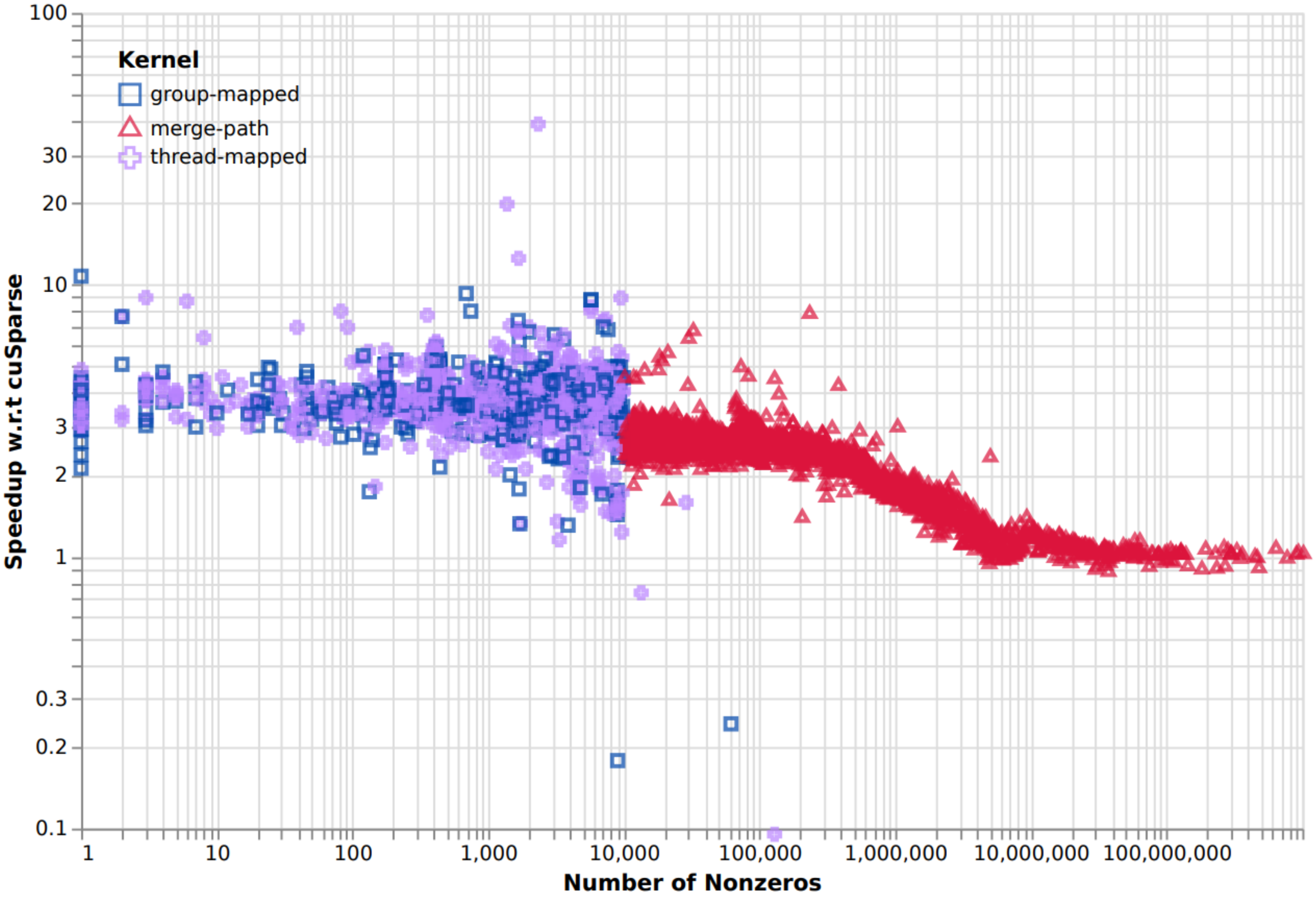
(1) GITHUB.COM/NVIDIA/CUB

```
__global__ void spmv(const csr_t* A, const float *x, float *y) {
  using namespace loops;

  auto config = setup<schedule::merge_path>(
       A->offsets, A->rows, A->nnz);

  float temp = 0.0f;
  // loop over all assigned tiles
  for (auto m : tile::range(config)) {
    // loop over all assigned atoms per tile
    for (auto k : atom::range(config))
      temp += A->values[k] * x[A->indices[k]];
    y[m] = temp;
    temp = 0.0f;
  }
}
```

```
__global__ void spmv(const csr_t* A, const float *x, float *y) {
  using namespace loops;

  auto config = setup<schedule::merge_path>(
       A->offsets, A->rows, A->nnz);

  float temp = 0.0f;
  // loop over all assigned tiles
  for (auto m : tile::range(config)) {
    // loop over all assigned atoms per tile
    for (auto k : atom::range(config))
      temp += A->values[k] * x[A->indices[k]];
    y[m] = temp;
    temp = 0.0f;
  }
}
```

```
__global__ void spmv(const csr_t* A, const float *x, float *y) {
  using namespace loops;

  auto config = setup<schedule::merge_path>(
       A->offsets, A->rows, A->nnz);

  float temp = 0.0f;
  // loop over all assigned tiles
  for (auto m : tile::range(config)) {
    // loop over all assigned atoms per tile
    for (auto k : atom::range(config))
      temp += A->values[k] * x[A->indices[k]];
    y[m] = temp;
    temp = 0.0f;
  }
}
```

```
// return the range of tiles to process
__device__ step_range_t<tile_size_t> tiles() const {
    return loops::grid_stride_range(tile_size_t(0),
                                    tile_traits_t::size());
}

// return the range of atoms to process
__device__ auto atoms(const tile_size_t& tid) const {
    return loops::range(tile_traits_t::begin()[tid],
                        tile_traits_t::begin()[tid + 1]);
}
```



LOAD BALANCING ABSTRACTED: SPMV

Everything else.

Use `thrust` when you can.

You’ll be tempted to write your own “heterogeneous” array (CPU/GPU). **Don’t.**

CUDA error reporting is terrible. **Good luck.**

Spend a few days learning Nsight (System and Compute).

Don’t write your own binary search.

Use GitHub, Doxygen, GitHub actions (CI) and Googletests (Unit testing).

```
constexpr int N = 1<<20;
float* x; float* y;
cudaMalloc(&x, N * sizeof(float));
cudaMalloc(&y, N * sizeof(float));
cudaMemset(x, 1.0f, N * sizeof(float));
cudaMemset(y, 2.0f, N * sizeof(float));

constexpr int block_size = 128;
int grid_size = (N - block_size + 1) / block_size;
saxpy<<<grid_size, block_size>>>(N, 2.0f, x, y);

cudaFree(x); cudaFree(y);
```

You are going to forget to free the memory.

```
constexpr std::size_t N = 1<<20;

thrust::device_vector<float> x(N, 1.0f);
thrust::device_vector<float> y(N, 2.0f);

constexpr int block_size = 128;
int grid_size = 2 * SMs;
saxpy<<<grid_size, block_size>>>(N, 2.0f, x.data(), y.data());
```

Smart memory.
No free required.

Use `thrust` when you can.

You'll be tempted to write your own "heterogeneous" array (CPU/GPU). **Don't.**

CUDA error reporting is terrible. **Good luck.**

Spend a few days learning Nsight (System and Compute).

Don't write your own binary search.

Use GitHub, Doxygen, GitHub actions (CI) and Googletests (Unit testing).

Use `thrust` when you can.

You'll be tempted to write your own "heterogeneous" array (CPU/GPU). **Don't.**

CUDA error reporting is terrible. **Good luck.**

Spend a few days learning Nsight (System and Compute).

Don't write your own binary search.

Use GitHub, Doxygen, GitHub actions (CI) and Googletests (Unit testing).

Use `thrust` when you can.

You’ll be tempted to write your own “heterogeneous” array (CPU/GPU). **Don’t.**

CUDA error reporting is terrible. **Good luck.**

Spend a few days learning Nsight (System and Compute).

Don’t write your own binary search.

Use GitHub, Doxygen, GitHub actions (CI) and Googletests (Unit testing).

Use `thrust` when you can.

You’ll be tempted to write your own “heterogeneous” array (CPU/GPU). **Don’t.**

CUDA error reporting is terrible. **Good luck.**

Spend a few days learning Nsight (System and Compute).

Don’t write your own binary search.

Use GitHub, Doxygen, GitHub actions (CI) and Googletests (Unit testing).

```
__device__ __host__
int binary_search(int* array, int x, int low, int high) {
  while (low <= high) {
    int mid = (low + high) / 2;
    if (array[mid] == x)
      return mid;
    if (array[mid] < x)
      low = mid + 1;
    else
      high = mid - 1;
  }
  return -1;
}
```

```
__device__ __host__
int binary_search(int* array, int x, int low, int high) {
  while (low <= high) {
    int mid = (low + high) / 2;
    if (array[mid] == x)
      return mid;
    if (array[mid] < x)
      low = mid + 1;
    else
      high = mid - 1;
  }
  return -1;
}
```

```
__device__ __host__
int binary_search(int* array, int x, int low, int high) {
  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (array[mid] == x)
      return mid;
    if (array[mid] < x)
      low = mid + 1;
    else
      high = mid - 1;
  }
  return -1;
}
```

Use `thrust` when you can.

You’ll be tempted to write your own “heterogeneous” array (CPU/GPU). **Don’t.**

CUDA error reporting is terrible. **Good luck.**

Spend a few days learning Nsight (System and Compute).

Don’t write your own binary search.

Use GitHub, Doxygen, GitHub actions (CI) and Googletests (Unit testing).

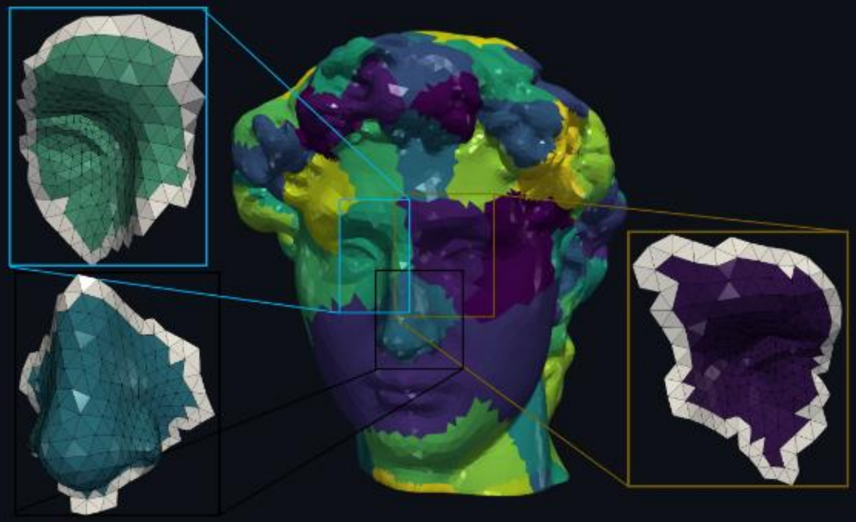
main 4 branches 3 tags

Ahdhn RXMesh v0.2.1 (#13) ...

.github/workflows	RXMesh v0.2
apps	RXMesh v0.2
assets	RXMesh v0.2
cmake	RXMesh v0.2
include	RXMesh v0.2
input	RXMesh v0.2
tests	RXMesh v0.2
.clang-format	RXMesh v0.2
.gitignore	RXMesh v0.2
CMakeLists.txt	RXMesh v0.2
LICENSE	RXMesh v0.2
README.md	RXMesh v0.2

README.md

RXMesh Ubuntu passing Windows passing



Contents

main 5 branches 0 tags

maawad Add link to website with documentation

benchmarks	Add benchmarking results
cmake	Add code and archived results
figs	Add A100 results
include	Add A100 results
results/arxiv	Add A100 results
scripts	Add A100 results
test	Fix typo
.clang-format	Add initial source code
.gitignore	Add results for Titan Xp
CMakeLists.txt	Add initial unit tests
LICENSE	Add code and archived results
README.md	Add link to website with docur
reproduce.md	Add code and archived results
results.md	Add A100 results

README.md

BGHT: Better GPU Hash Tables

Examples/Tests	Benchmarks	Results

BGHT is a collection of high-performance static GPU hash tables. BGHT contains hash tables that use three different probing schemes 1) bucketed cuckoo, 2) power-of-two, 3) iceberg hashing. Our bucketed static cuckoo hash table is the state-of-art static hash table. For more information, please check our paper:

Better GPU Hash Tables
Muhammad A. Awad, *Saman Ashkiani*, *Serban D. Porumbescu*, *Martín Farach-Colton*, and *John D. Owens*

Key features

- State-of-the-art static GPU hash tables
- Device and host side APIs
- Support for different types of keys and values
- Standard-like APIs

How to use

BGHT is a header-only library. To use the library, you can add it as a submodule or use CMake Package Manager (CPM) to fetch the library into your CMake-based project (complete example).

```
cmake_minimum_required(VERSION 3.8 FATAL_ERROR)
CPMAddPackage(
  NAME bght
  GITHUB_REPOSITORY owensgroup/BGHT
  GIT_TAG main
  OPTIONS
     "build_tests OFF"
     "build_benchmarks OFF"
)
target_link_libraries(my_library PRIVATE bght)
```

https://github.com/neoblizz/neoblizz/wiki

CUDA 301.

Solve Multi-GPUs and Multi-Node.

CUDA 301.

Solve Multi-GPUs and Multi-Node.

You got this.

THANK YOU ALL!

John D. Owens

Serban D. Porumbescu

Jason Lowe-Power, Soheil Ghiasi, Venkatesh Akella

Stephen Jones, Michael Garland, Duane Merrill, Cris Cecka

Sean Treichler, Aamer Jaleel

Owensgroup lab students!