The 8th Vulkan Developer Conference
San Diego, USA | February 9-11, 2026

Vulkan at the Speed of Light: Benchmarking & Auto-Tuning Tactics

Reiner Dolp, Karlsruhe Institute of Technology (KIT)

Motivation

Radixsort Shaders
NVIDIA RTX 2080

1. **Elements Per Thread** = 14
   Workgroup Size = 512
2. **Elements Per Thread** = 8
   Workgroup Size = 512
3. **Elements Per Thread** = 8
   Workgroup Size = 256

Motivation

Radixsort Shaders
NVIDIA RTX 2080

1. **Elements Per Thread** = 14
   Workgroup Size = 512
2. **Elements Per Thread** = 8
   Workgroup Size = 512
3. **Elements Per Thread** = 15
   Workgroup Size = 256

Motivation

Radixsort Shaders
NVIDIA RTX 2080

the histogram bins different radixsort shaders by runtime,
fastest on 128MiB 32-bit numbers to the right

Best @ 128.00 KiB
Best @ 512.00 KiB
Best @ 4.00 MiB
Best @ 32.00 MiB
Best @ 128.00 MiB (current)
Other

Number of Configurations
Cumulative %
Speedup Relative to Best Configuration

Even on the same device, shader performance is argument dependent

Motivation

Radixsort Shaders
NVIDIA RTX 2080

the histogram bins different radixsort shaders by runtime,
fastest on 128MiB 32-bit numbers to the right

Grain Size
8 12 16 21 25 30

Number of Configurations
Cumulative %
Speedup Relative to Best Configuration

Even on the same device, shader performance is argument dependent

Motivation

Radixsort Shaders
NVIDIA RTX 2080

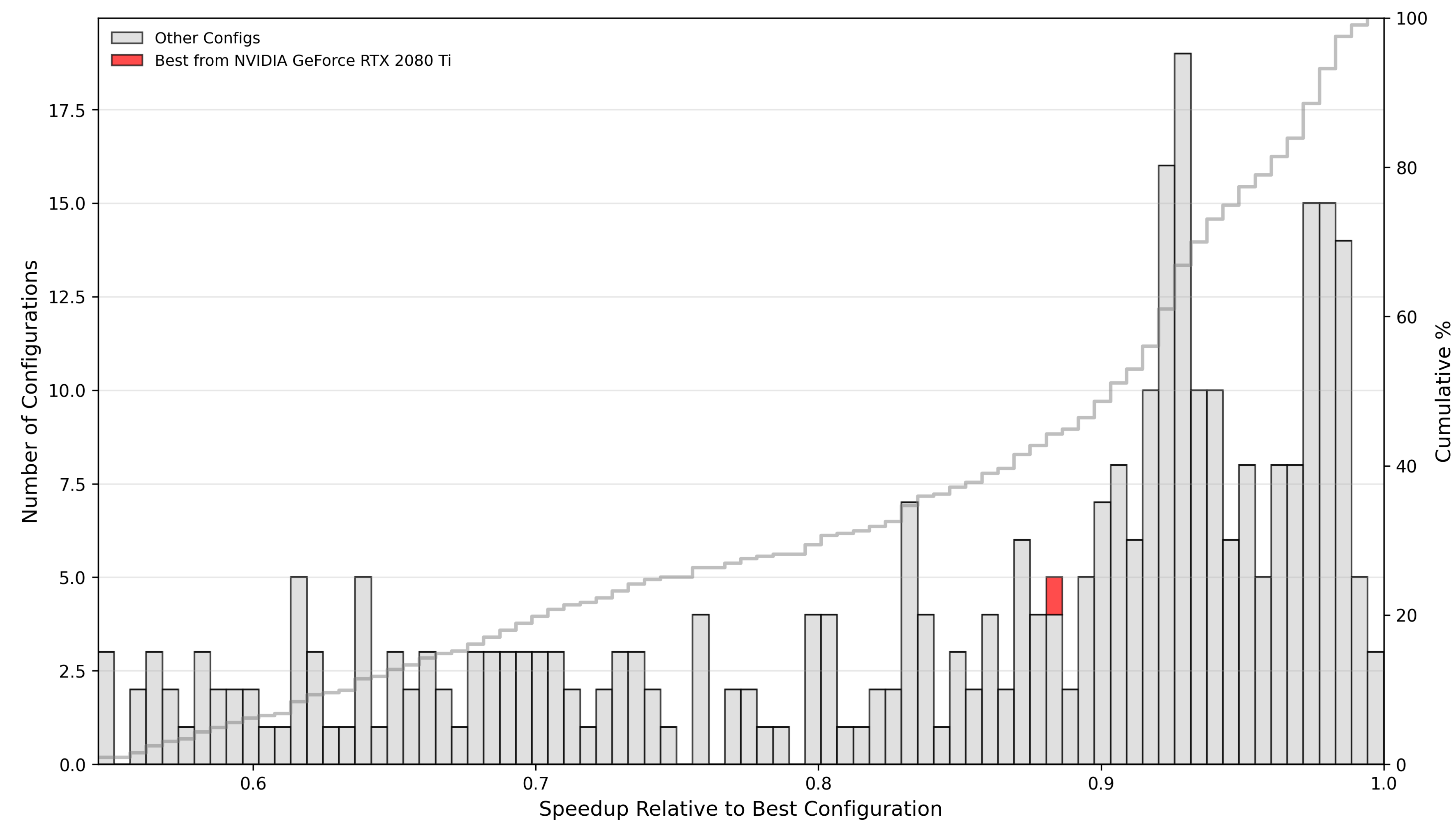
the histogram bins different radixsort shaders by runtime,
fastest on 128MiB 32-bit numbers to the right

Group Size X
256 384 544 704 864 1024
Number of Configurations
Cumulative %
Speedup Relative to Best Configuration

Even on the same device, shader performance is argument dependent

Motivation

Radixsort Shaders
NVIDIA RTX 2080 vs NVIDIA RTX 3070, 512KiB



Vulkan is portable & performant

Shaders are not performance portable. There is no “single best shader”

Approach — Policy-based Autotuning [1]

Runtime Workflow

```C++
void sort(buffer_t<T> out, buffer_t<T> in) {
  sort_decision_vars_t decision_vars = {
     .element_type = type_id<T>,
     .problem_size = in.size(),
     .max_shmem = device.maxShmemBytes()
     .atomic64 = device.supports64bitAtomics(),
     .deviceFingerprint = device.fingerprint()
  };
  auto impl = sort_t::select(decision_vars);
  impl->alloc_pipeline(pipeline_cache);
  impl->alloc_temporaries(mem_allocator);
  impl->set_arguments(out,in);
  impl->dispatch();
  impl->free_temporaries(mem_allocator);
}
```

Policy Pattern, also known as **Strategy Pattern:**
From a family of algorithms, select an algorithm at runtime.
Natural pattern on GPUs: staged compilation, any shader is parametric. For each logical operation:

1. **Policy Selection**
   Inspect decision variables (argument list and Environment Constraints) to get a policy:
   a pipeline, temporary storage requirements, values for tunable variables (Workgroup Size, ...)
2. **Setup Dispatch**
   Write Descriptors, Allocate Temporary Memory
3. **Dispatch**
   Record Commands

Approach — Policy-based Autotuning [1]

Runtime Workflow

```cpp
void sort(buffer_t<T> out, buffer_t<T> in) {
  sort_decision_vars_t decision_vars = {
     .element_type = type_id<T>,
     .problem_size = in.size(),
     .max_shmem = device.maxShmemBytes()
     .atomic64 = device.supports64bitAtomics(),
     .deviceFingerprint = device.fingerprint()
  };
  auto impl = sort_t::select(decision_vars);
  impl->alloc_pipeline(pipeline_cache);
  impl->alloc_temporaries(mem_allocator);
  impl->set_arguments(out,in);
  impl->dispatch(commandBuffer);
  impl->free_temporaries(mem_allocator);
}
```

Policy Pattern, also known as **Strategy Pattern:** From a family of algorithms, select an algorithm at runtime.
Natural pattern on GPUs: staged compilation, any shader is parametric. For each logical operation:

1. **Policy Selection**
   Inspect decision variables (argument list and Environment Constraints) to get a policy:
   a pipeline, temporary storage requirements, values for tunable variables (Workgroup Size, ...)
2. **Setup Dispatch**
   Write Descriptors, Allocate Temporary Memory
3. **Dispatch**
   Record Commands

Approach — Policy-based Autotuning [1]

Runtime Workflow

C++

```
void sort(buffer_t<T> out, buffer_t<T> in) {
  sort_decision_vars_t decision_vars = {
     .element_type = type_id<T>,
     .problem_size = in.size(),
     .max_shmem = device.maxShmemBytes()
     .atomic64 = device.supports64bitAtomics(),
     .deviceFingerprint = device.fingerprint()
  };
  auto impl = sort_t::select(decision_vars);
  impl->alloc_pipeline(pipeline_cache);
  impl->alloc_temporaries(mem_allocator);
  impl->set_arguments(out,in);
  impl->dispatch(commandBuffer);
  impl->free_temporaries(mem_allocator);
}
```

Policy Pattern, also known as **Strategy Pattern:** From a family of algorithms, select an algorithm at runtime.
Natural pattern on GPUs: staged compilation, any shader is parametric. For each logical operation:

1. **Policy Selection**
   Inspect decision variables (argument list and Environment Constraints) to get a policy:
   a pipeline, temporary storage requirements, values for tunable variables (Workgroup Size, ...)
2. **Setup Dispatch**
   Write Descriptors, Allocate Temporary Memory
3. **Dispatch**
   Record Commands

Approach — Policy-based Autotuning [1]

Runtime Workflow

```cpp
void sort(buffer_t<T> out, buffer_t<T> in) {
  sort_decision_vars_t decision_vars = {
     .element_type = type_id<T>,
     .problem_size = in.size(),
     .max_shmem = device.maxShmemBytes()
     .atomic64 = device.supports64bitAtomics(),
     .deviceFingerprint = device.fingerprint()
  };
  auto impl = sort_t::select(decision_vars);
  impl->alloc_pipeline(pipeline_cache);
  impl->alloc_temporaries(mem_allocator);
  impl->set_arguments(out,in);
  impl->dispatch(commandBuffer);
  impl->free_temporaries(mem_allocator);
}
```

Policy Pattern, also known as **Strategy Pattern:**
From a family of algorithms, select an algorithm at runtime.
Natural pattern on GPUs: staged compilation, any shader is parametric. For each logical operation:

1. **Policy Selection**
   Inspect decision variables (argument list and Environment Constraints) to get a policy:
   a pipeline, temporary storage requirements, values for tunable variables (Workgroup Size, ...)
2. **Setup Dispatch**
   Write Descriptors, Allocate Temporary Memory
3. **Dispatch**
   Record Commands

Approach — Policy-based Autotuning [1]

Development Workflow

1 Search Space Construction
GPU Expert uses *profiling* [8] to build parametric implementations

2 Search Space Evaluation
Benchmark all shaders for all possible parameterizations

3 Policy Creation
analyse benchmark results to create policies

tuning space is cartesian product of all tuning parameters

- **shader inputs and outputs:** types, problem size, value distribution

any compute shader implementation has tuning parameters:

- **environment restrictions:** shmem size, has atomics
- **environment:** workgroup size, dispatch size

Approach — Policy-based Autotuning [1]

Development Workflow

1 Search Space Construction
GPU Expert uses *profiling* [8] to build parametric implementations

2 Search Space Evaluation
Benchmark all shaders for all possible parameterizations

3 Policy Creation
analyse benchmark results to create policies

tuning space is cartesian product of all tuning parameters

- **shader inputs and outputs:** types, problem size, value distribution

any compute shader implementation has tuning parameters:

- **environment restrictions:** shmem size, has atomics
- **environment:** workgroup size, dispatch size

histogram_gmem.glsl — GLSL

```glsl
for (uint i = 0u; i < GRAIN_SIZE; i++) {
    uint idx = // ...
    if (idx < pc.numElements) {
        TYPE_INPUT value = inputBuffer.data[idx];
        uint32_t bin = // ...

        atomicAdd(globalHistogram.bins[bin], 1u);
    }
}
```

Approach — Policy-based Autotuning [1]

Development Workflow

1 Search Space Construction
GPU Expert uses *profiling* [8] to build parametric implementations

2 Search Space Evaluation
Benchmark all shaders for all possible parameterizations

3 Policy Creation
analyse benchmark results to create policies

tuning space is cartesian product of all tuning parameters
- **shader inputs and outputs:** types, problem size, value distribution

any compute shader implementation has tuning parameters:
- **environment restrictions:** shmem size, has atomics
- **environment:** workgroup size, dispatch size

histogram_shmem.glsl — GLSL

```glsl
shared uint s_histogram[NUM_BINS * SHMEM_HIST_COUNT];
for(/*...*/) { s_histogram[i] = 0u; } barrier();
for (uint i = 0u; i < GRAIN_SIZE; i++) {
    uint idx = /* ... */; uint s_offset = /* ... */;
    if (idx < pc.numElements) {
        TYPE_INPUT value = inputBuffer.data[idx];
        uint32_t bin = // ...

        atomicAdd(s_histogram.bins[s_offset + bin], 1u);
    }
}
for(/* each bin...*/) { uint32_t wg_bin = /*...*/;
    if(wg_bin) atomicAdd(globalHistogram.bins[bin], wg_bin); }}
```

Approach — Policy-based Autotuning [1]

Development Workflow

1 Search Space Construction
GPU Expert uses *profiling* [8] to build parametric implementations

2 Search Space Evaluation
Benchmark all shaders for all possible parameterizations

3 Policy Creation
analyse benchmark results to create policies

tuning space is cartesian product of all tuning parameters
- **shader inputs and outputs:** types, problem size, value distribution

any compute shader implementation has tuning parameters:
- **environment restrictions:** shmem size, has atomics
- **environment:** workgroup size, dispatch size

histogram_shmem.glsl — GLSL

```glsl
shared TYPE_SHMEM_BIN s_histogram[NUM_BINS * SHMEM_HIST_COUNT];
for(/*...*/) { s_histogram[i] = 0u; } barrier();
for (uint i = 0u; i < TILE_COUNT; i++) {
for (uint i = 0u; i < GRAIN_SIZE; i++) {
    uint idx = /* ... */; uint s_offset = /* ... */;
    if (idx < pc.numElements) {
        TYPE_INPUT value = inputBuffer.data[idx];
        uint32_t bin = // ...

        atomicAdd(s_histogram.bins[s_offset + bin], 1u);
    }
}}
for(/* each bin...*/) { uint32_t wg_bin = /*...*/;
    if(wg_bin) atomicAdd(globalHistogram.bins[bin], wg_bin); }}
```

Approach — Policy-based Autotuning [1]

Development Workflow

1. **Search Space Construction**
   GPU Expert uses *profiling* [8] to build parametric implementations
2. **Search Space Evaluation**
   Benchmark all shaders for all possible parameterizations
3. **Policy Creation**
   analyse benchmark results to create policies

Yields a **Runtime Database:**
For each comparable benchmark (same decision variables), a ranking of algorithm tunings

console window

```
PROBLEM SIZE: 134217728 bytes (with additional 0 bytes padding) = 128 MiB = 33554432 elements to histogram
TYPE INPUT: uint32_t, TYPE HIST: uint32_t, NUM_BINS: 256, LOWER_LEVEL: 0, UPPER_LEVEL: 4294967295

Workload        Algorithm                                                    Time (ms)           Throughput    Speedup    Scratch  Valid
---------------------------------------------------------------------------------------------------------------------------------------
histogram<...>shmem_persistent<GRAIN_SIZE=32u,PROCESSOR_COUNT=2048u...      0.5163 +- 0.0005 ms   484.23 GiB/s   1.00x      4 B    YES
histogram<...>shmem<GRAIN_SIZE=32u,SHMEM_HIST_COUNT=1u,groupSizeX=1...      0.5168 +- 0.0003 ms   483.79 GiB/s   1.00x      0 B    YES
histogram<...>shmem<GRAIN_SIZE=32u,SHMEM_HIST_COUNT=1u,groupSizeX=2...      0.5172 +- 0.0002 ms   483.33 GiB/s   1.00x      0 B    YES
histogram<...>shmem_persistent<GRAIN_SIZE=32u,PROCESSOR_COUNT=1024u...      0.5173 +- 0.0011 ms   483.29 GiB/s   1.00x      4 B    YES
histogram<...>shmem_persistent<GRAIN_SIZE=64u,PROCESSOR_COUNT=512u,...      0.5176 +- 0.0007 ms   483.03 GiB/s   1.00x      4 B    YES
histogram<...>shmem_persistent<GRAIN_SIZE=64u,PROCESSOR_COUNT=2048u...      0.5177 +- 0.0004 ms   482.88 GiB/s   1.00x      4 B    YES
histogram<...>shmem_persistent<GRAIN_SIZE=64u,PROCESSOR_COUNT=1024u...      0.5179 +- 0.0005 ms   482.76 GiB/s   1.00x      4 B    YES
histogram<...>shmem_persistent<GRAIN_SIZE=64u,PROCESSOR_COUNT=2048u...      0.5180 +- 0.0006 ms   482.65 GiB/s   1.00x      4 B    YES
histogram<...>shmem_persistent<GRAIN_SIZE=64u,PROCESSOR_COUNT=1024u...      0.5181 +- 0.0006 ms   482.49 GiB/s   1.00x      4 B    YES
histogram<...>shmem<GRAIN_SIZE=32u,SHMEM_HIST_COUNT=1u,groupSizeX=1...      0.5182 +- 0.0004 ms   482.40 GiB/s   1.00x      0 B    YES
... skipping 3052 rows.
histogram<...>gmem<GRAIN_SIZE=8u,groupSizeX=608u>                          11.5296 +- 0.6002 ms    21.68 GiB/s   0.04x      0 B    YES
histogram<...>gmem<GRAIN_SIZE=4u,groupSizeX=992u>                          11.5356 +- 0.7664 ms    21.67 GiB/s   0.04x      0 B    YES
histogram<...>gmem<GRAIN_SIZE=1u,groupSizeX=544u>                          11.5429 +- 0.5302 ms    21.66 GiB/s   0.04x      0 B    YES
histogram<...>gmem<GRAIN_SIZE=1u,groupSizeX=576u>                          11.5448 +- 0.5191 ms    21.65 GiB/s   0.04x      0 B    YES
histogram<...>gmem<GRAIN_SIZE=4u,groupSizeX=800u>                          11.5505 +- 0.4891 ms    21.64 GiB/s   0.04x      0 B    YES
histogram<...>gmem<GRAIN_SIZE=1u,groupSizeX=992u>                          11.5546 +- 0.4943 ms    21.64 GiB/s   0.04x      0 B    YES
```

Approach — Policy-based Autotuning [1]

Development Workflow

1 Search Space Construction
GPU Expert uses *profiling* [8] to build parametric implementations

2 Search Space Evaluation
Benchmark all shaders for all possible parameterizations

3 Policy Creation
analyse benchmark results to create policies

e.g. limitting ourselves to *K=2* shaders:

```
implementation: hist_shmem.glsl
enable_if: sizeof(bin_type) * bin_count * privatization
           ≤ shmem_size
tuning: groupSizeX=256 grainSize=8
```

```
implementation: hist_gmem.glsl
enable_if: sizeof(bin_type) * bin_count * privatization
           > shmem_size
tuning: groupSizeX=256 grainSize=8
```

Approach — Policy-based Autotuning [1]

Development Workflow

1 Search Space Construction
GPU Expert uses *profiling* [8] to build parametric implementations

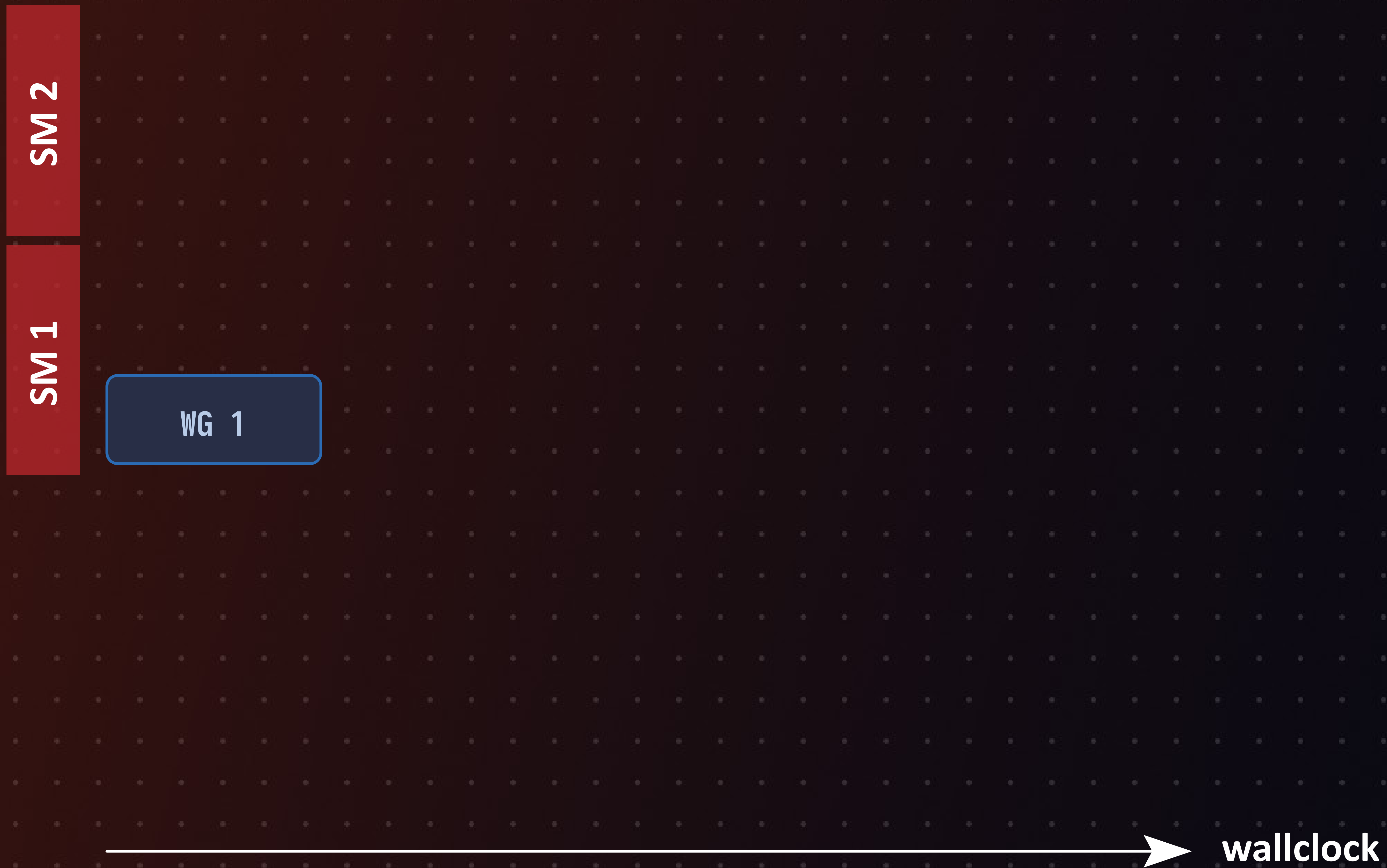
2 Search Space Evaluation
Benchmark all shaders for all possible parameterizations

3 Policy Creation
analyse benchmark results to create policies

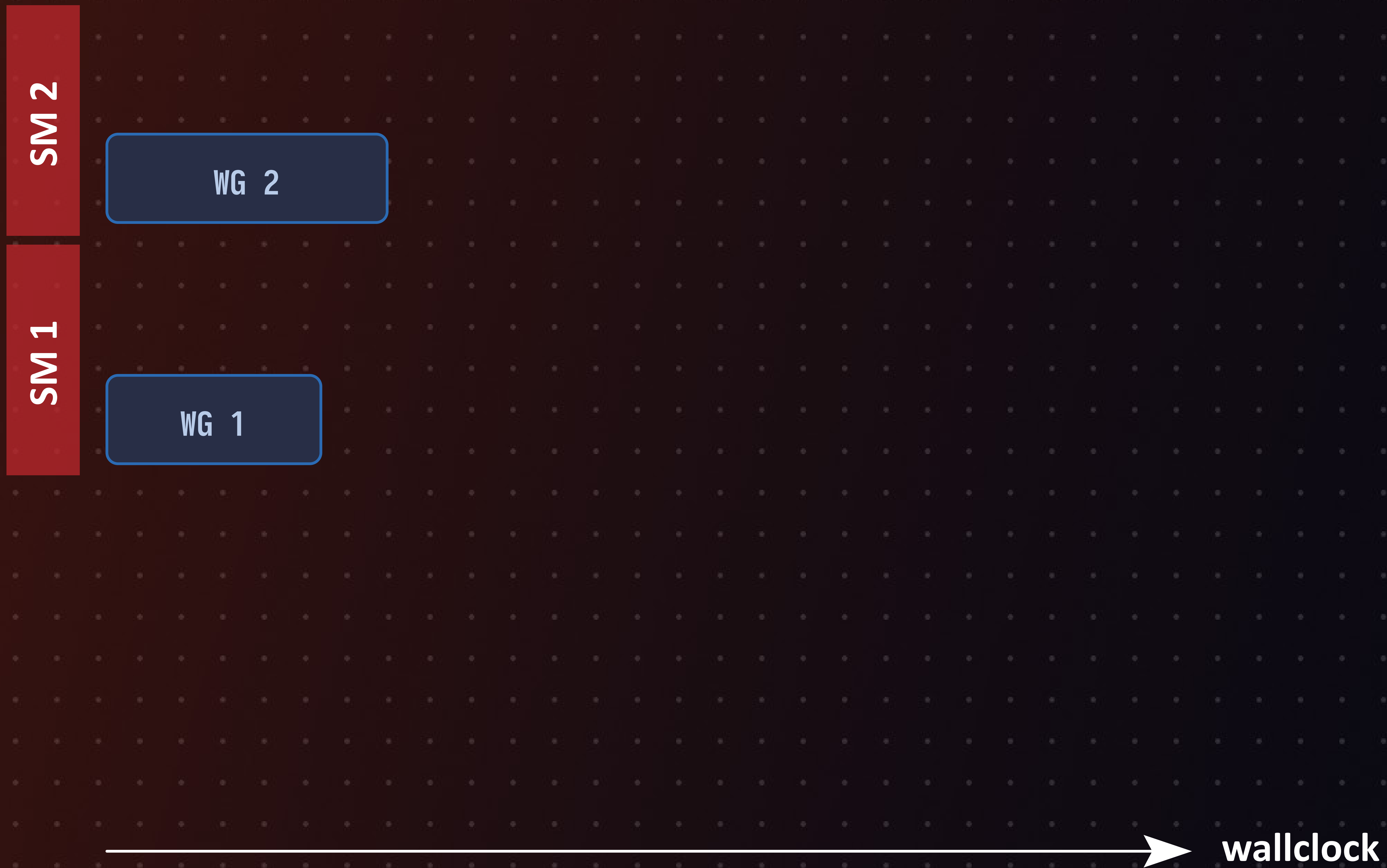
Feedback Loop:

- remove or add tuning parameters, or even whole algorithms
- reparameterize tuning space

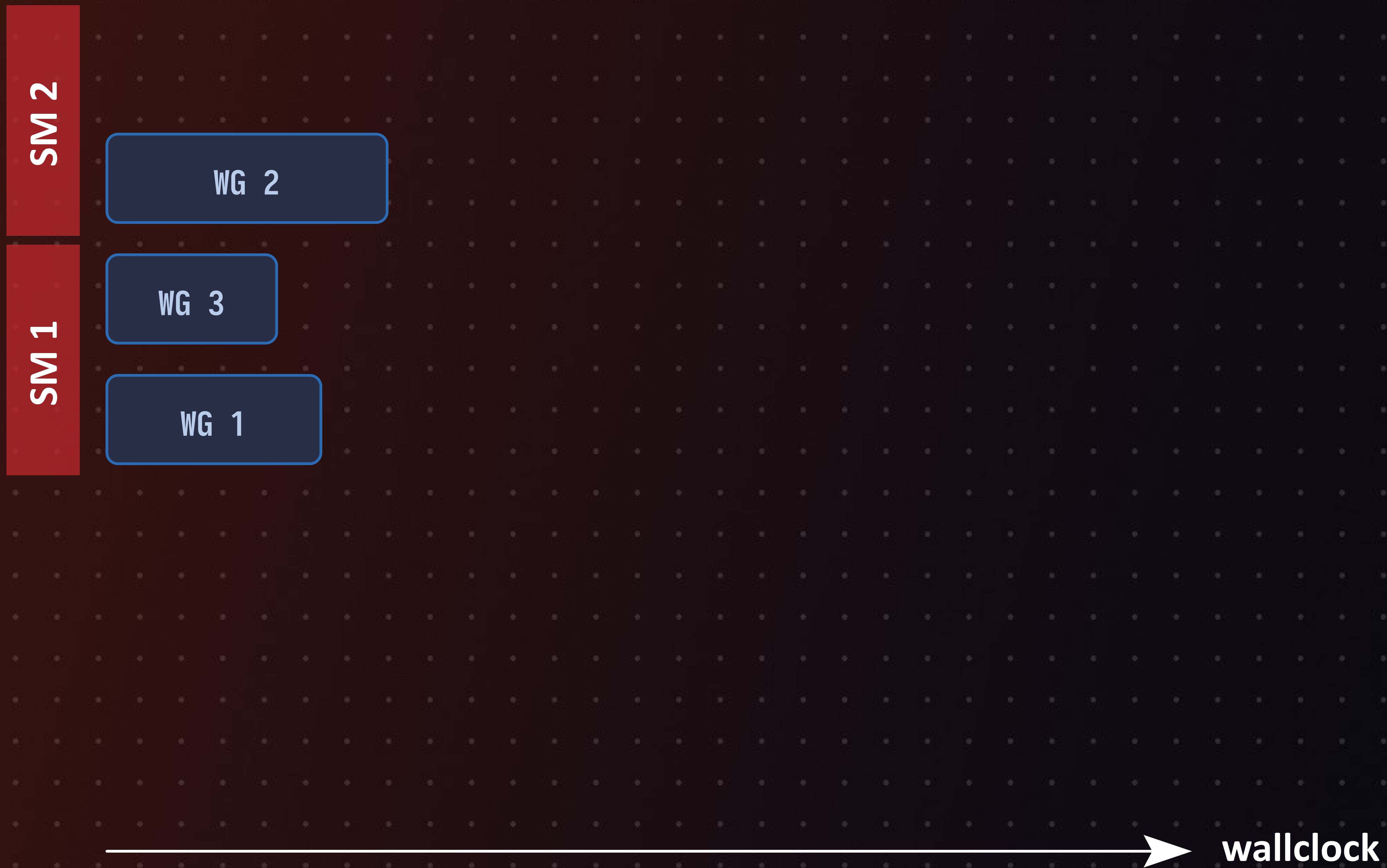
GPU — Programming Model vs Execution



GPU — Programming Model vs Execution



GPU — Programming Model vs Execution



GPU — Programming Model vs Execution

wallclock

GPU — Programming Model vs Execution

wallclock

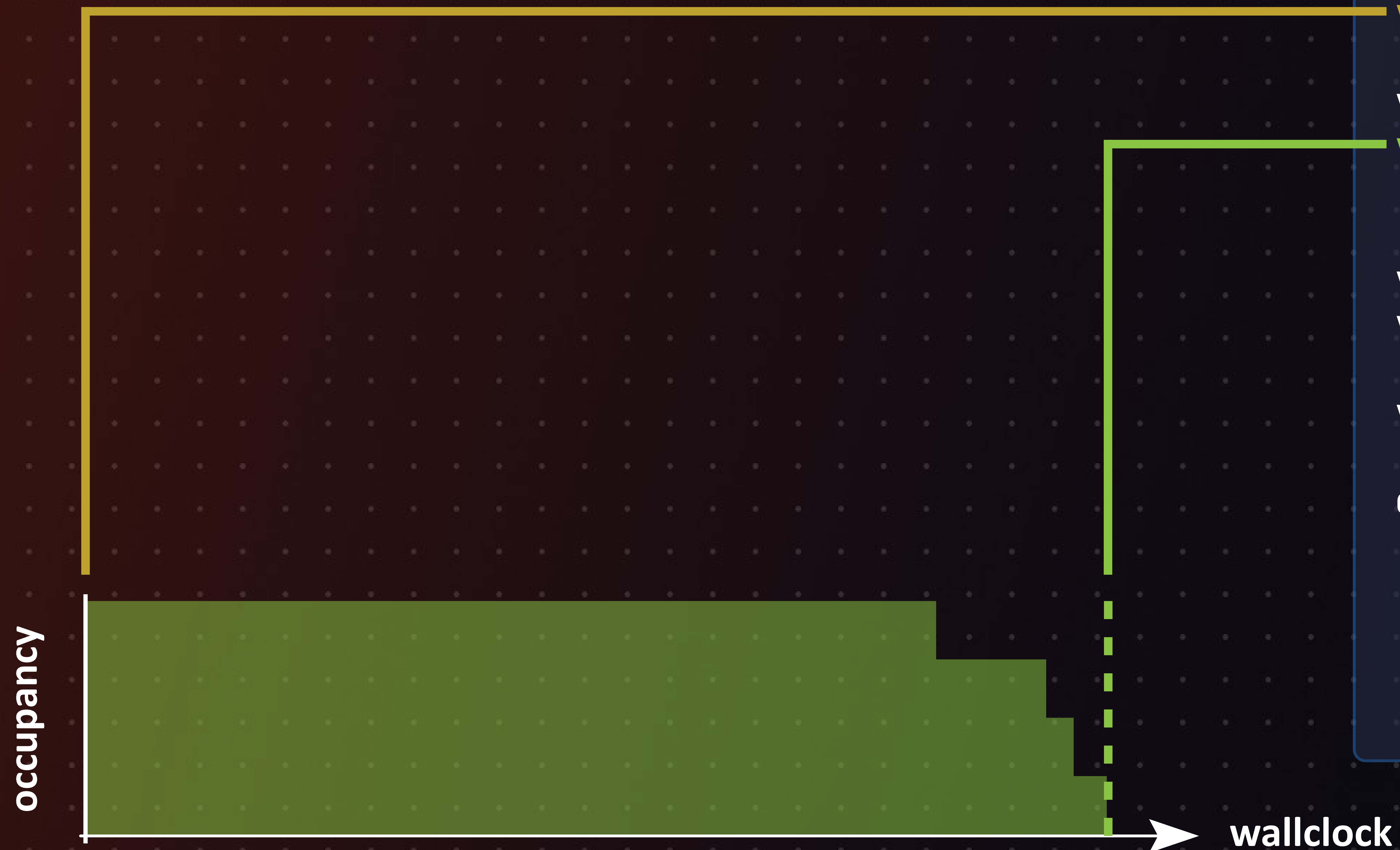
GPU — Programming Model vs Execution

GPU — Programming Model vs Execution

wallclock

GPU — Programming Model vs Execution

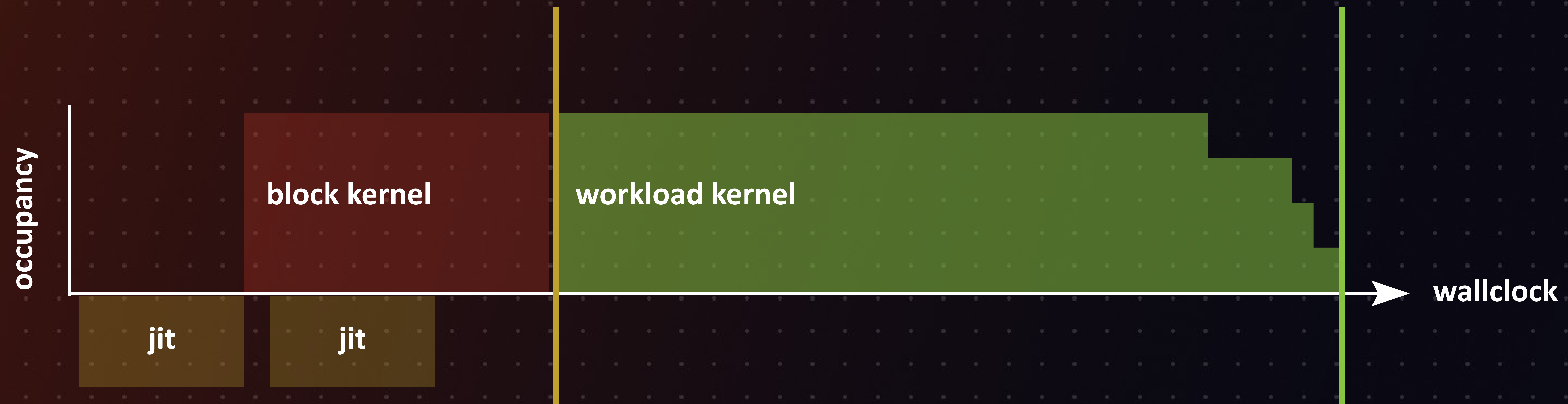
Benchmarking



C++

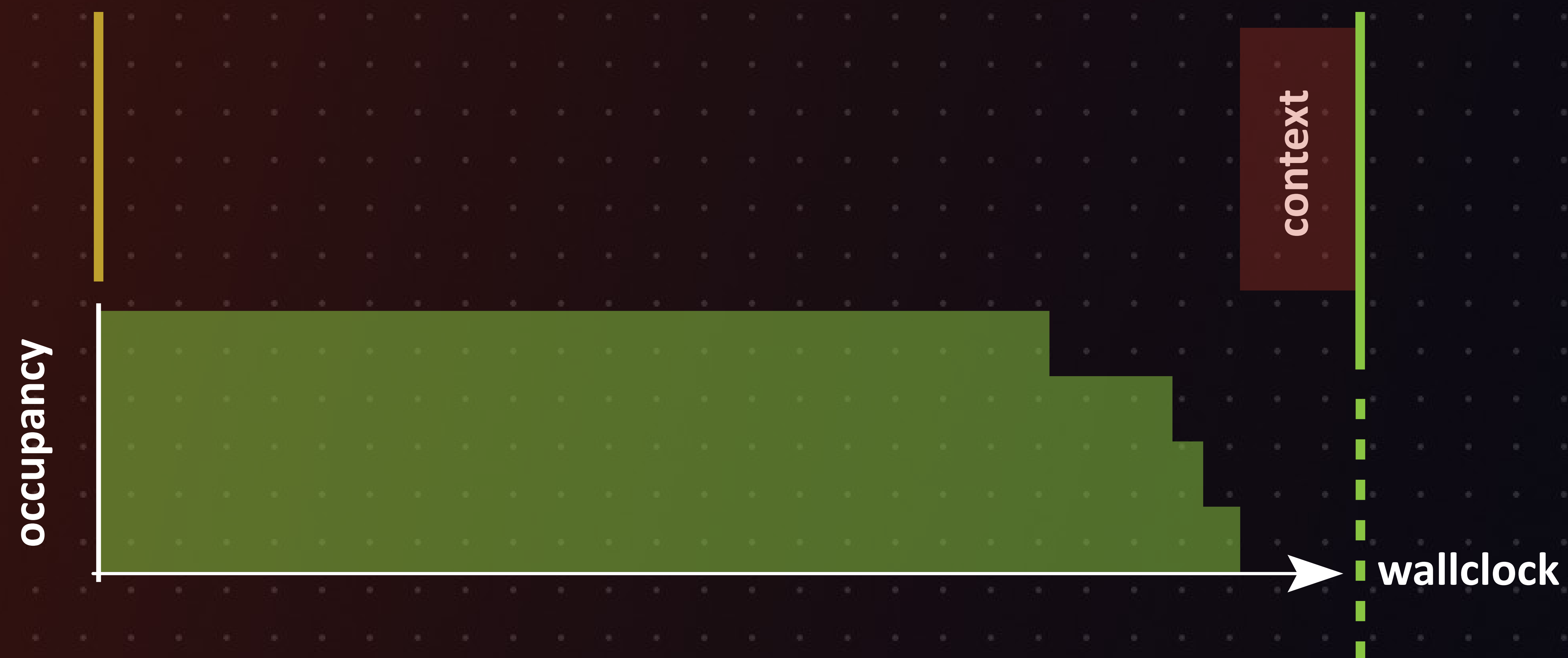
```cpp
// pipeline creation, descriptor writes, ...

vkBeginCommandBuffer(cmd,begin_info);

vkCmdWriteTimestamp(cmd, VK_PIPELINE_STAGE_ALL_COMMANDS_BIT,
                    query_pool, 0);
vkCmdDispatch(/*...*/);
vkCmdWriteTimestamp(cmd, VK_PIPELINE_STAGE_ALL_COMMANDS_BIT,
                    query_pool, 1);

vkEndCommandBuffer(cmd);
vkQueueSubmit(/*...*/);

vkGetQueryPoolResults(timestamps, /*...*/
     VK_QUERY_RESULT_WAIT_BIT);
double delta_in_ns =
    double(timestamps[1] - timestamps[0])
    * device_limits.timestampPeriod;
```

Benchmarking — of CUDA Kernels [2]



Benchmarking — Reproducability

- **background activity:** context switch included in measured interval
- **clock frequency:** lock clock to thermal design power (TDP) using vendor API



console window

```
[INFO] 9056 / 36864 (24.566%)
[INFO] benchmark histogram::gmem<GRAIN_SIZE=32u,groupSizeX=1024u>...
[INFO] validation OK
```

Benchmarking — In-Shader Timings (KHR_shader_clock)

- `uint64_t clockRealtimeEXT(void)` device consistent clock
- `uint64_t clockARB(void)` subgroup consistent clock
- unit of time is not defined, monotone unless it wraps
- get timer period by calibrating clockRealtimeEXT against vkCmdWriteTimestamp, not necessarily equal to VkPhysicalDeviceLimits::timestampPeriod!

NVIDIA Microbenchmarks

```
clockRealtimeEXT // nanosecond timer, %globaltimer
clockARB // clock cycle counter, clock64()
```

AMD Disassembly + ISA Docs

```
S_MEMREALTIME // 25MHz or 100MHz clock
S_MEMTIME // clock counter
S_SENDMSG_RTN_B64 S[2:3] REALTIME // 100MHz
```

Benchmarking — In-Shader Timings (`KHR_shader_clock`)

- `uint64_t clockRealtimeEXT(void)` device consistent clock
- `uint64_t clockARB(void)` subgroup consistent clock
- unit of time is not defined, monotone unless it wraps
- get timer period by calibrating clockRealtimeEXT against vkCmdWriteTimestamp, not necessarily equal to VkPhysicalDeviceLimits::timestampPeriod!

clockrate.glsl

```
sample[i].wallclock = clockRealtimeEXT();
sample[i].cycles = clockARB()
```

nanosleep.glsl

```
uint64_t time_end = clockRealtimeEXT();
while(time_end < calibrated_target_time) {
   time_end = clockRealtimeEXT();
   // detect and handle wrapping
}
```

Occupancy Estimation — Calculator Application

Portable GPU Occupancy Calculator Estimates Properties of your GPU Through Microbenchmarks

Shader Parameters

Threads Per Workgroup: 128

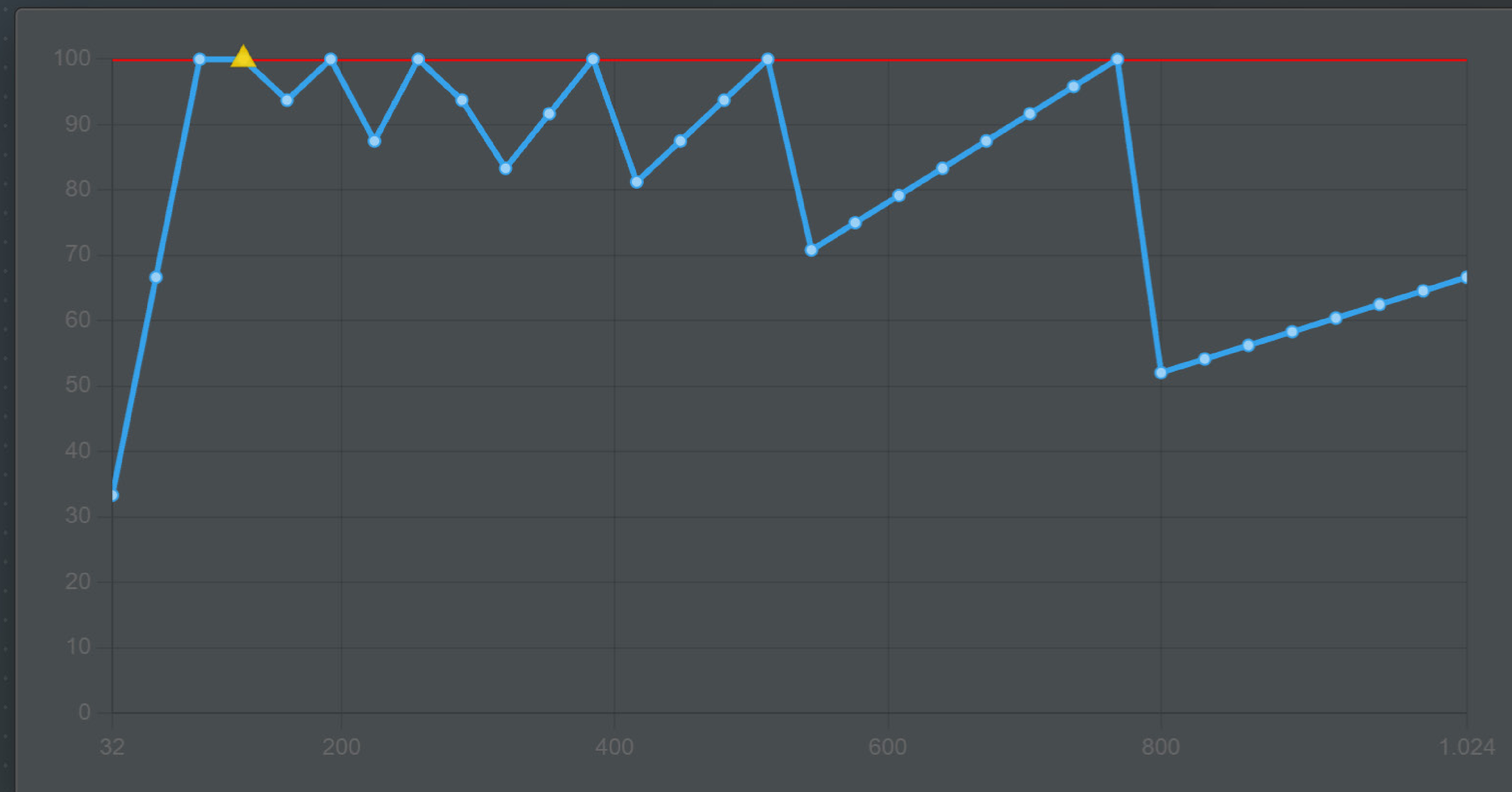
Shared Memory Per Workgroup: 0

Estimated Device Properties

Estimated SM Count	48
Estimated Maximum of Resident Workgroups Per SM	16
Estimated Maximum of Resident Subgroups Per SM	48
Estimated Maximum of Resident Threads Per SM	1536

Impact of Varying Workgroup Size

Download As: Select Format... ▼ Show As: Occupancy [%] ▼



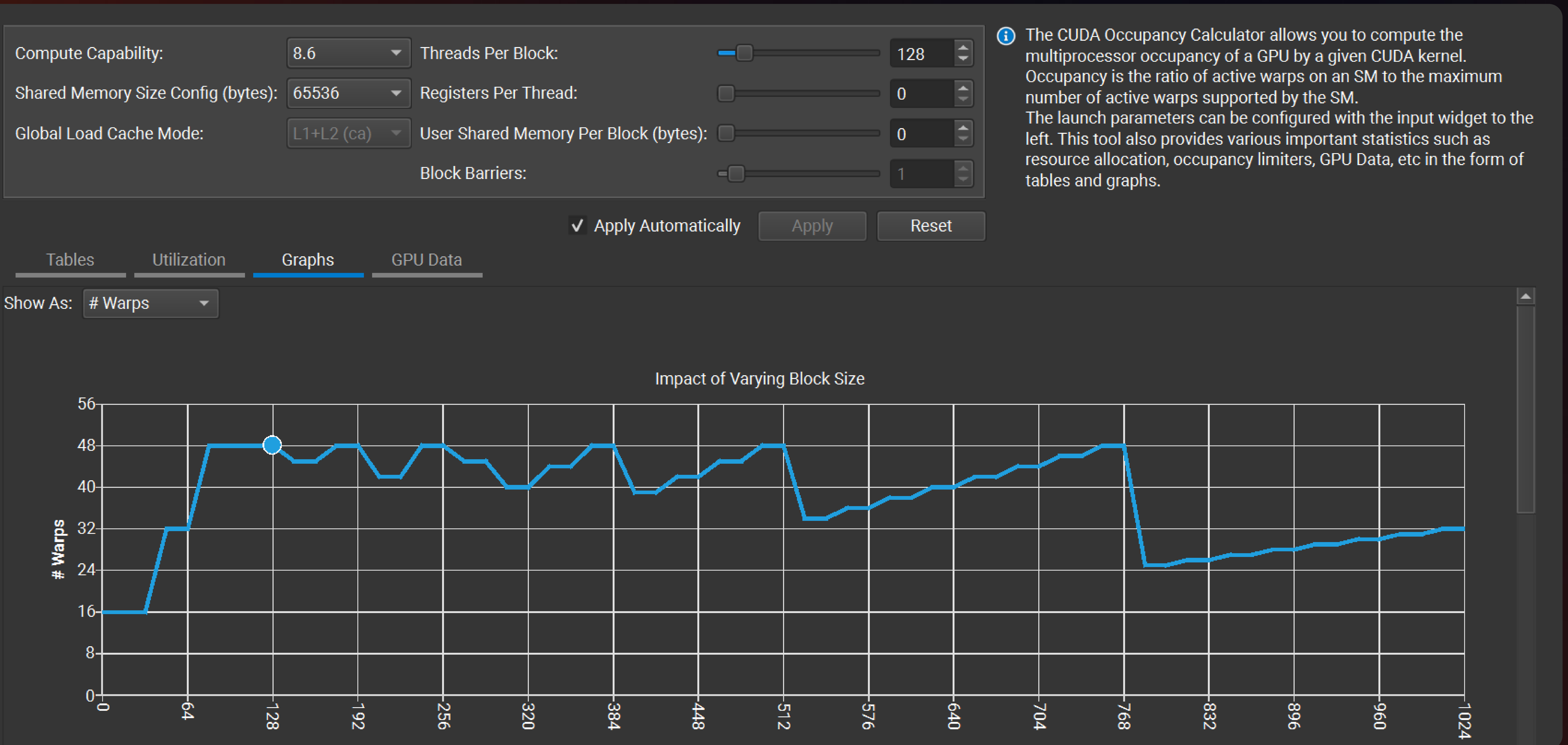
Tabular Occupancy Listing

Download As: Select Format... ▼

WG Size	Shared Memory [Bytes]	Occupancy [%]	WGs	Subgroups	Threads
32	0	33.33%	768	768	24576
64	0	66.67%	768	1536	49152
96	0	100.00%	768	2304	73728
128	0	100.00%	576	2304	73728
160	0	93.75%	432	2160	69120
192	0	100.00%	384	2304	73728
224	0	87.50%	288	2016	64512
256	0	100.00%	288	2304	73728
288	0	93.75%	240	2160	69120
320	0	83.33%	192	1920	61440

Expand

Occupancy Estimation — Calculator Application



Screenshot from Nvidia Nsight Compute

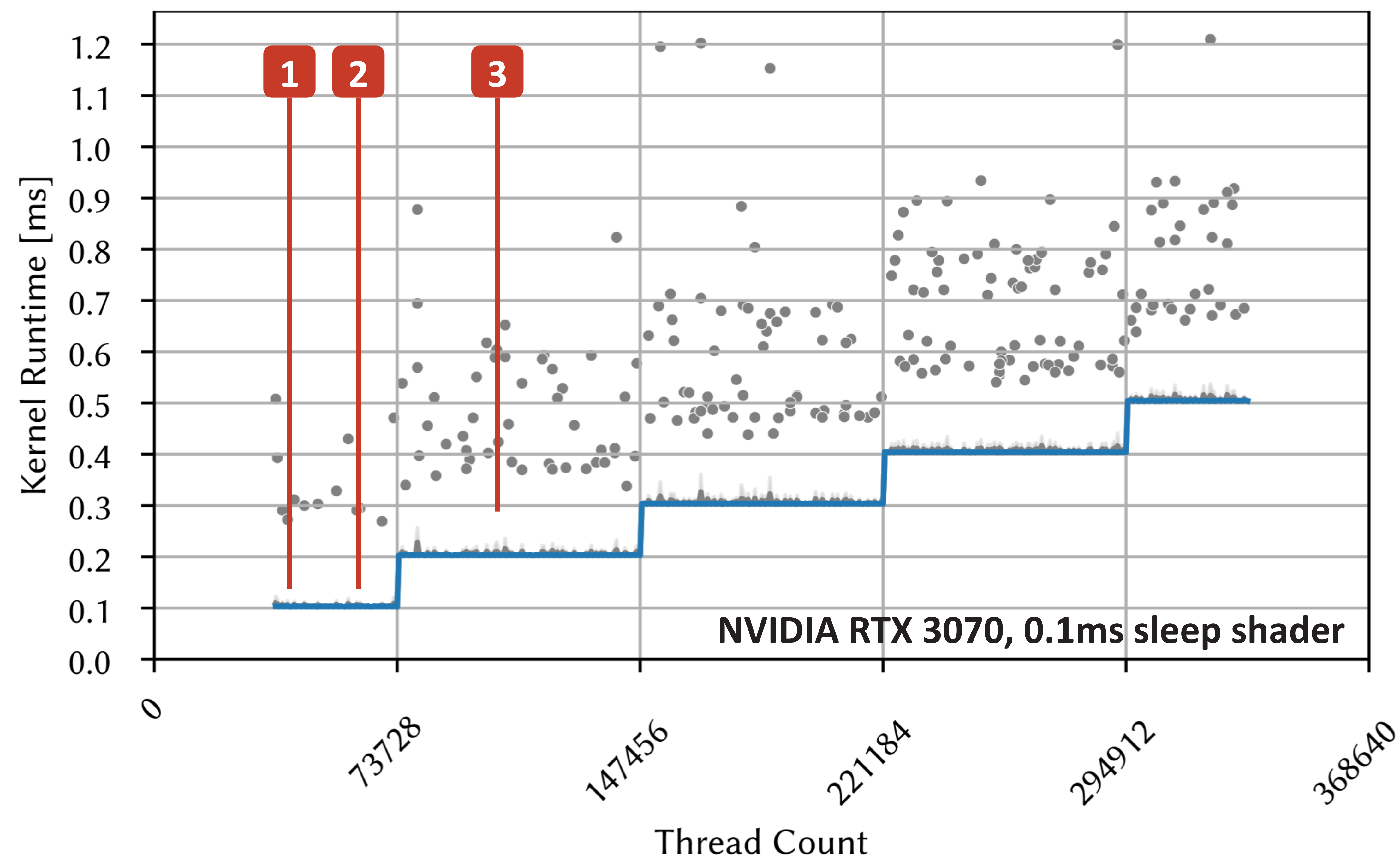
GPU — Execution of Regular Work

GPU — Execution of Regular Work

GPU — Execution of Regular Work

Occupancy Estimation — vkCmdWriteTimestamp

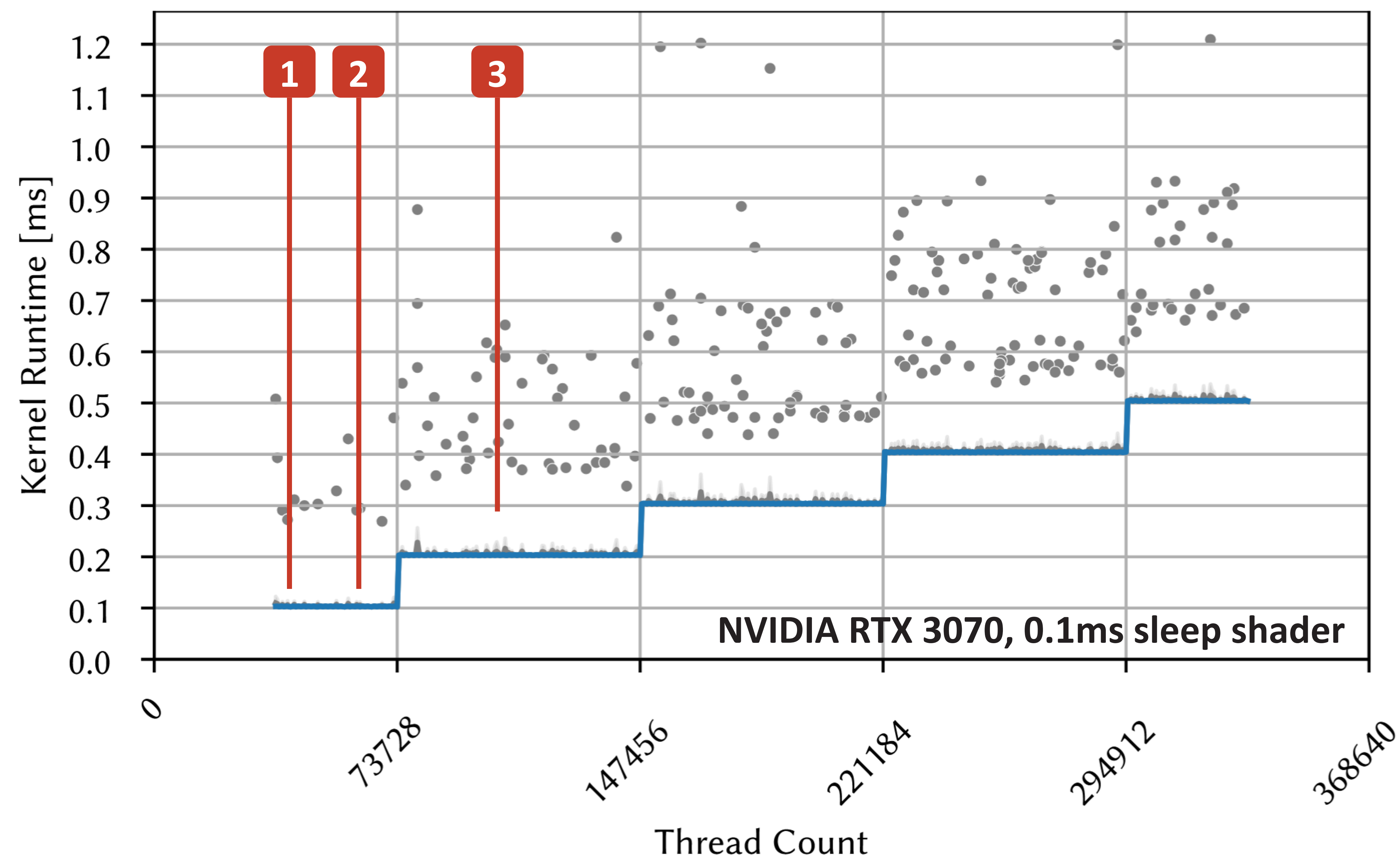
- Run benchmarks with a regular workload while varying the dispatch size to locate occupancy ***P***



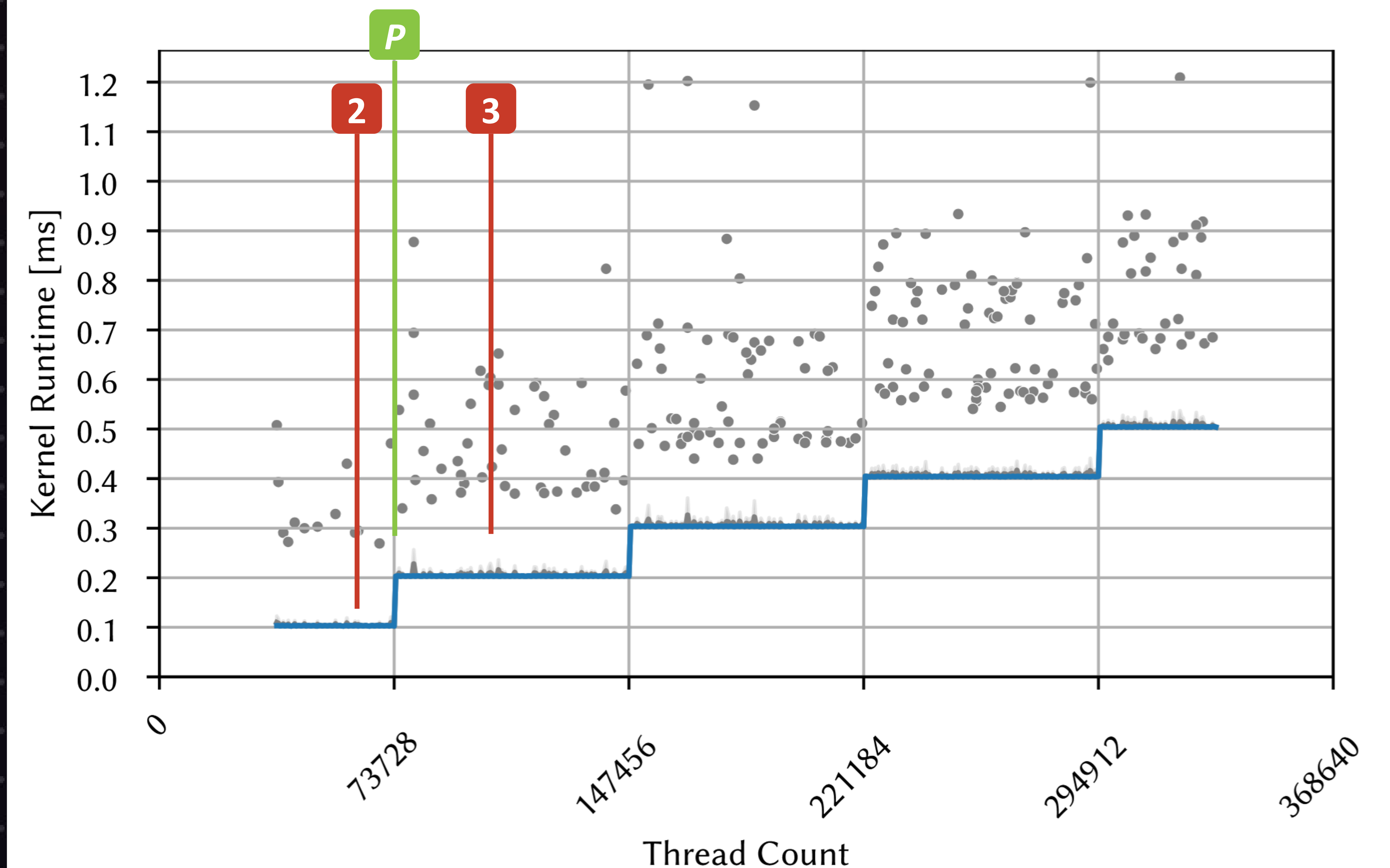
Occupancy Estimation — vkCmdWriteTimestamp

- Run benchmarks with a regular workload while varying the dispatch size to locate occupancy ***P***

step size doubling until we detect serialization



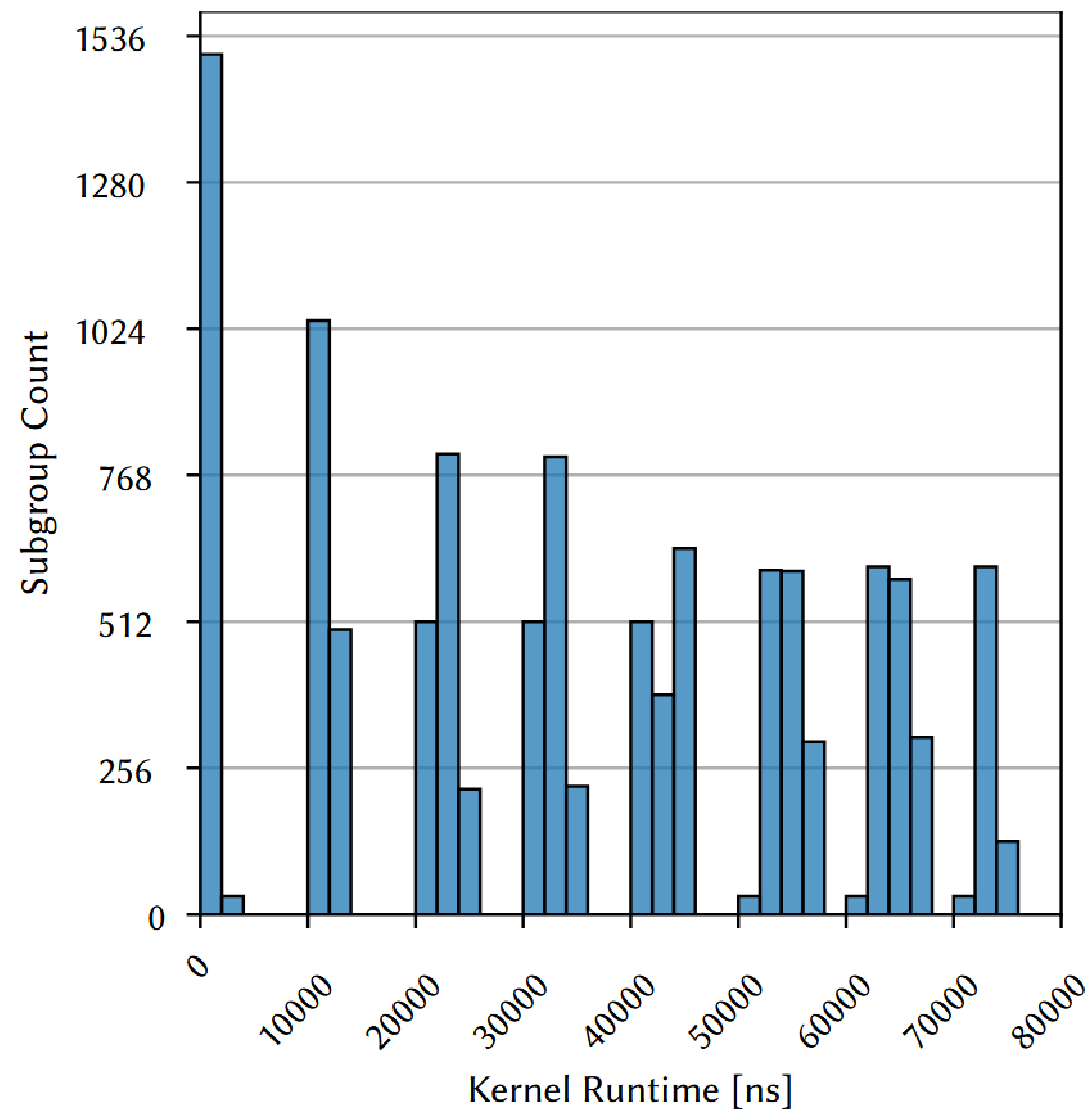
bisect to locate the first step



Occupancy Estimation — In-shader Clock Algorithm

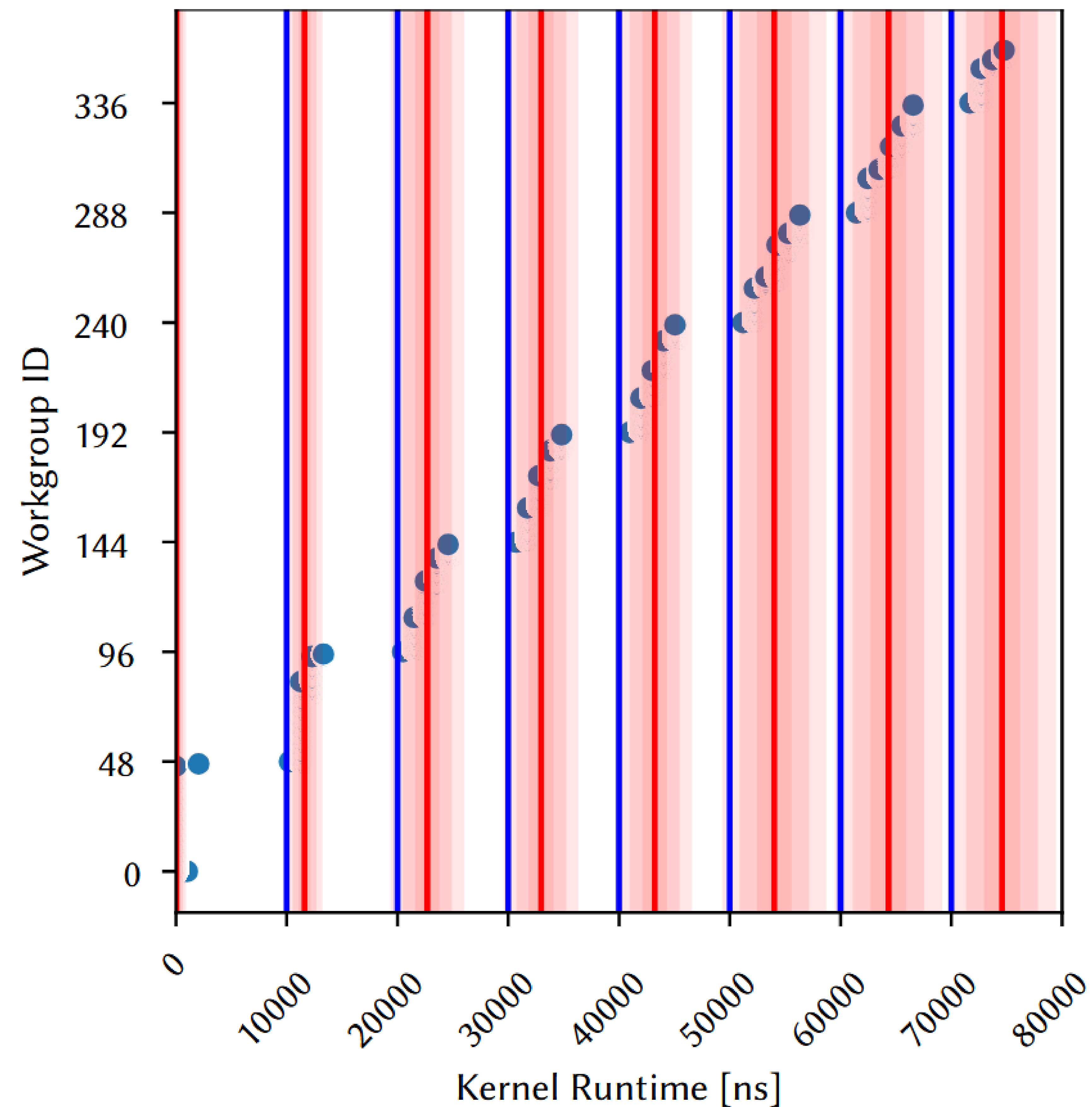
- Each workgroup records its start time with the device consistent clock before sleeping.
- On idealized hardware we would expect a signal comb with magnitude ***P***

Occupancy Estimation — In-shader Clock Algorithm



- Each workgroup records its start time with the device consistent clock before sleeping.
- On idealized hardware we would expect a signal comb with magnitude ***P***

Occupancy Estimation — In-shader Clock Algorithm



- Each workgroup records its start time with the device consistent clock before sleeping.
- On idealized hardware we would expect a signal comb with magnitude ***P***

Occupancy Estimation — Shader Augmentation

- Conditionally branch into estimator in a way that is opaque to the shader compiler
- *Actual shader code* is not executed, but reserves occupancy limitting resources (registers, shared memory, …)

GLSL

```glsl
// naive estimator
void estimator() {
  if(gl_LocalInvocationIndex == 0) {
      nanosleep(100ms);
  } barrier();
}

void main() {
  if(pc.run_estimator) {
     estimator();
  } else {
     // actual shader code reserves
     // occupancy limiting resources
  }
}
```

Emulation — Device-wide Barrier [9]

- Counting spin-lock waiting on ***P = occupant workgroups*** to arrive
- Usage Example: Device-cooperative acceleration structure traversal of a single “ray”. Alternate traversal and scheduling phase in a single shader. 5.7x speedup over [6].

SINGLE-USE BARRIER — GLSL

```glsl
void device_barrier() {
  if(lid == 0) {
     atomicAdd(&barrier, 1);
     while(atomicAdd(&barrier,0) < P){}
  }
  barrier();
}
```

MULTI-USE, SELF RESETTING BARRIER — GLSL

```glsl
void device_barrier() {
  if(lid == 0) {
  target = atomicAdd(&barrier, gid == 0 ?
                    P2-P : 1) & P2MSK + P2;
  while(atomicAdd(&barrier,0) & P2MSK !=
        target){} barrier();
}}
```

Reducing Tuning Parameters — Scan with Ringbuffers

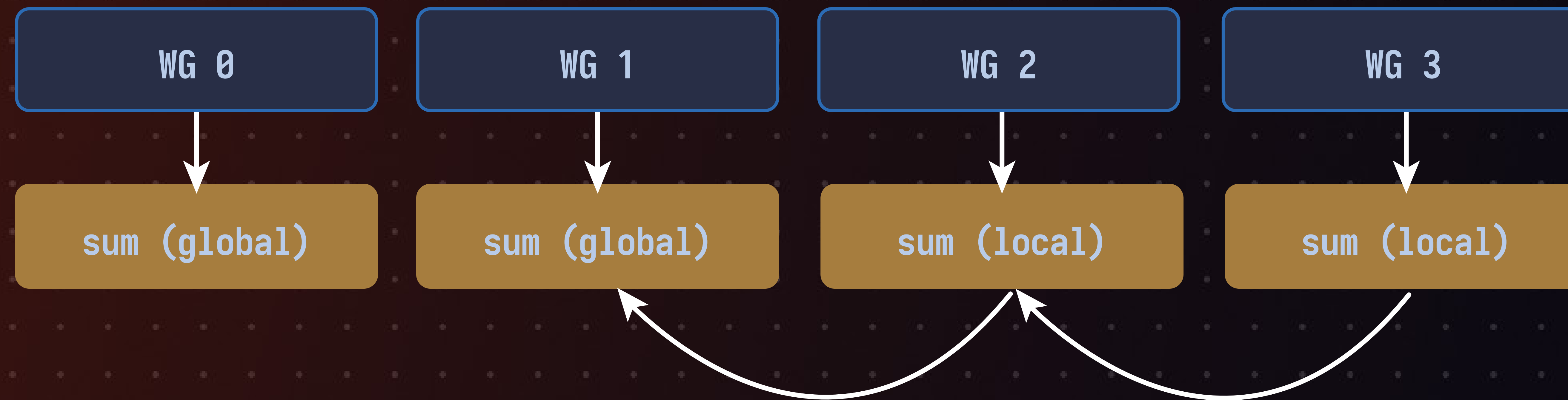
- Decoupled Lookback [4,3,5] allocates scratch memory per workgroup for communication

Decoupled Lookback — C++

```cpp
vkCmdFillBuffer(cmd, scratch, 0,
groupCountX * sizeof(uint32_t), 0);

// ...

vkCmdDispatch(cmd,groupCountX,1,1);
```

Reducing Tuning Parameters — Scan with Ringbuffers

- Decoupled Lookback [4,3,5] allocates scratch memory per workgroup for communication



Decoupled Lookback C++

```cpp
vkCmdFillBuffer(cmd, scratch, 0,
groupCountX * sizeof(uint32_t), 0);

// ...

vkCmdDispatch(cmd,groupCountX,1,1);
```

Reducing Tuning Parameters — Scan with Ringbuffers

- Bounded Decoupled Lookback [2] uses a ringbuffer of constant size instead.
- **Problem:** What value should each of the 3 ring buffer parameters be set to?
- Define based on occupancy ***P***, remove parameters!

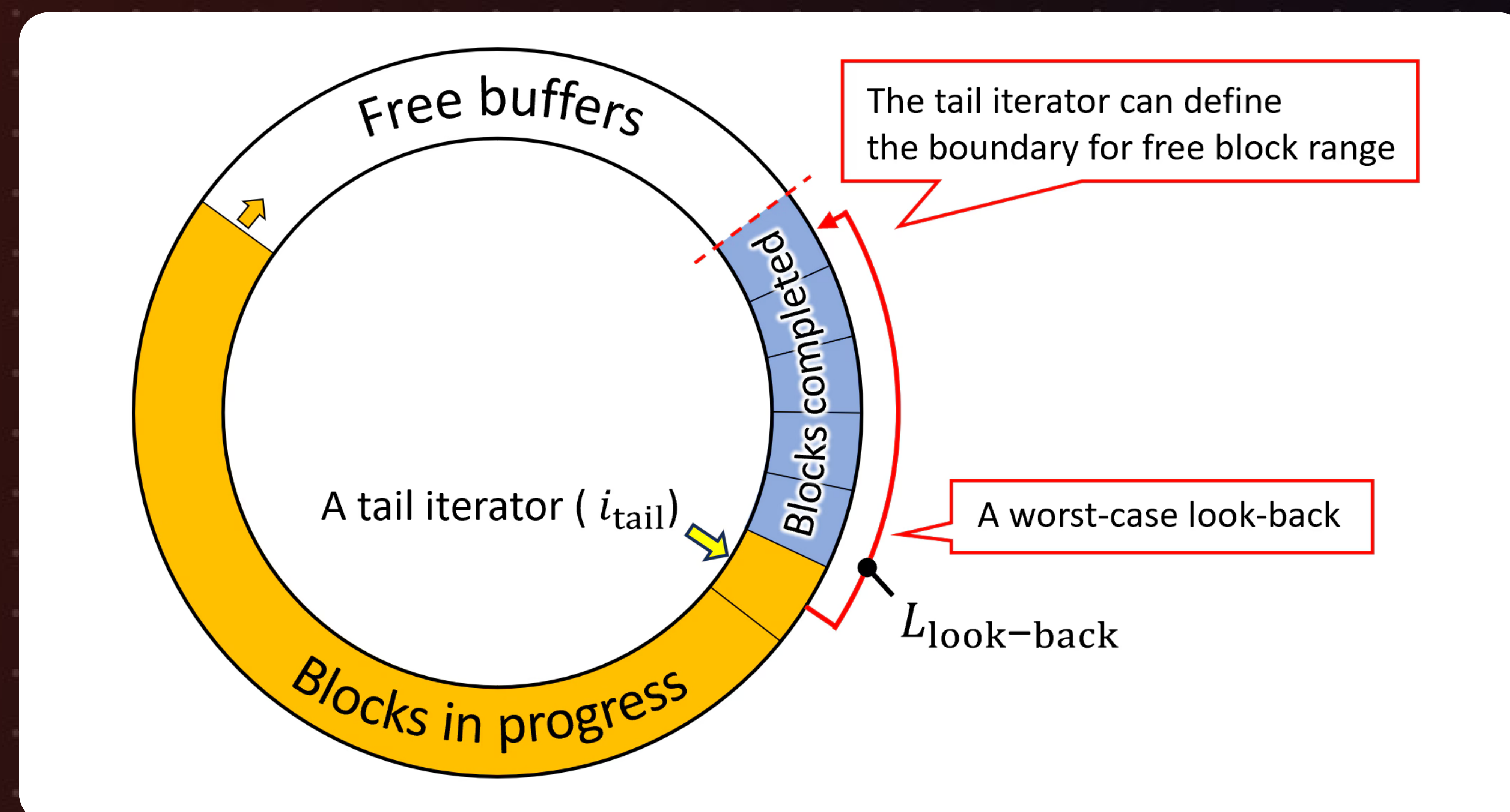


Figure source [2]

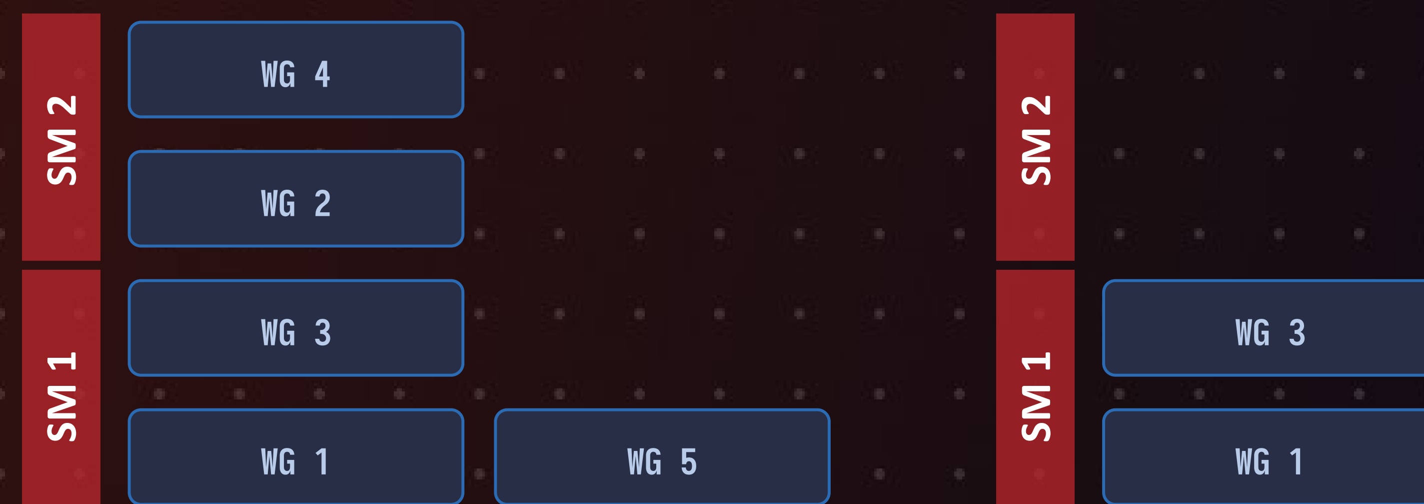
Decoupled Lookback with Ringbuffer C++

```cpp
if(groupCountXPrev < ringbufferSize*2) {
  vkCmdFillBuffer(cmd, scratch, 0,
  ringbufferSize * sizeof(uint64_t), 0);
}
// ...

vkCmdDispatch(cmd,groupCountX,1,1);
```

Reducing Tuning Parameters — Histogram with Grid Strided Loop and Persistent Threads

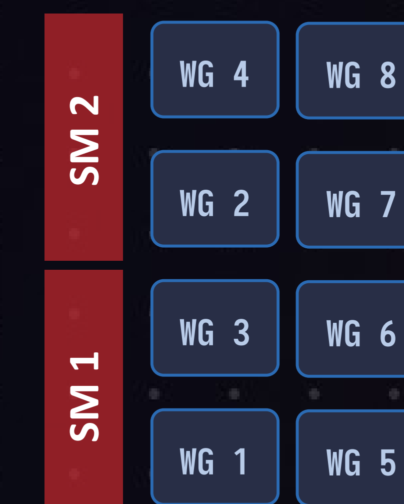
- Instead of large tile sizes, use grid-strided loops [11,12] or persistent threads [10]
- Both are efficient if the group count is >= occupancy ***P***
- Establishes a devirtualized programming model parameterized by ***P*** and load balancing chances ***C***

(a) too aggressive setting of TILES_PER_WG



(b) persistent thread count > P

(c) grid-strided loop with C=2



Policy Selection — GPU Similarity

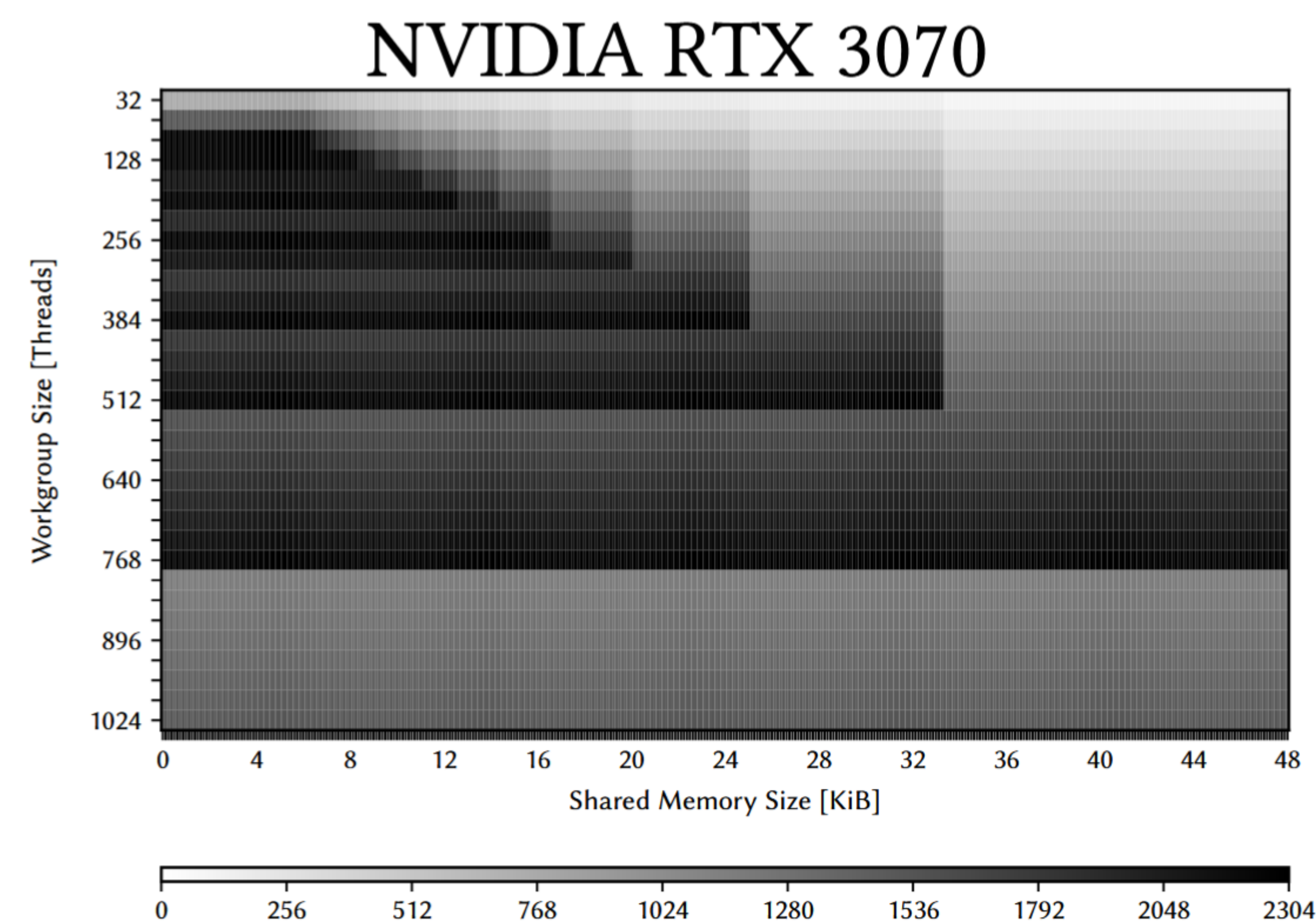
- Key contains argument list and environment (capabilities, features, limits, ...)
- Challenging if constraint not explicitly modeled.
- `VkPhysicalDeviceProperties::{vendorID, deviceID}`
- Performance similarity with a 'Compute Capability' like a not yet existing '`VkPhysicalDeviceProperties::architectureID`'?

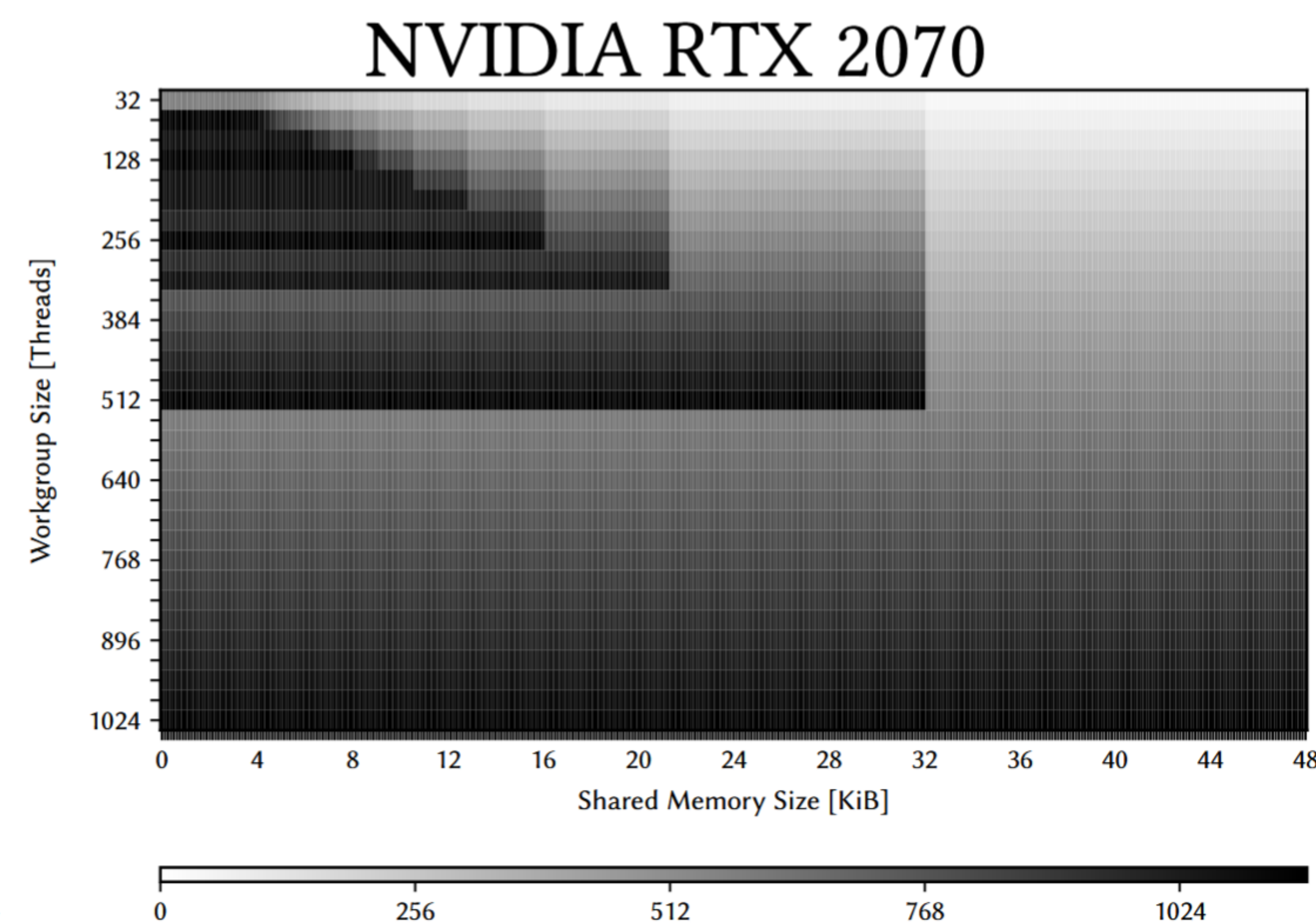
wine/dlls/win32u/sysparams.c

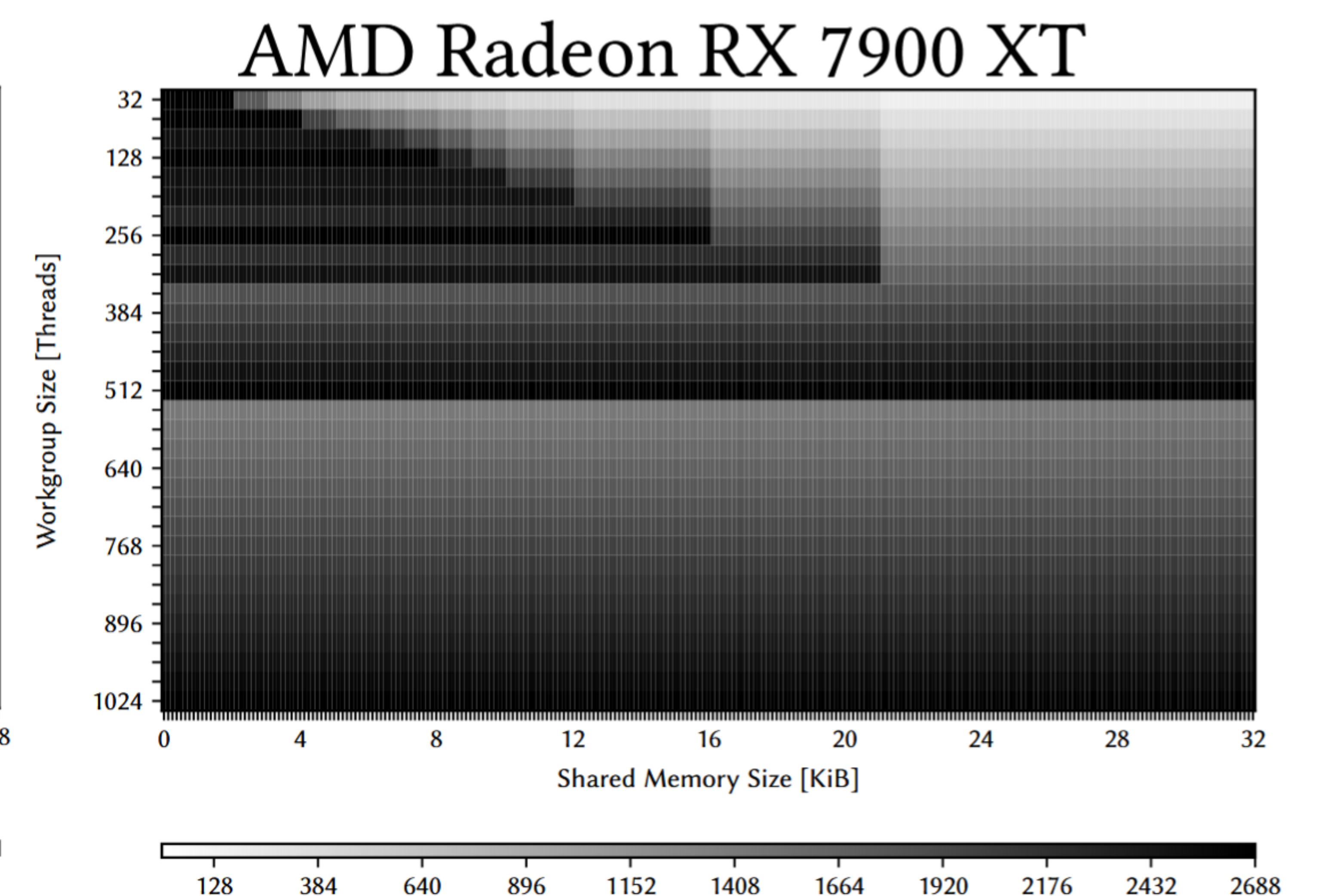
```
static void fixup_device_id( const struct pci_id **pci_id )
{
    static struct pci_id fake_id;
    const char *sgi;

    if ((*pci_id)->vendor == 0x10de /* NVIDIA */
        && (sgi = getenv("WINE_HIDE_NVIDIA_GPU")) && *sgi != '0')
    {
        fake_id.vendor = 0x1002; /* AMD */
        fake_id.device = 0x73df; /* RX 6700XT */
        *pci_id = &fake_id;
    }
```

Policy Selection — GPU Similarity

- Instead of database mapping devices to an architecture, use normalized occupancy surface to fingerprint the multiprocessor

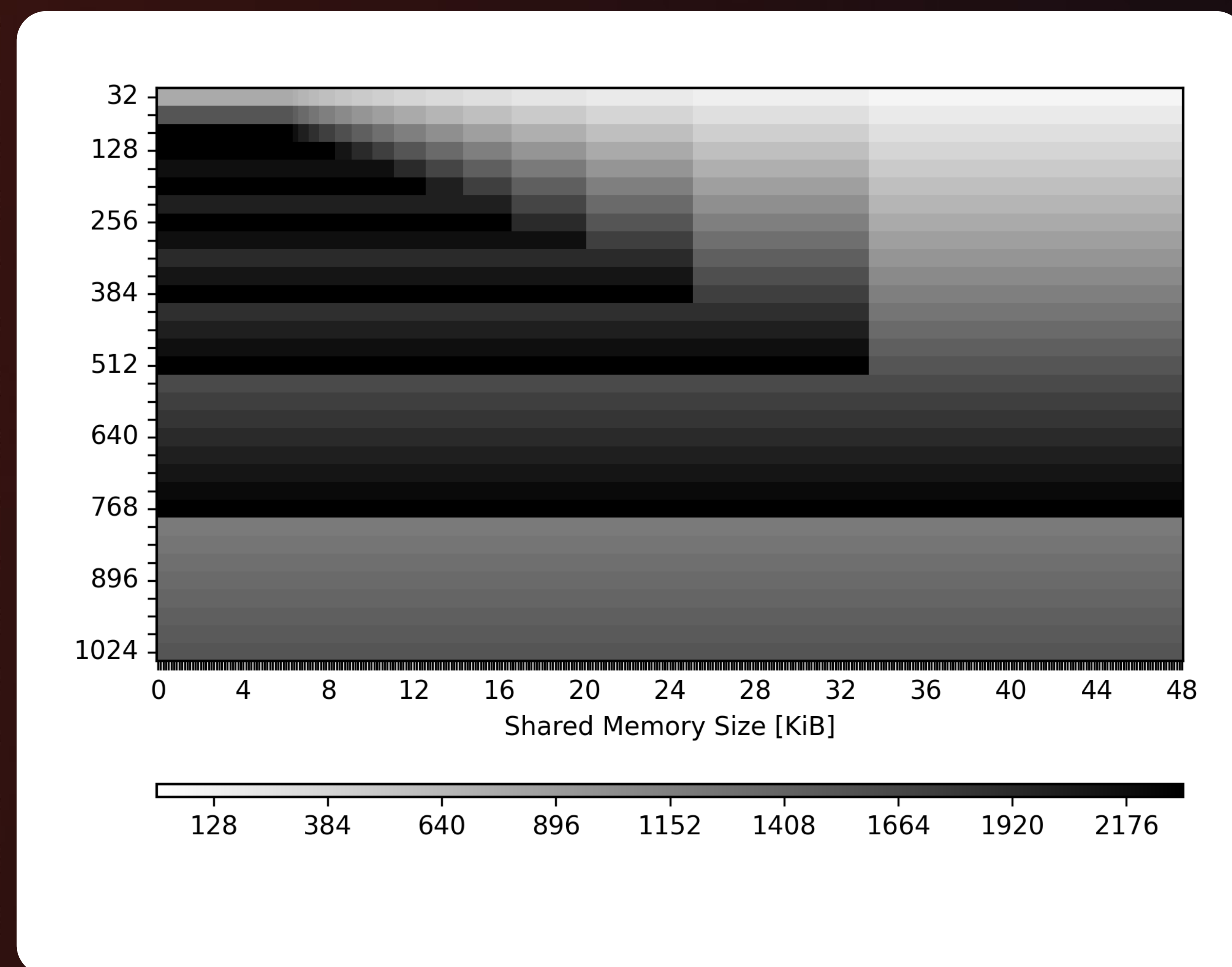






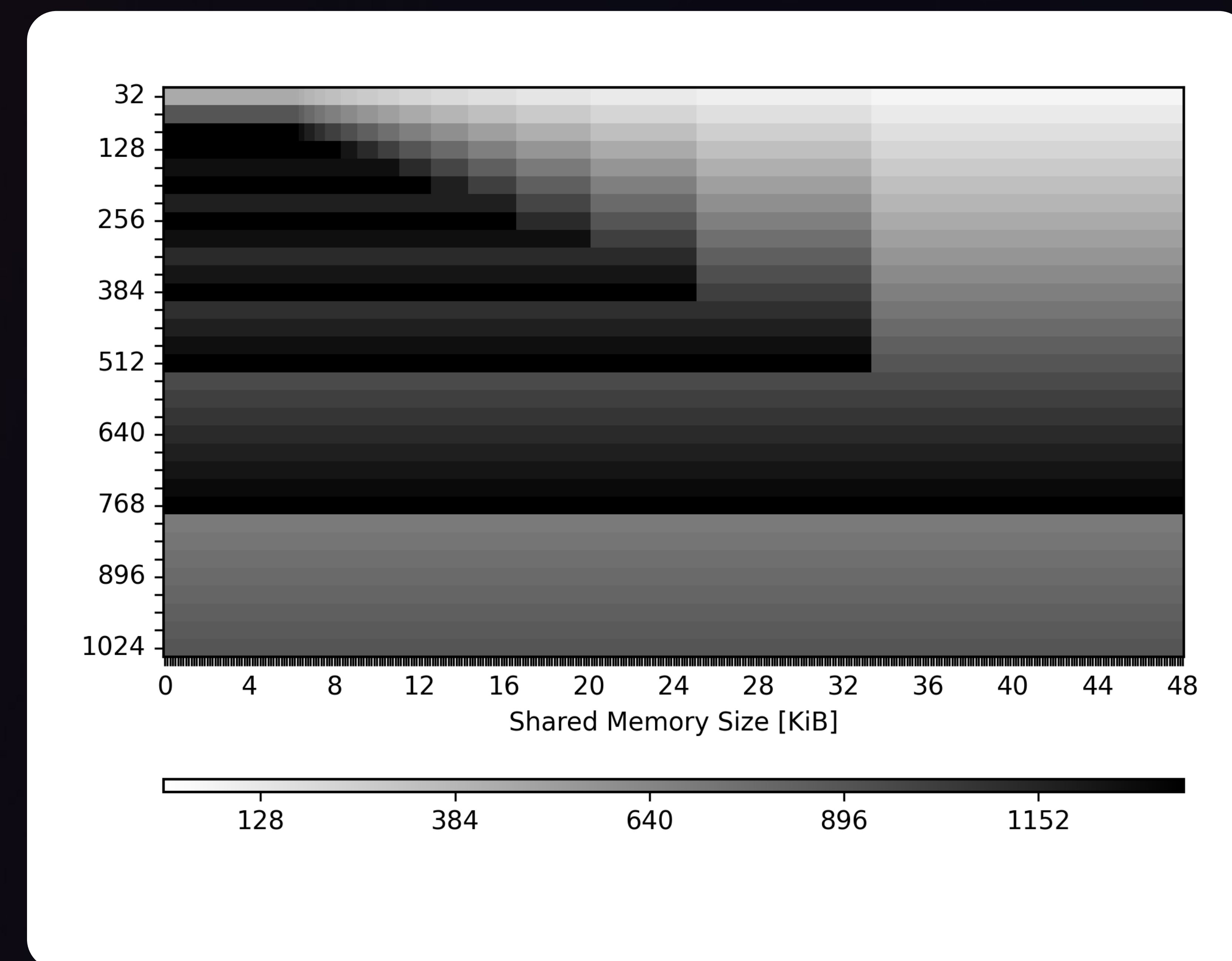
Policy Selection — GPU Similarity

- Instead of database mapping devices to an architecture, use normalized occupancy surface to fingerprint the multiprocessor



NVIDIA RTX 3070
Estimated SMs=48

/ 48 =



/ 28

NVIDIA RTX 3060
Estimated SMs=28

Conclusion — Takeaways

- **Benchmarking** The Vulkan API design avoids many pitfalls, be aware of others (e.g. clock rates, interval tightness,...)

    Missing APIs for clock control

    In-shader benchmarking with clock extensions. Specification language is too general for our purposes

- **Devirtualized Programming Patterns** (using occupancy ***P***) allow many optimizations

    Vulkan is missing extensions to expose this information

    We showed that careful benchmarking allows side-channeling of occupancy P as a workaround

- **Auto Tuning** is massively important for peak performance, hard challenge for portability

    Vulkan is missing extensions to expose this information, e.g.
    Missing API to group GPUs into architecture clusters with similar performance characteristics

References & Further Reading

Autotuning

[1] Merrill et al., 2012, “Policy-based tuning for performance portability and library co-optimization”

Algorithms

[2] Kao & Yoshimura, 2025, “Boosting GPU Radix Sort performance: A memory-efficient extension to Onesweep with circular buffers”

[3] Adinets & Merrill, 2022, “Onesweep: A faster least significant digit radix sort for GPUs”

[4] Merrill & Garland, 2016, “Single-pass Parallel Prefix Scan with Decoupled Look-back”

[5] Smith et al., 2025, “Decoupled Fallback: A Portable Single-Pass GPU Scan”

References & Further Reading

[6] Fan et al., 2024, “gDist: Efficient Distance Computation between 3D Meshes on GPU”

Benchmarking and Profiling

[7] Evtushenko, 2025, “Kernel Benchmarking Tales”
https://www.youtube.com/watch?v=CtrqBmYtSEk

[8] Bavoil, 2019, “The Peak-Performance-Percentage Analysis Method for Optimizing Any GPU Workload”
https://developer.nvidia.com/blog/the-peak-performance-analysis-method-for-optimizing-any-gpu-workload/

References & Further Reading

Feature Emulation, Software Load Balancing & Programming Models

[9] Sorensen et al., 2016, “Portable inter-workgroup barrier synchronisation for GPUs”

[10] Aila & Lane, 2009, “Understanding the Efficiency of Ray Traversal on GPUs”

[11] Flegar & Anzt, 2017, “Overcoming load imbalance for irregular sparse matrices”

[12] Harris, 2013, “CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops“
https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/