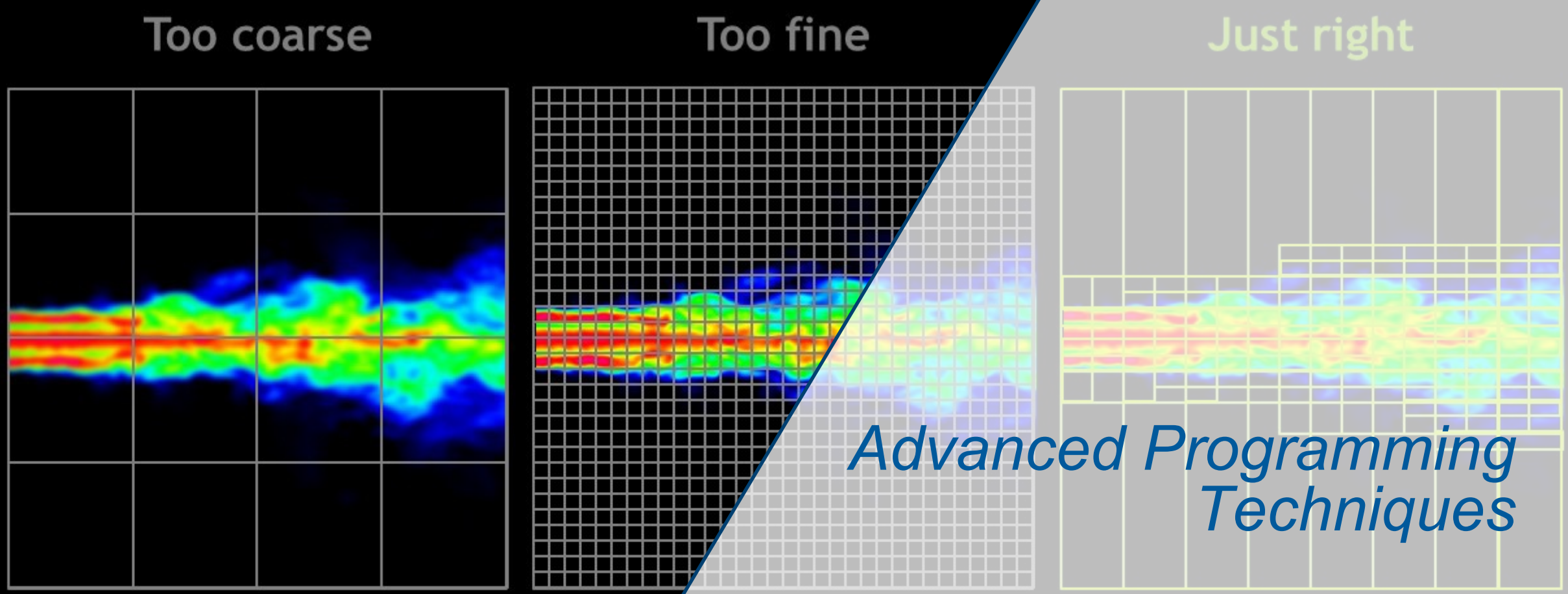



# Advanced Programming Techniques

24.05.23

# *Performance Guidelines*

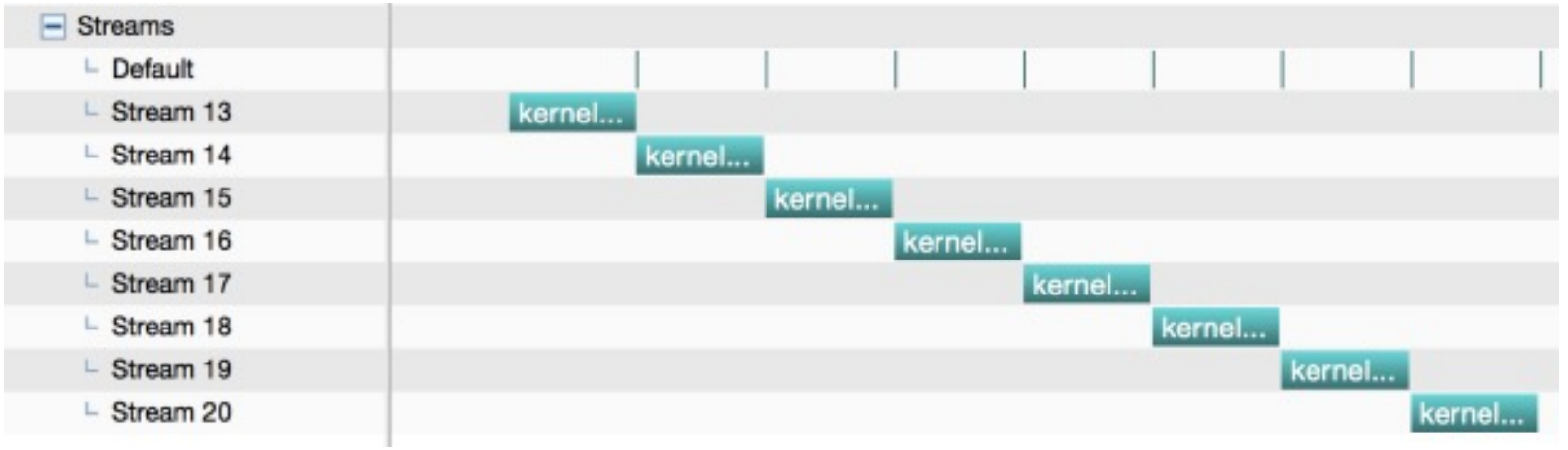
- Maximize parallel execution to achieve maximum **utilization**
- Optimize memory usage to achieve maximum **memory throughput**
- Optimize instruction usage to achieve maximum **instruction throughput**

# *Maximize Utilization*

- Structure application to **expose as much parallelism as possible**
- Application level
  - Asynchronous calls and streams
  - Sequential work on CPU, parallel work on GPU
  - Minimize global synchronization
  - Prefer inter-thread communication within thread block
- Device level
  - Maximize parallel execution between SMs
  - Use streams to execute multiple kernels concurrently

# Maximize Utilization

- Streams are queues with independent tasks the GPU scheduler can select tasks from
  - Overlapped execution of
    - Independent kernels
    - Data transfers




# *Maximize Utilization*

- Parallel execution between various functional units within an SM
- Next instruction can be independent instruction of same warp (ILP) or of another warp (more common)
- To hide latency of **L** cycles
  - Need 4L instructions (for 5.x/6.1/7.x)
  - SM issues 1 instruction per warp over one cycle for 4 warps
- Warp not ready if inputs operands are not available yet
  - Register dependencies (some input not done, …)
- Warp waits on memory fence or synchronization point
  - Multiple resident blocks can help reduce idling
- Number of registers used is significant factor

# *Maximize Memory Throughput*

- Minimize data transfers with low bandwidth
  - Maximize use of on-chip memory
  - Can make sense to execute code with less parallelism on GPU to reduce memory transfers
- Memory access pattern is important
  - Different types for different types of memory
  - Global memory
    - 32-/64-/128-byte segments, naturally aligned, transactions coalesced within warp
    - If size and alignment is not 1, 2, 4, 8, or 16 bytes → cannot coalesce
    - Allocated memory always aligned to 256 bytes
  - Constant memory
    - Only for uniform reads of the same address, otherwise serialized

# *Maximize Memory Throughput*

- Memory access pattern is important
  - …
  - Textures
    - Optimized for 2D spatial locality, addressing calculations are done by dedicated units, conversion for free
  - Local memory
    - Organized as consecutive 32-bit words → fully coalesced if all threads in warp access same relative address
  - Shared memory
    - Avoid bank conflicts

# *Maximize Instruction Throughput*

- Minimize use of instructions with low throughput
  - Prefer intrinsics to regular functions
  - Single-precision instead of double-precision
- Minimize divergent warps
  - Rewrite conditions to minimize intra-warp divergence
- Reduce number of instructions by optimizing out synchronization points

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions

# *Instruction Optimization*

- Prefer **bitwise** operations over **integer division** and **modulo** (for powers of 2)
  - Compiler will do this if **n** is **literal**
- Use **signed** integers for **loop counters**
  - Compiler can ignore overflow semantics and optimize more
- Avoid automatic conversion of double to float
  - Use floating point values with suffix ***f***

- Avoid different execution paths within the same warp
  - Compiler may use predication to avoid short branch
    - Predication: All instructions scheduled, per-thread condition decides execution
    - Possible if number of instructions per branch is low
- Make it easier for compiler to use predication
  - Use ***#pragma unroll*** on loops if possible
  - Small loops with known count are unrolled automatically

# Pointer Aliasing

- Two pointers **alias** if the memory to which they point overlaps
  - Compiler cannot make assumptions here
    - Must reload ***c***

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

# Pointer Aliasing

```
__global__
void example_aliasing(float* a, float* b, float* c) {
        3                1                2
    a[threadIdx.x] = a[threadIdx.x] + c[threadIdx.x];
    b[threadIdx.x] = b[threadIdx.x] + c[threadIdx.x];
        6                5                4
}
```

c is read twice

(because writing to a could have changed its value)

https://godbolt.org/z/7z9zTz8r1

```
example_alias(float*, float*, float*):
MOV R1, c[0x0][0x28]
NOP
S2R R6, SR_TID.X
MOV R7, 0x4
IMAD.WIDE.U32 R2, R6, R7, c[0x0][0x160]
IMAD.WIDE.U32 R4, R6, R7, c[0x0][0x170]
read a  LDG.E.SYS R0, [R2]
read c  LDG.E.SYS R9, [R4]
IMAD.WIDE.U32 R6, R6, R7, c[0x0][0x168]
FADD R9, R0, R9
write a STG.E.SYS [R2], R9
read c  LDG.E.SYS R11, [R4]
read b  LDG.E.SYS R0, [R6]
FADD R11, R0, R11
write b STG.E.SYS [R6], R11
EXIT
.L_2:
BRA `(.L_2)
.L_21:
```

# Pointer Aliasing

```
__global__
void example_aliasing(float* __restrict a, float* __restrict b, float* __restrict c) {
        3                 1                 2
    a[threadIdx.x] = a[threadIdx.x] + c[threadIdx.x];
    b[threadIdx.x] = b[threadIdx.x] + c[threadIdx.x];
        5                 4
}
```

c is read only once

```
example_alias(float*, float*, float*):
MOV R1, c[0x0][0x28]
NOP
S2R R6, SR_TID.X
MOV R7, 0x4
IMAD.WIDE.U32 R4, R6, R7, c[0x0][0x170]
IMAD.WIDE.U32 R2, R6, R7, c[0x0][0x160]
LDG.E.CONSTANT.SYS R5, [R4]          read a
LDG.E.SYS R0, [R2]                   read c
IMAD.WIDE.U32 R6, R6, R7, c[0x0][0x168]
FADD R9, R0, R5
STG.E.SYS [R2], R9                   write a
LDG.E.SYS R0, [R6]                   read b
FADD R11, R5, R0
STG.E.SYS [R6], R11                  write b
EXIT
.L_2:
BRA `(.L_2)
.L_21:
```

https://godbolt.org/z/58MY4z48s

# Pointer Aliasing

- `__restrict__` keyword (or `__restrict`)
  - Promise compiler that any data written to that pointer is not read by any other pointer with same keyword
  - Especially important on older architectures (e.g., Kepler)

```
__global__ void example_restrict_a(float* a, float* b, int* c) {
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    b[idx] = a[c[idx]];
}
__global__ void example_restrict_b(const float* __restrict__ a, float* __restrict__ b,
                                   const int* __restrict__ c) {
    int index = blockIdx.x*blockDim.x + threadIdx.x;
    b[idx] = a[c[idx]];
}
```

# Vectorized Memory Access

- Many kernels bandwidth bound
  - Ever copy operation consists of six steps
    - Compute load & store address (IMAD)
    - Load (LD) & store (ST)

```
__global__ void device_copy_scalar_kernel(int* in, int* out, int N) {
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    for (int i = idx; i < N; i += blockDim.x * gridDim.x)
        out[i] = in[i];
}
```

```
/*0070*/ MOV R5, 0x4
/*0080*/ IMAD.WIDE R2, R0, R5, c[0x0][0x160]
/*0090*/ LDG.E.SYS R3, [R2]
/*00a0*/ IMAD.WIDE R4, R0, R5, c[0x0][0x168]
         ...
/*00e0*/ STG.E.SYS [R4], R3
```

# Vectorized Memory Access

```
__global__ void example_aliasing(int* __restrict__ out, const int* __restrict__ in, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    for (int i = idx; i < N; i += blockDim.x * gridDim.x)
        out[i] = in[i];
}
```

```
copy(int*, int*, int):
MOV R1, c[0x0][0x28]
NOP
S2R R0, SR_CTAID.X
S2R R3, SR_TID.X
IMAD R0, R0, c[0x0][0x0], R3
ISETP.GE.AND P0, PT, R0, c[0x0][0x170], PT
@P0 EXIT
.L_2:
MOV R5, 0x4
IMAD.WIDE R2, R0, R5, c[0x0][0x160]
LDG.E.CONSTANT.SYS R3, [R2]
IMAD.WIDE R4, R0, R5, c[0x0][0x168]
MOV R7, c[0x0][0xc]
IMAD R0, R7, c[0x0][0x0], R0
ISETP.GE.AND P0, PT, R0, c[0x0][0x170], PT
STG.E.SYS [R4], R3
@!P0 BRA `(.L_2)
EXIT
.L_3:
BRA `(.L_3)
.L_24:
```

**address to load from**
**load**
**address to store to**
**store**

# Vectorized Memory Access

```
__global__ void example_aliasing(int4* __restrict__ out, const int4* __restrict__ in, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    for (int i = idx; i < N/4; i += blockDim.x * gridDim.x)
        out[i] = in[i];
}
```

```
copy(int4*, int4*, int):
IMAD.MOV.U32 R1, RZ, RZ, c[0x0][0x28]
NOP
S2R R3, SR_CTAID.X
IMAD.MOV.U32 R0, RZ, RZ, c[0x0][0x170]
S2R R4, SR_TID.X
SHF.R.S32.HI R0, RZ, 0x1f, R0
LEA.HI R2, R0, c[0x0][0x170], RZ, 0x2
SHF.R.S32.HI R9, RZ, 0x2, R2
IMAD R0, R3, c[0x0][0x0], R4
ISETP.GE.AND P0, PT, R0, R9, PT
@P0 EXIT
.L_2:
MOV R3, 0x10
IMAD.WIDE R4, R0, R3, c[0x0][0x160]
LDG.E.128.CONSTANT.SYS R4, [R4]
IMAD.WIDE R2, R0, R3, c[0x0][0x168]
MOV R11, c[0x0][0xc]
IMAD R0, R11, c[0x0][0x0], R0
ISETP.GE.AND P0, PT, R0, R9, PT
STG.E.128.SYS [R2], R4
@!P0 BRA `(.L_2)
```

**address to load from**
**load**
**address to store to**

**store**

# *Vectorized Memory Access*

- Improve performance by using vectorized loads and stores
  - Use vector data types (e.g., **int2, float4, uchar4,…**)
    - Require aligned data
  - Still requires 6 instructions
    - But loads 2x/4x more data
- Almost always preferable, but
  - Increases register pressure
  - Decreases parallelism

# Math API

- Integer intrinsic functions

```
// reverse the bit order of a 32-bit unsigned integer
__device__ unsigned int __brev(unsigned int x)

// return the number of consecutive high-order zero bits in a 32-bit integer
__device__ int __clz(int x)

// find the position of the least significant bit set to 1 in a 32-bit integer
__device__ int __ffs(int x)

// compute average of signed input arguments, avoiding overflow in the intermediate sum
__device__ int __hadd(int, int)

// count the number of bits that are set to 1 in a 32-bit integer
__device__ int __popc(unsigned int x)
```

# Launch Bounds

- Compiler tries to minimize register usage while keeping register spilling low
- Can aid compiler using **launch bounds**

```
__global__ void
__launch_bounds__(MaxThreadsPerBlock, MinBlocksPerMultiprocessor)
my_kernel(...) { ... }
```

- Compiler will try to use these as guidelines for register allocation

# Profiler Counter

- Each SM has 16 hardware counters for profiling
  - Can be incremented calling

```
void __prof_trigger(int counter);
```

- Counter 0-7 can be queried by **`nvprof`** and are automatically reset before each kernel launch

# Timing on the GPU

- It is possible to retrieve the clock counter of a SM
  - Stored in a special register

```
clock_t clock();
long long int clock64();
```

```
CS2R R2, SR_CLOCKLO
```

- Can be used to measure the number of clock cycles to execute device code
  - Timing of the SM, not of the instructions
  - Multiple warps can be active at the same time

# Performance Metrics

- CUDA provides calls to place into default stream
  - Avoid CPU timers

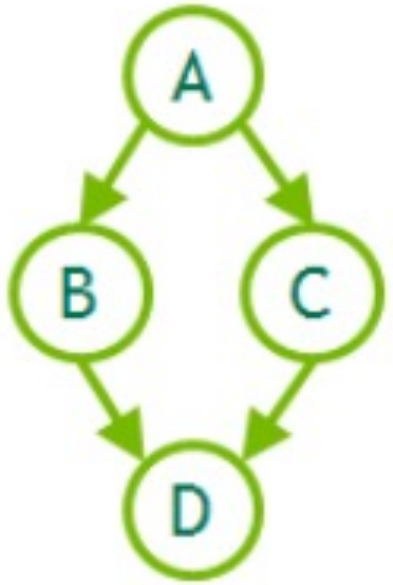
```
void inline start_clock(cudaEvent_t &start,
                        cudaEvent_t &end) {
    cudaEventCreate(&start);
    cudaEventCreate(&end);
    cudaEventRecord(start, 0);
}
```

```
float inline end_clock(cudaEvent_t &start,
                       cudaEvent_t &end) {
    float time;
    cudaEventRecord(end, 0);
    cudaEventSynchronize(end);
    cudaEventElapsedTime(&time, start, end);
    cudaEventDestroy(start);
    cudaEventDestroy(end);
    return time;
}
```

# CUDA Graphs

# CUDA Graphs

- Many (HPC) applications build on iterative structure
  - Work submitted for every iteration
    - Consumes time & resources
- CUDA graphs
  - Series of operations
    - Memory copies, kernel launches, …
  - Connected by dependencies
  - Defined separately from execution
    - Define once → run repeatedly



Workflow Graph

# Benefits

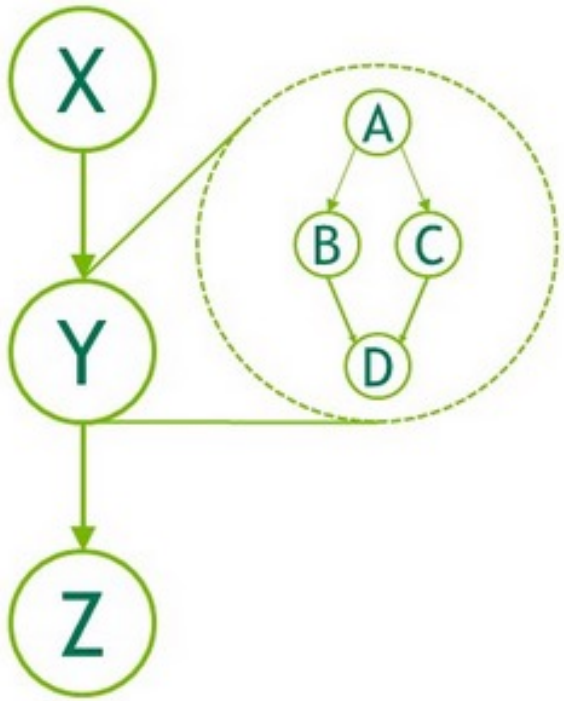
## CUDA Graphs

- Overhead of kernel launch can be significant
  - Separation of definition and execution reduces CPU overhead
- Optimizations
  - As whole workflow is visible, including
    - Execution
    - Data movement
    - Synchronization

# Node Types

## CUDA Graphs

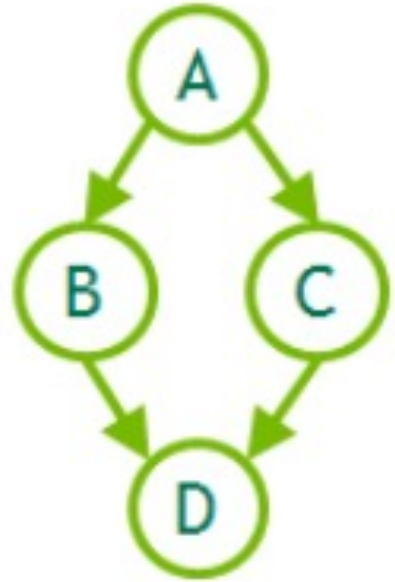
- Kernel
- CPU function call
- Memory copy
- Memset
- Empty Node
- Child graph



# Example

## CUDA Graphs

```
cudaGraph_t graph;
cudaGraphExec_t instance;

// define graph of work + dependencies
cudaGraphCreate(&graph);
cudaGraphAddNode(graph, kernel_a, {}, ...);
cudaGraphAddNode(graph, kernel_b, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_c, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_d, { kernel_b, kernel_c }, ...);

// instantiate graph and apply optimizations
cudaGraphInstantiate(&instance, graph);

// launch executable graph 100 times
for (int i = 0; i < 100; ++i)
    cudaGraphLaunch(instance, stream);
```

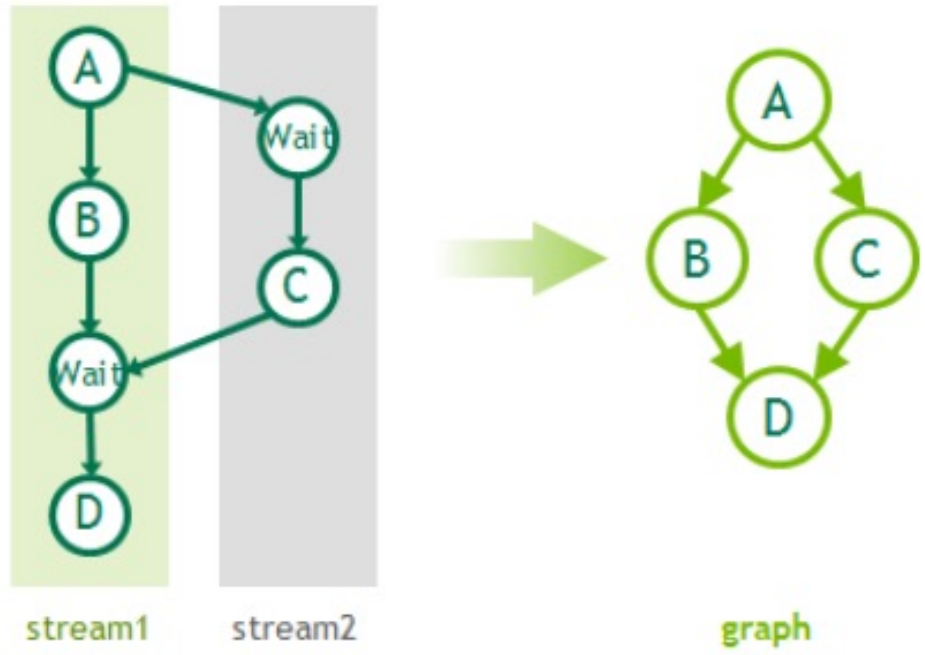



Workflow Graph

# Example

## CUDA Graphs

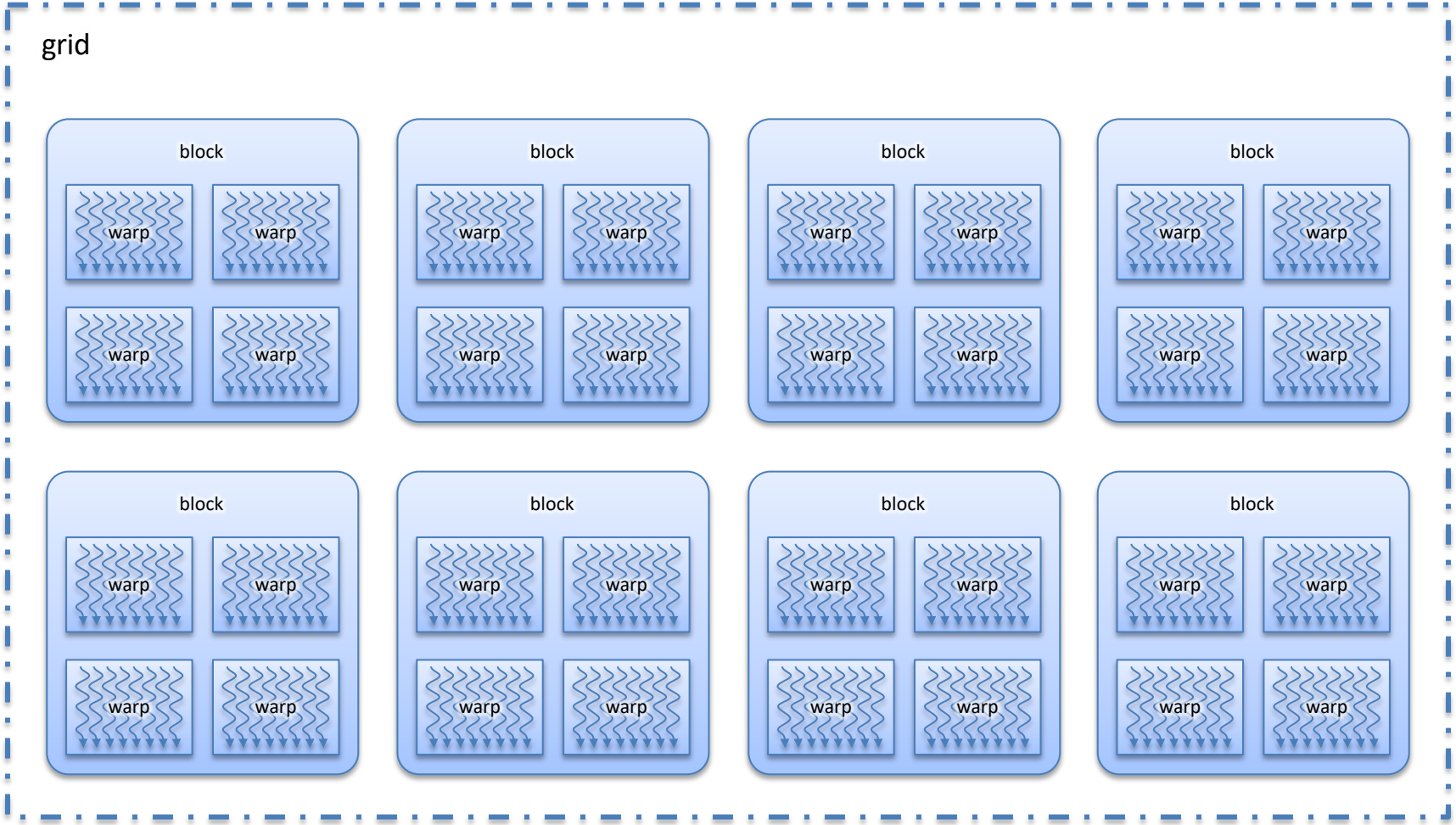
```
// start by initializing stream capture
cudaStreamBeginCapture(stream1);

// build stream work as usual
A<<< ..., stream1 >>>();
cudaEventRecord(e1, stream1);
B<<< ..., stream1 >>>();
cudaStreamWaitEvent(stream2, e1);
C<<< ..., stream2 >>>();
cudaEventRecord(e2, stream2);
cudaStreamWaitEvent(stream1, e2);
D<<< ..., stream1 >>>();

// now convert the stream to a graph
cudaStreamEndCapture(stream1, &graph);
```



# *Execution Models*

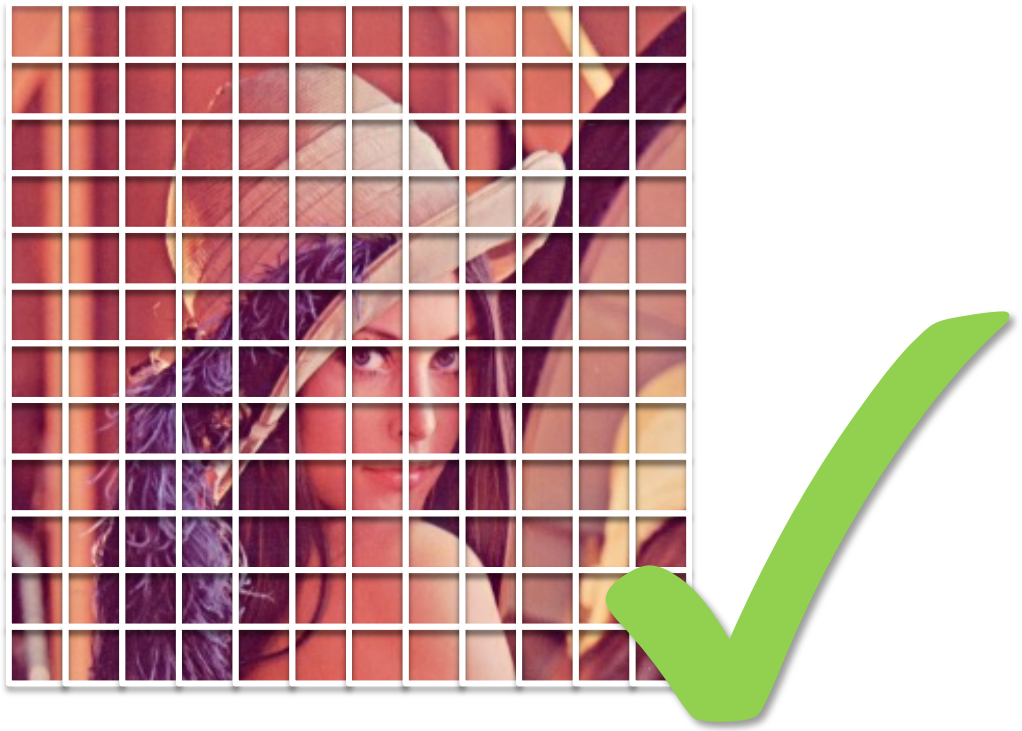
# Kernel-based Programming Model

## Execution Models



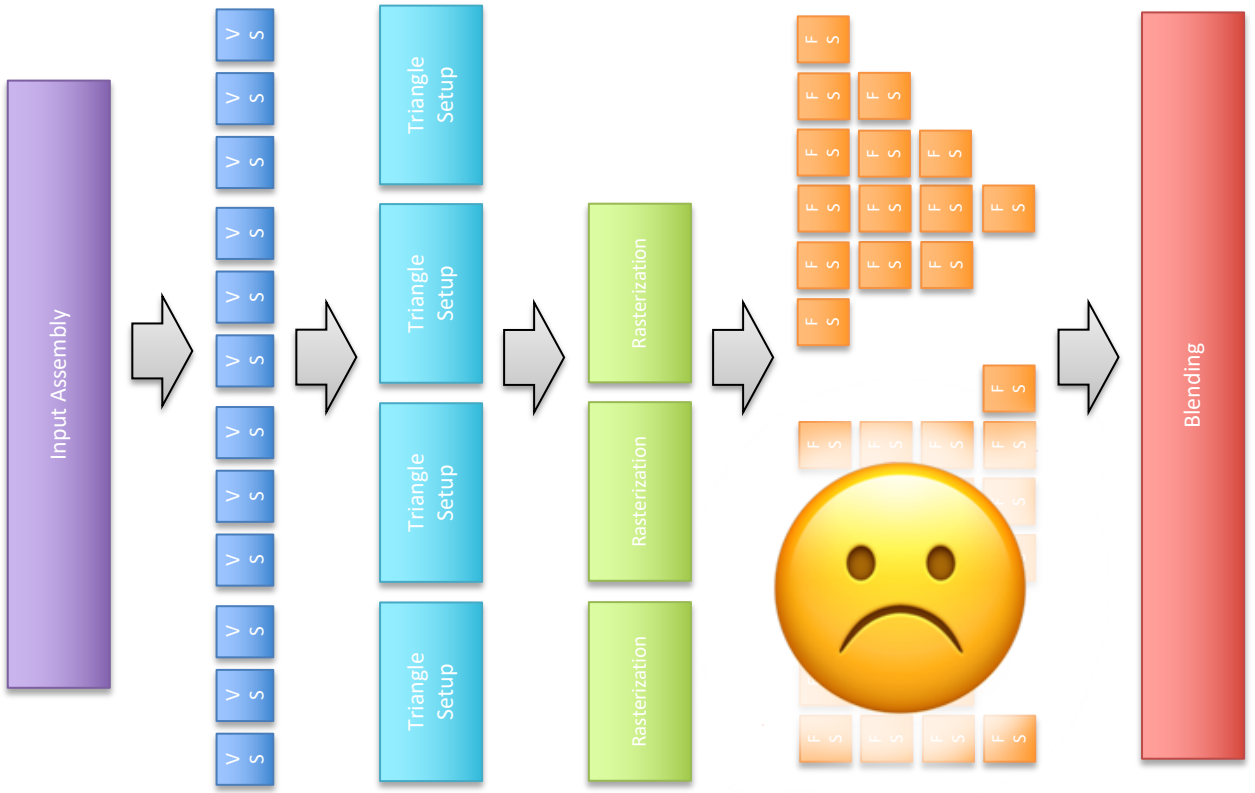
# Kernel-based Programming Model

## Execution Models



# Kernel-based Programming Model

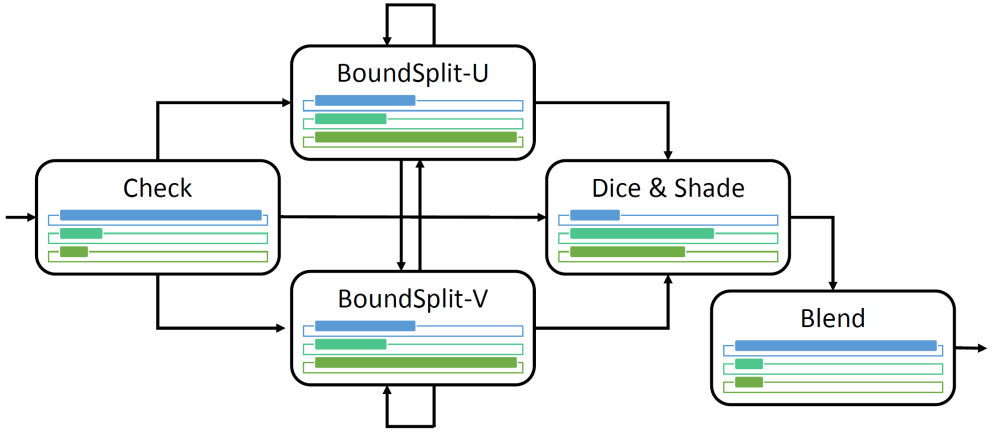
## Execution Models

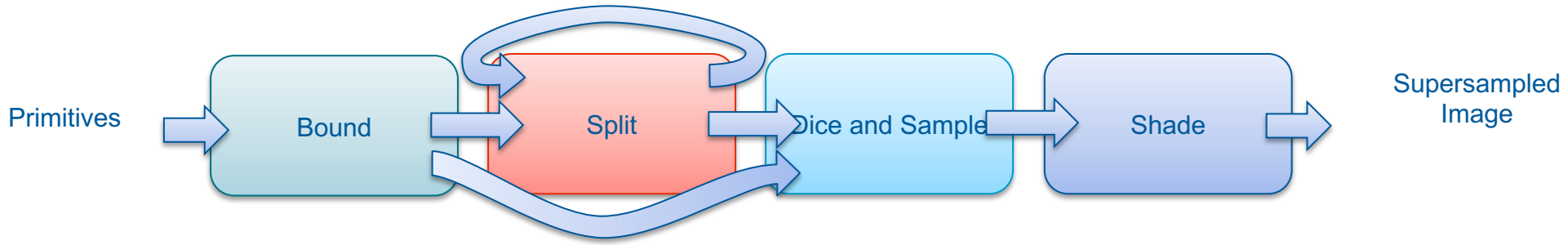


# What do we need to solve?

## Execution Models

- We want
  - Handle heterogeneous workloads
  - Dynamic work generation
  - Efficient scheduling
  - Exploit shared memory

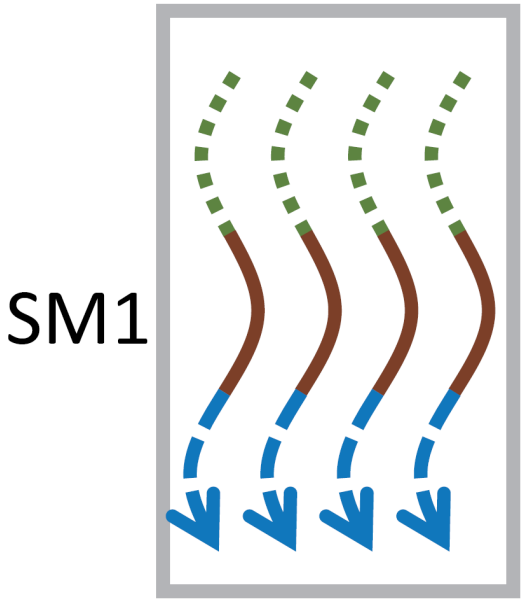


# Run to Completion

## Execution Models

- Simplest execution model
- All stages in a single kernel
  - Does NOT support
    - Global synchronization
    - Dynamic work generation
  - Resource requirements of largest stage

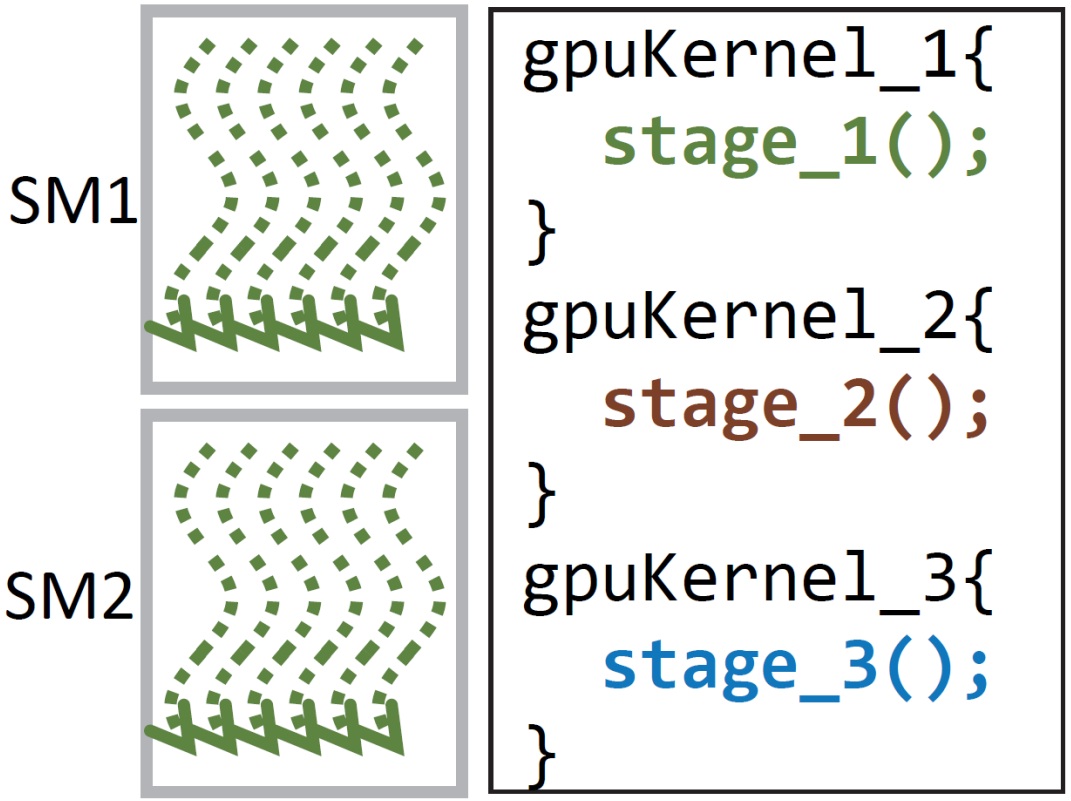



```
gpuKernel
{
   stage_1();
   stage_2();
   stage_3();
}
```

# *Kernel by Kernel*

*Execution Models*

- Most commonly used
- Split application into series of kernel launches
  - Each kernel tailored to task
    - Requirements per kernel
  - CPU synchronization
  - Requires controller on **CPU** for dynamic work generation



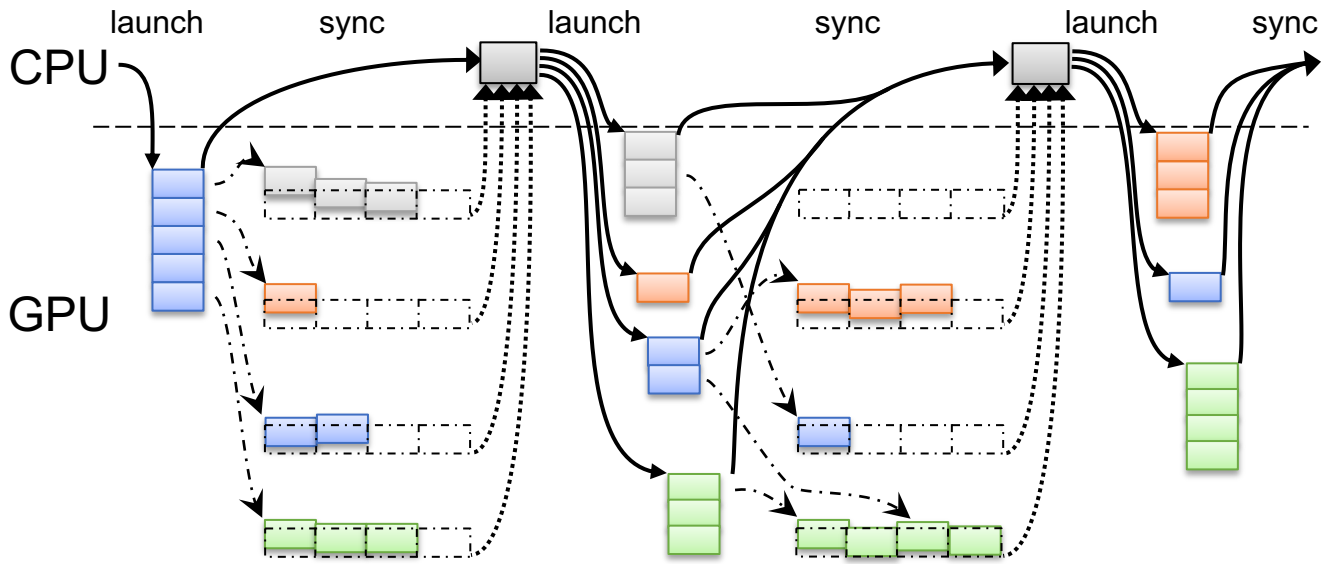
# *Time-Sliced Kernel*

*Execution Models*

- Variant of **Kernel-by-Kernel** that supports dynamic work generation
- **CPU** checks amount of work per task
  - Launches kernels with work
    - Into separate streams for concurrent execution
  - Wait for kernels to finish
    - Check work again and start launching again

# Time-Sliced Kernel

## Execution Models




+ no divergence

+ optimal occupancy

– CPU synchronization

Laine et al. [HPG'13]

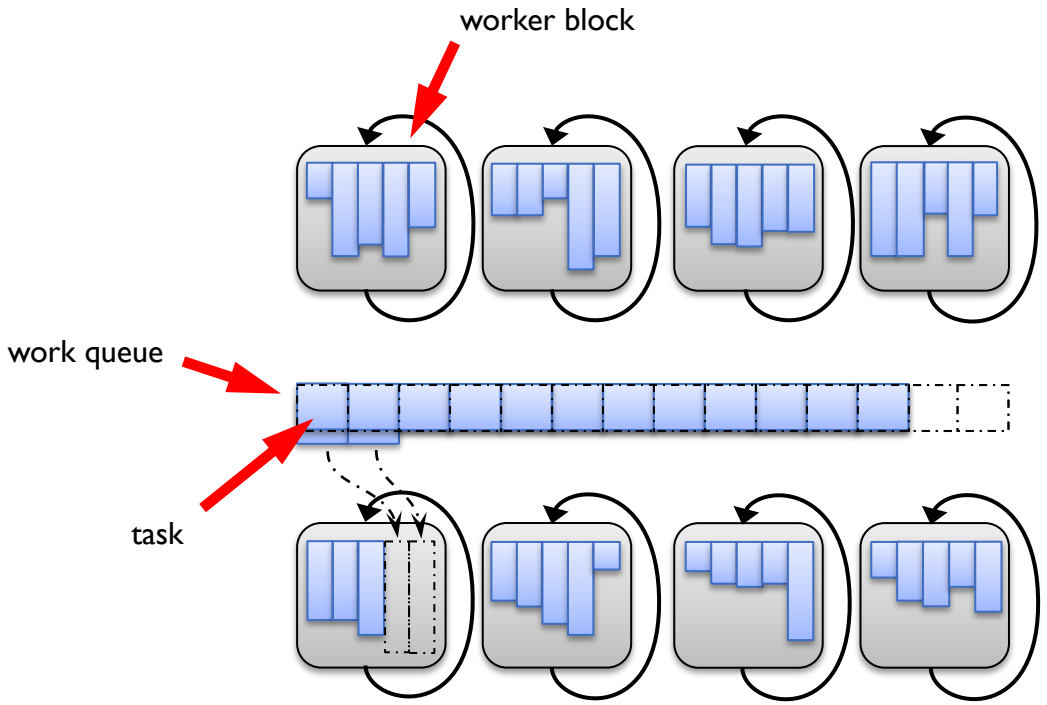
# *Persistent Threads*

## *Execution Models*

- Threads execute in a loop
- Global work queue
  - Draw in new work from queue
  - Execute work
  - Enqueue new work
    - Depends on the queue implementation
  - Continue until no work left
- Implicit load balancing

# Persistent Threads

## Execution Models




+ load balancing

+ (dynamic work generation)

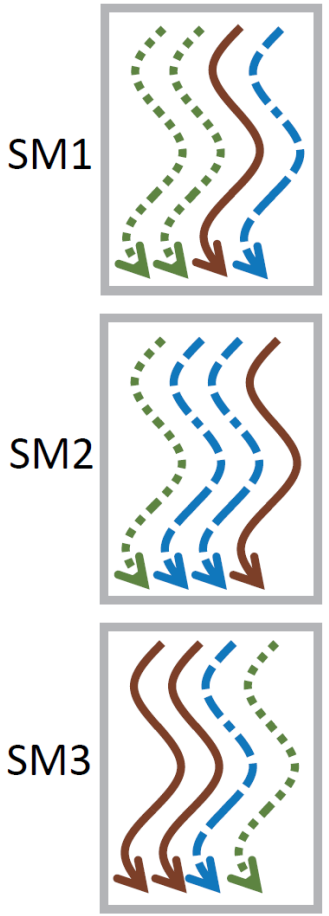
– only one type of task

Aila and Laine [HPG'09]

# *Megakernel*

*Execution Models*

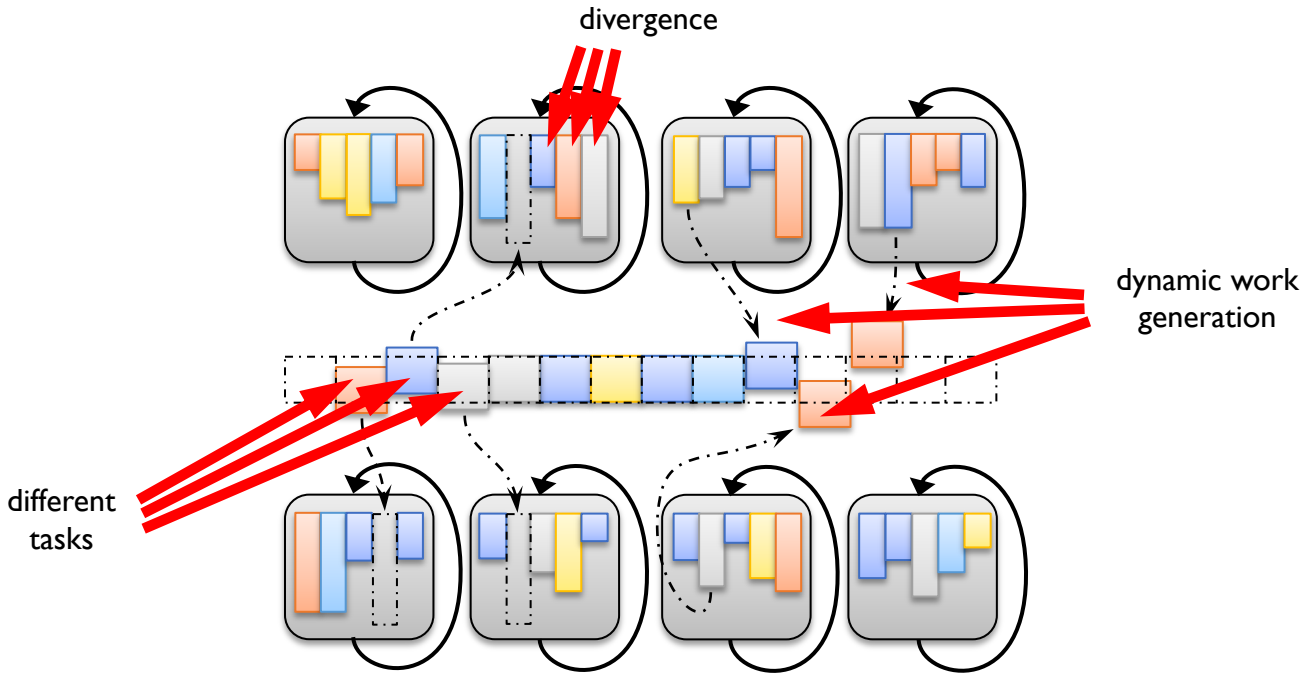
- Generalized version of **persistent threads**
  - Can handle different task types
  - Depending on queue also dynamic work generation
- May suffer from divergence
- Occupancy still bound by largest procedure




```
gpuKernel
{
  while(item=schedule())
  {
    switch item.type
    {
    case 1:
      stage_1(); break;
    case 2:
      stage_2(); break;
    case 3:
      stage_3(); break;
    }
  }
}
```

# Persistent Megakernel

## Execution Models

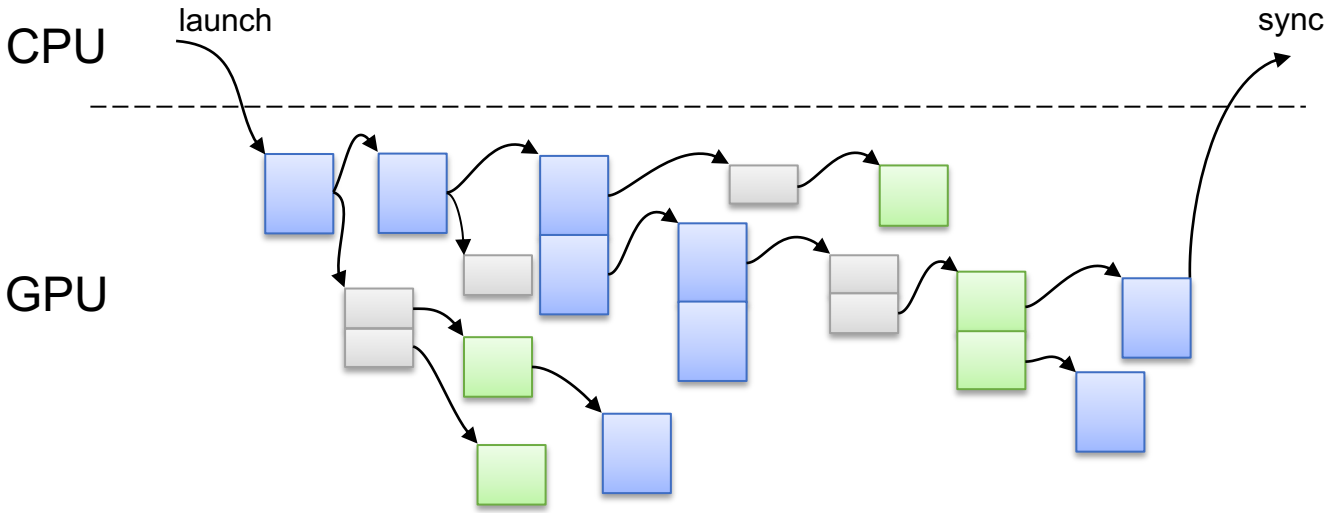



+ load balancing
+ dynamic work generation
+ multitasking
– divergence
– suboptimal occupancy
– bottleneck: work queue

Steinberger et al. [TOG'2012]

# Dynamic Parallelism

## Execution Models

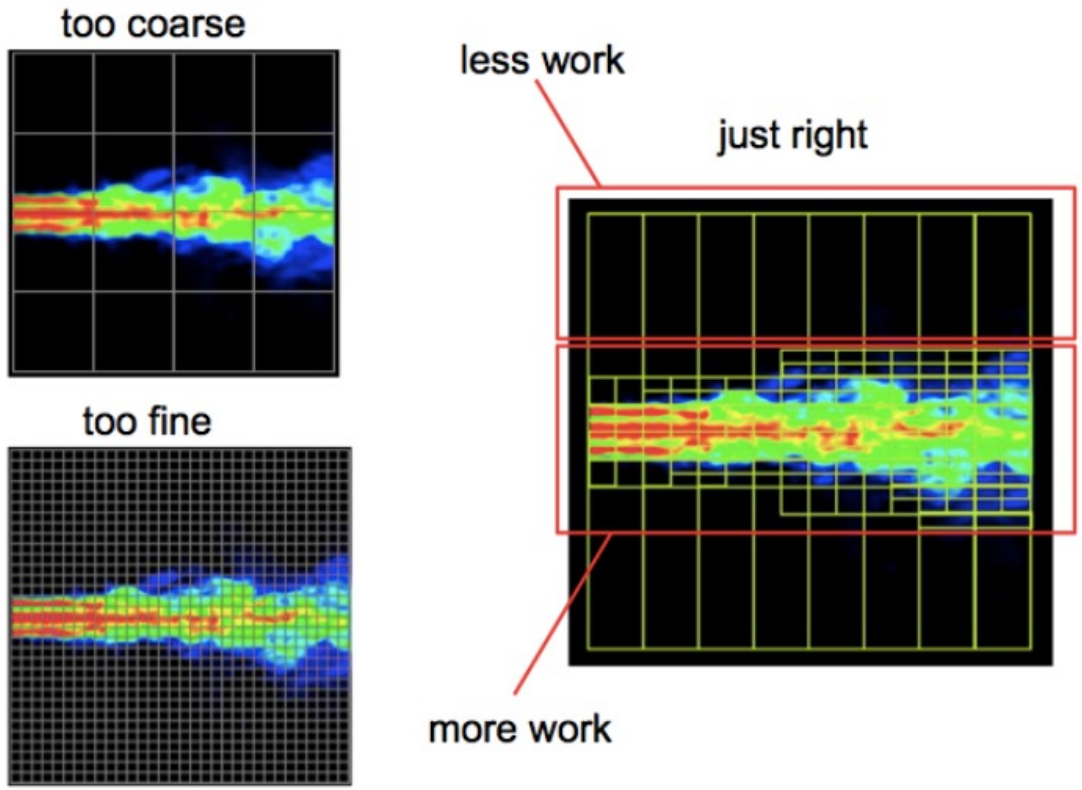



NVIDIA [2012]

# Dynamic Parallelism

## Execution Models

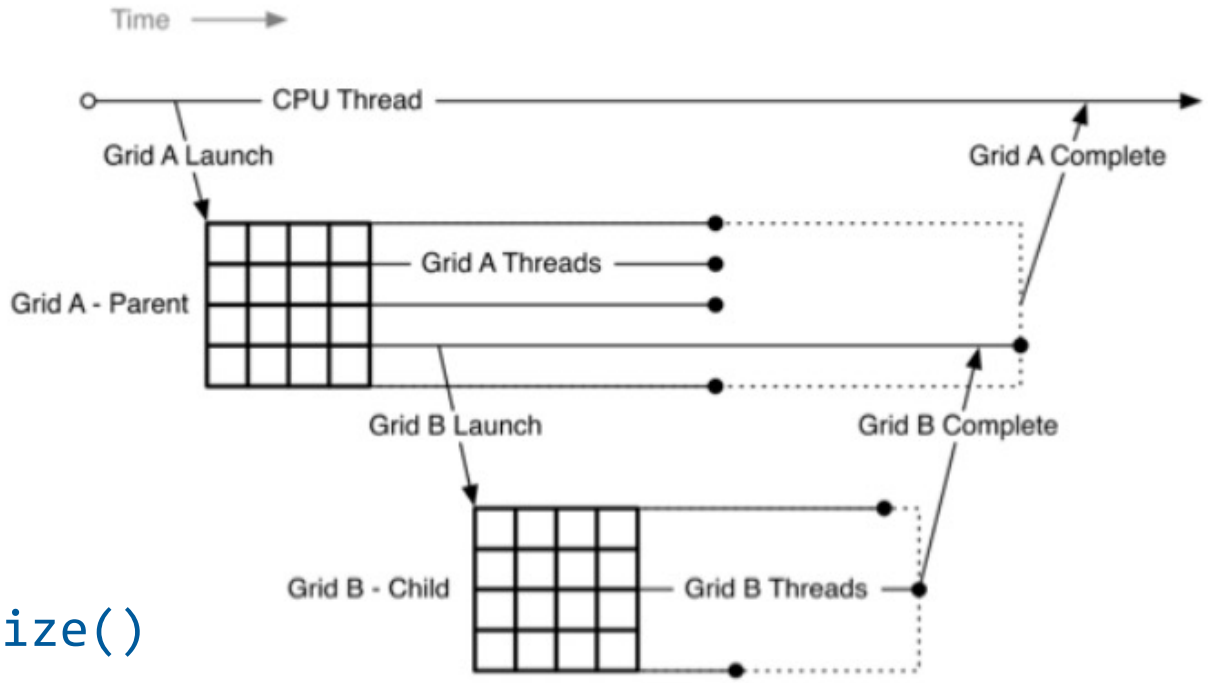
- Nested parallelism occurs in many applications
- Since CUDA 5.0
  - Kernels can launch other kernels
  - Dynamically adapt to amount of work
- Link with `cudadevrt`
  - Compile with `-rdc`



# *Dynamic Parallelism*

## *Execution Models*

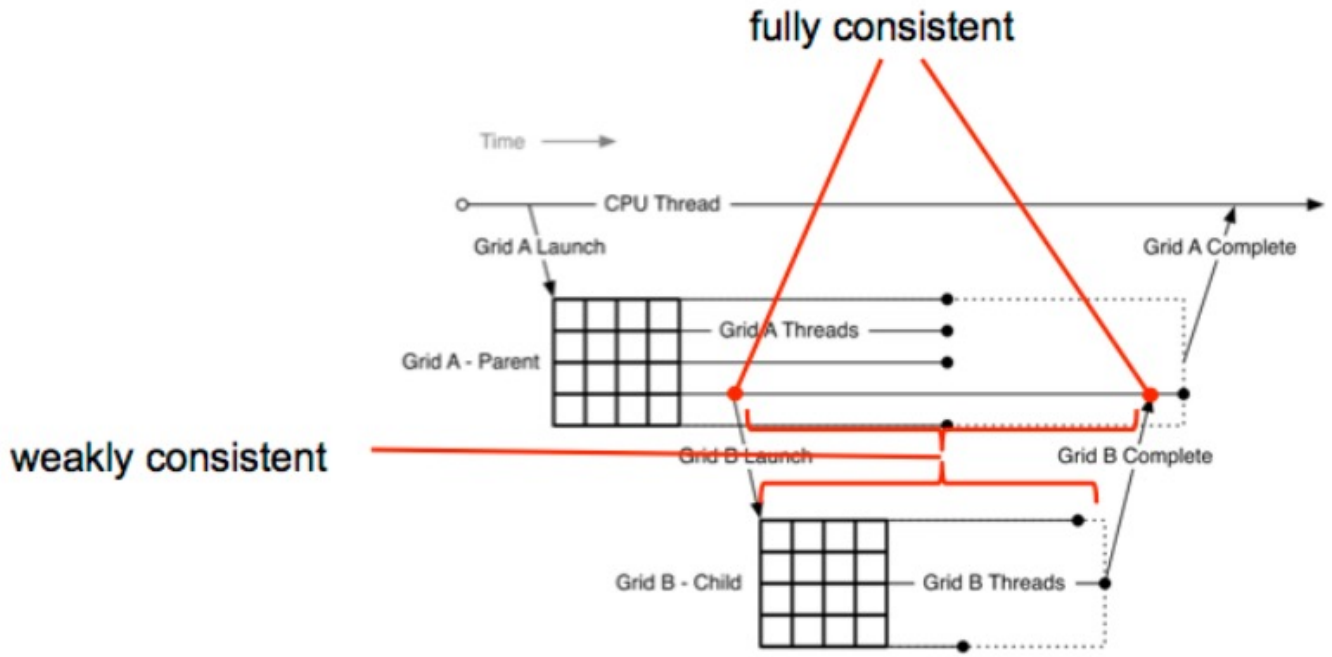
- Group of blocks of threads is called a *grid*
  - *Parent grid* launches *child grids*
- **Child grid** inherits attributes
  - L1 cache
  - Shared memory configuration
  - Stack size
- **Child grids** are *fully nested*
  - **Parent grid** can call `cudaDeviceSynchronize()`
    - Only thread which launches is aware of kernel launch



# Dynamic Parallelism

## Execution Models

- `cudaDeviceSynchronize()` can be expensive
  - May cause the currently running block to be paused and swapped to global memory
- *Fully-consistent view* of global memory
  - Both directions with sync
  - Weakly consistent in-between
- Passing pointers to *child grid*
  - \+ Global, zero-copy host and constant
  - \- Shared and local memory



- *child grids* launched sequentially
  - Happens even if launched by different threads
  - **Use streams**

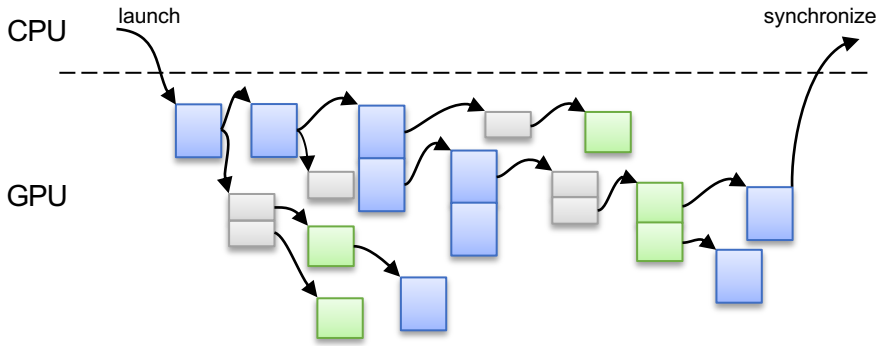
```
cudaStream_t stream;
cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
```

- Streams on device are *non-blocking*
  - *Kernels* in different streams **can** execute concurrently
    - **Do not rely on that!**
  - Streams in different blocks are *different*
    - Streams in same block can be used by all threads in block
  - `cudaStreamDestroy()` returns kernels immediately

- Recursion depth
  - Nesting depth
    - Kernels launched from host (depth = 0)
    - **Hardware limit = 24**
  - Synchronization depth
    - Deepest level to sync (default = 2)
      - `cudaLimitDevRuntimeSyncDepth()`
- Pending launches (default: 2048)
  - `cudaDeviceSetLimit(cudaLimitDevRuntimePendingLaunchCount, 123456)`
  - *Virtualized pool* (more flexible, but additional launches more costly)

# Dynamic Parallelism

## Execution Models




+ dynamic work generation
+ GPU autonomy
+ optimal occupancy
– no fine-grained work generation though
– cannot use local memory to pass on data
– limited launch depth