CSE 531

Heterogeneous Computing and GPU Programming (Part III)

Spring 2023

Mahmut Taylan Kandemir

ACK: M. Krulis, G. Barlas, X. Tang, L. Mao

Types of parallelism

Different Types of Parallelism

- Task parallelism
  - The problem is divided into tasks, which are processed independently
- Data parallelism
  - The same operation is performed over many data items
- Pipeline parallelism
  - Data are flowing through a sequence (or oriented graph) of stages, which operate concurrently
- Other types of parallelism
  - Event-driven, …

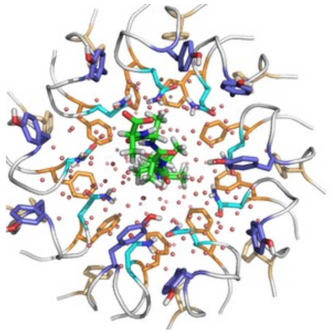
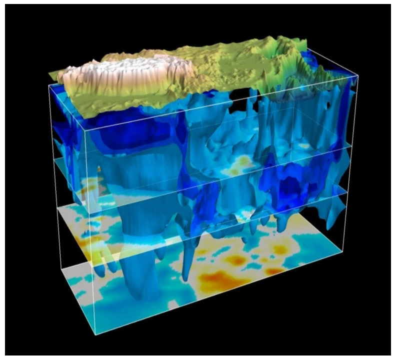
GPU execution model

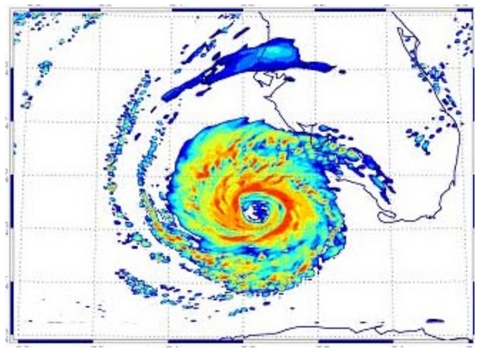
Parallelism in GPU

- Data parallelism
  - The same kernel is executed by many threads
  - Thread processes one iteration work
- Limited task parallelism
  - Multiple kernels executed simultaneously (since Fermi)
  - At most as many kernels as SMPs
  - Streams
- But, we do *not* have…
  - Any guarantees that two blocks/kernels will actually run concurrently
  - Efficient means of synchronization outside the block

Irregular applications

- Irregular applications
  - Molecular dynamics
  - Earthquake simulation
  - Weather simulations
  - Economics



- Most of these applications use GPUs for acceleration

Image sources:
http://www.nature.com/nature/journal/v516/n7530/fig_tab/nature13768_SF5.html
https://en.wikipedia.org/wiki/Weather_Research_and_Forecasting_Model
http://www.tibetnature.net/en/massive-fast-moving-structure-seen-beneath-tibetan-plateau-in-earthquake-simulation-study/
https://hgchicago.org/courses/advanced-courses/economic-science/

Irregular data accesses

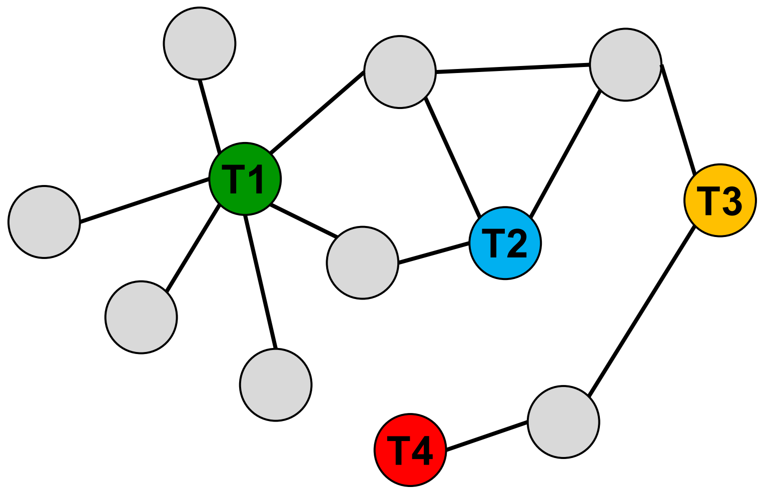
They come in many forms

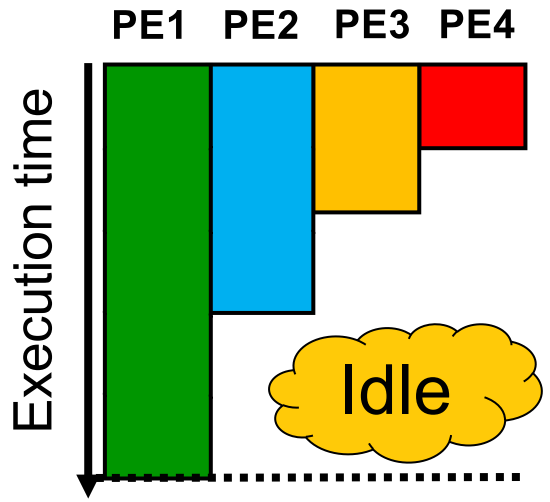
An example is **index arrays**:

```
for (i=0; i<N; i++)
    ind[i] = input();
…
for (j=0;j<N;j++)
    X[j] = …
    ….  =. X[ind[j]]
```

Irregularity: source of GPU inefficiency

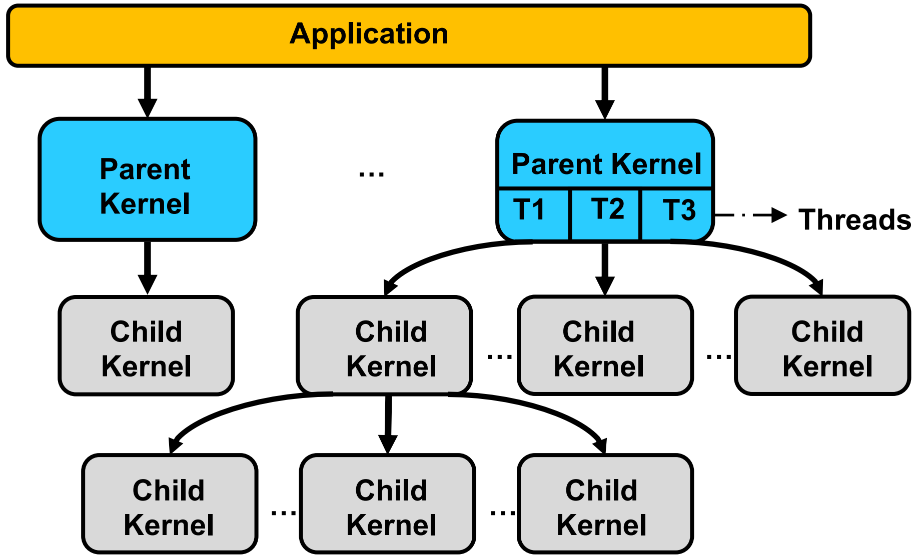
- Imbalanced workload among GPU threads
  - Thread-1 (T1) undertakes more computations than other threads.
- Underutilized hardware resources
  - Threads, register files, shared memory, etc.

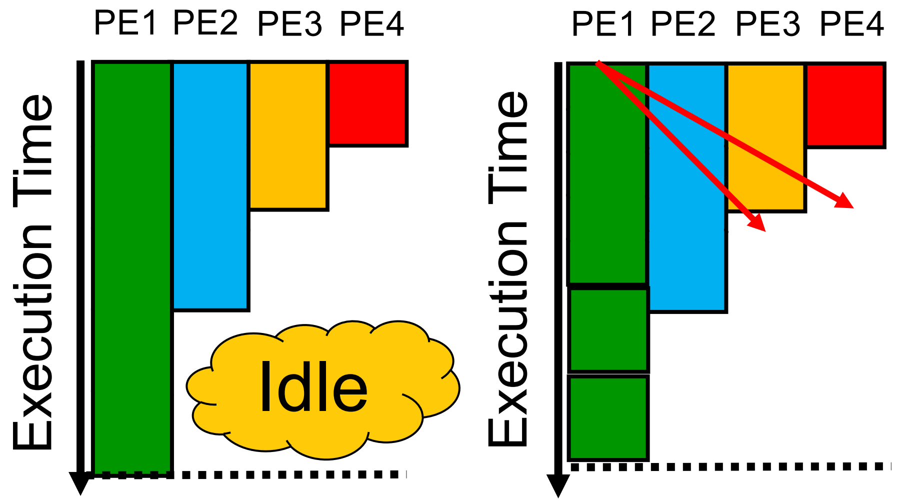




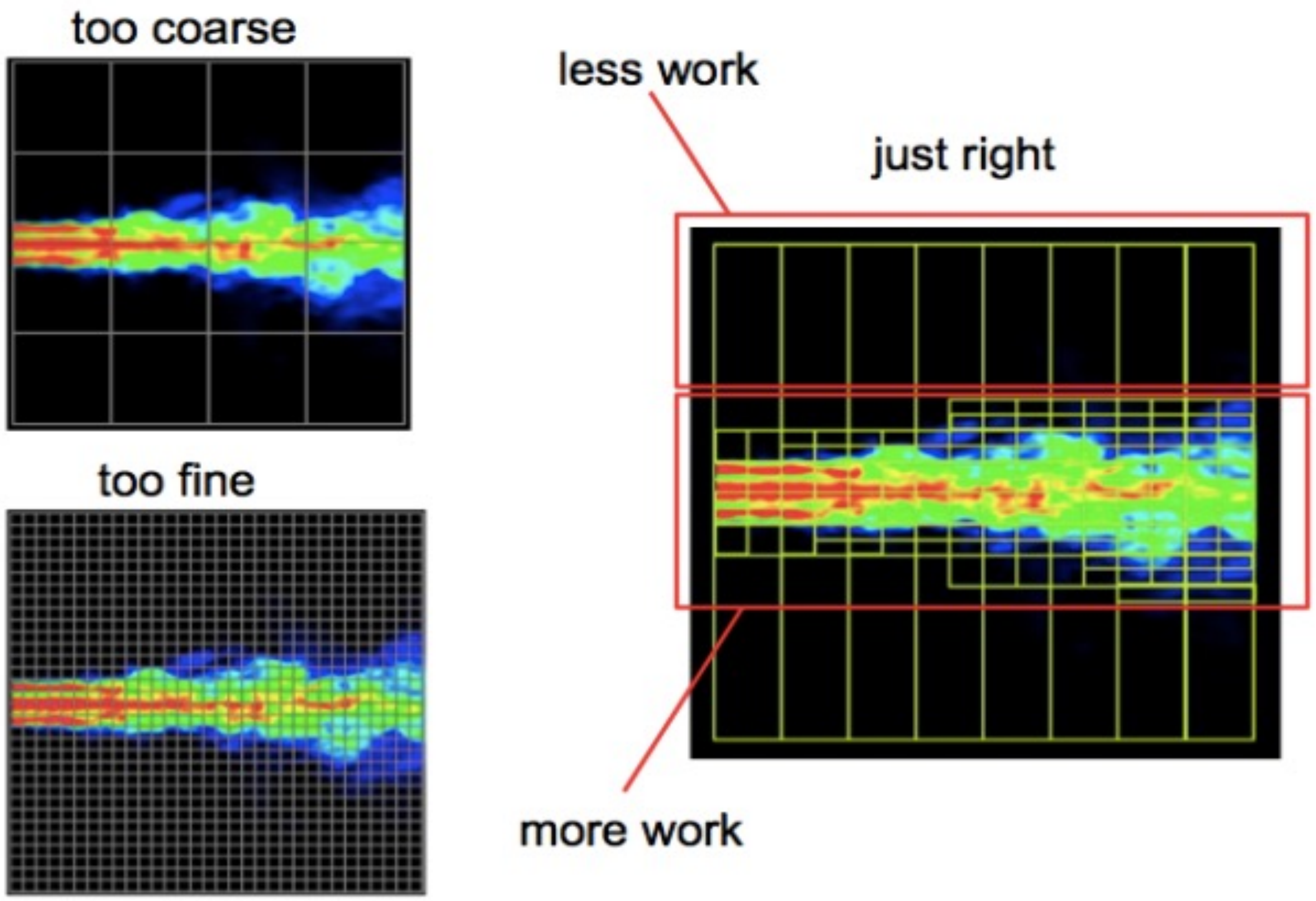
Dynamic parallelism in GPUs

- **Dynamic parallelism**
  - Allow applications to launch kernels at the device (GPU) side, without CPU intervention.
- **Advantages**
  - High GPU occupancy
  - Dynamic load balancing
  - Supports recursive parallel applications
    - Many applications are indeed recursive





Selling point of dynamic parallelism

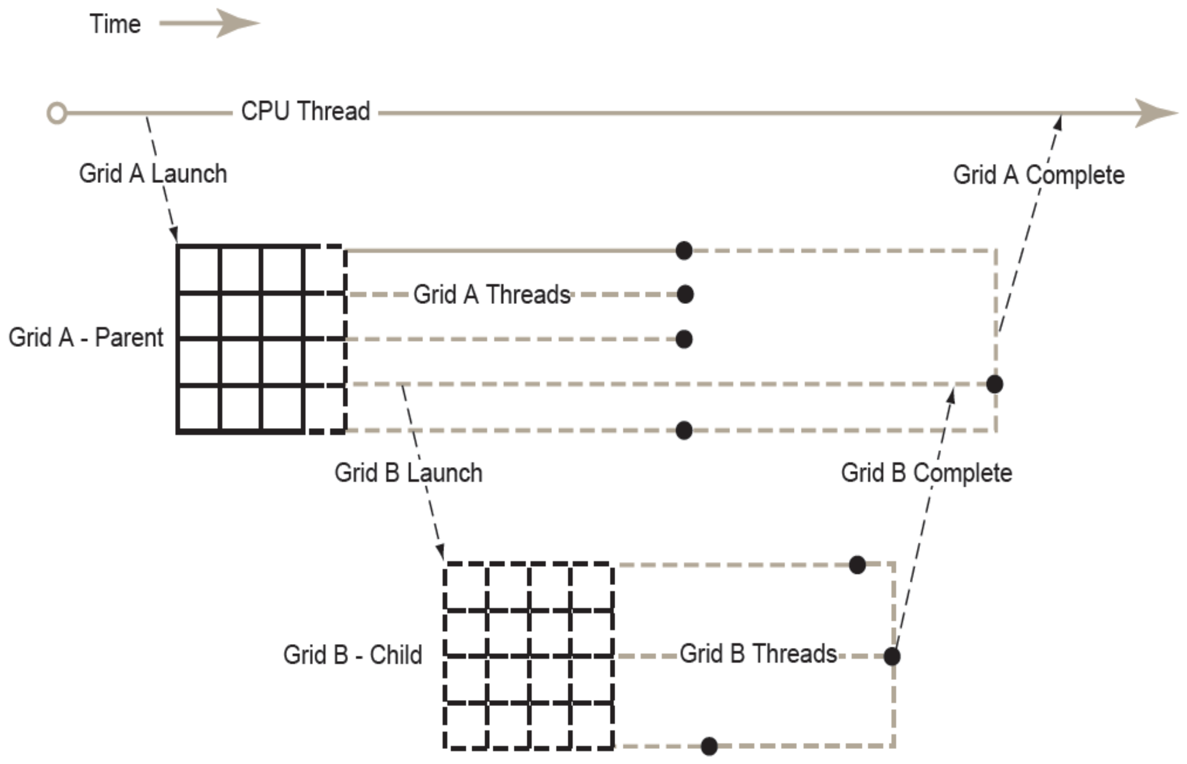


Dynamic parallelism

CUDA Dynamic Parallelism

Presented in CC 3.5 (Kepler)

GPU threads are allowed to launch new grids

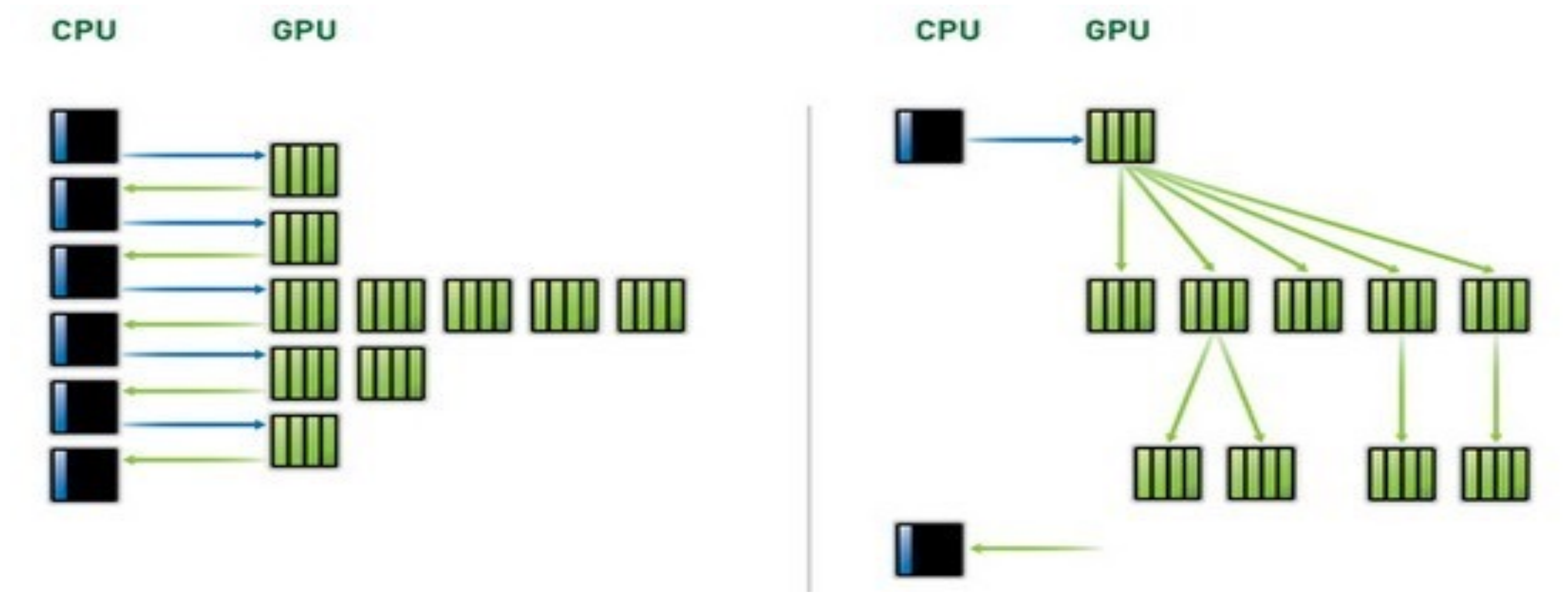


Dynamic parallelism

Dynamic Parallelism Purpose

The device does not need to synchronize with host to issue new work to the device

Irregular parallelism may be expressed more easily (naturally)



Dynamic parallelism

How It Works

- Portions of CUDA runtime are ported to the device
  - Kernel execution
  - Device synchronization
  - Streams, events, and async. memory operations
- Kernel launches are asynchronous
  - No guarantee the child kernel starts immediately
  - Synchronization points may cause a context switch
    - Context of an entire block has to be switched off an SMP
- Block-wise locality of resources
  - Streams and events are valid within a thread block

Dynamic parallelism

Example

```
__global__ void child_launch(int *data) {
    data[threadIdx.x] = data[threadIdx.x]+1;
}

__global__ void parent_launch(int *data) {
    data[threadIdx.x] = threadIdx.x;
    __syncthreads();
    if (threadIdx.x == 0) {
        child_launch<<< 1, 256 >>>(data);
        cudaDeviceSynchronize();
    }
    __syncthreads();
}

void host_launch(int *data) {
    parent_launch<<< 1, 256 >>>(data);
}
```

Thread 0 invokes a grid of child threads

Synchronization does not have to be invoked by all threads

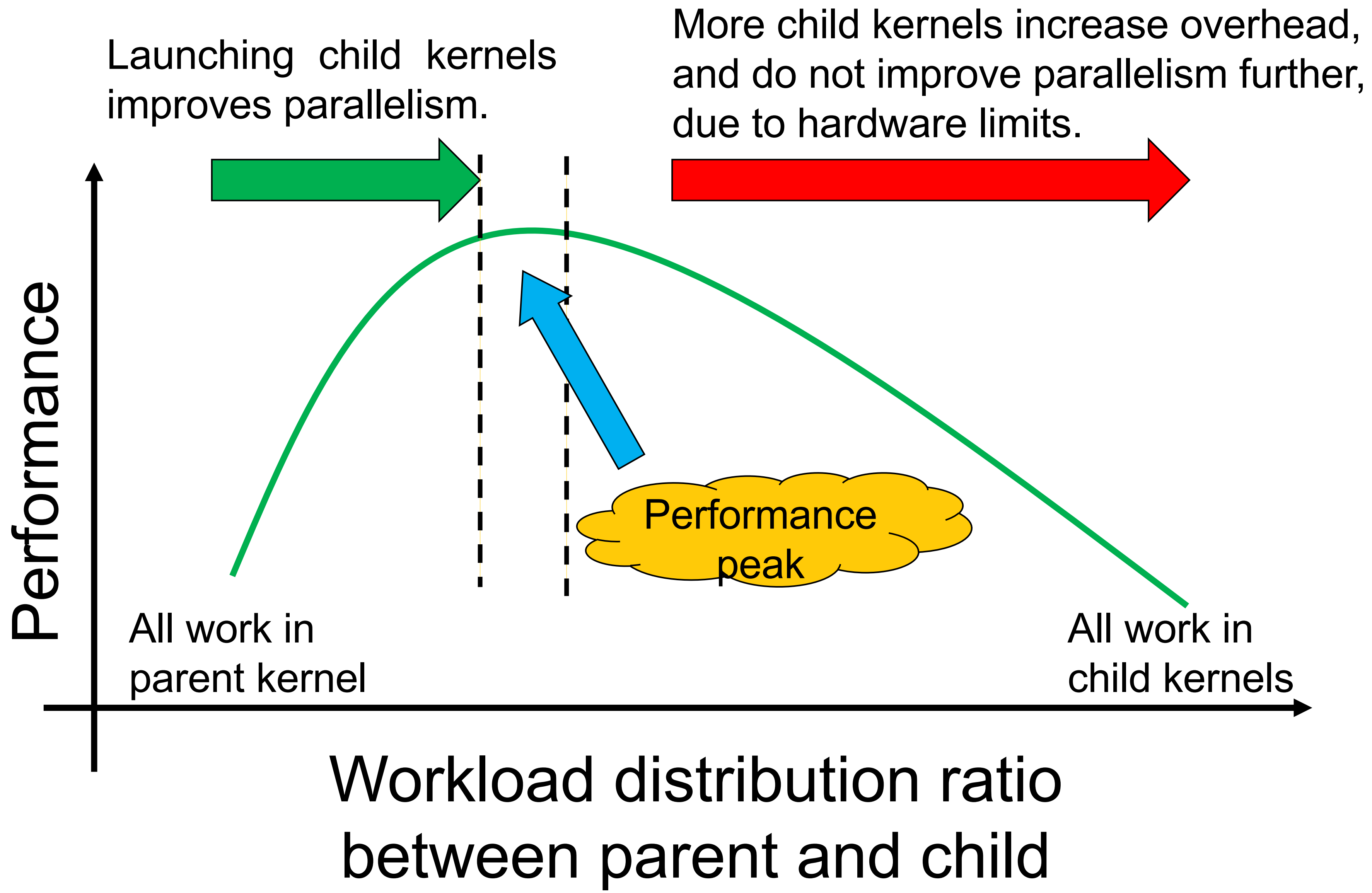
Parent threads are synchronized

Dynamic parallelism

Depth Limits

- Nesting depth (depth of the recursion)
- Synchronization depth (deepest level where **`cudaDeviceSynchronize()`** is invoked)
- **`cudaDeviceSetLimit()`**
  - **`cudaLimitDevRuntimeSyncDepth`**
  - **`cudaLimitDevRuntimePendingLaunchCount`** number of pending grids

Tradeoff in dynamic parallelism


Launching child kernels improves parallelism.
More child kernels increase overhead, and do not improve parallelism further, due to hardware limits.
Performance
Performance peak
All work in parent kernel
All work in child kernels
Workload distribution ratio between parent and child

Moral of the story

Dynamic parallelism improves parallelism for irregular applications running on GPUs.

However, aggressively launching child kernels without knowledge of hardware state degrades performance.

Dynamic parallelism needs to make smarter decisions on how many child kernels should be launched.

Parameters of the game

- Three key parameters related to child kernel.
  - Workload distribution ratio (THRESHOLD)
  - Child kernel dimension (c_grid, c_cta)
  - Child kernel execution order (c_stream)

Host code

```
int main(){
    …
    parent<<< p_grid, p_cta>>>(type *workload);
    …}
```

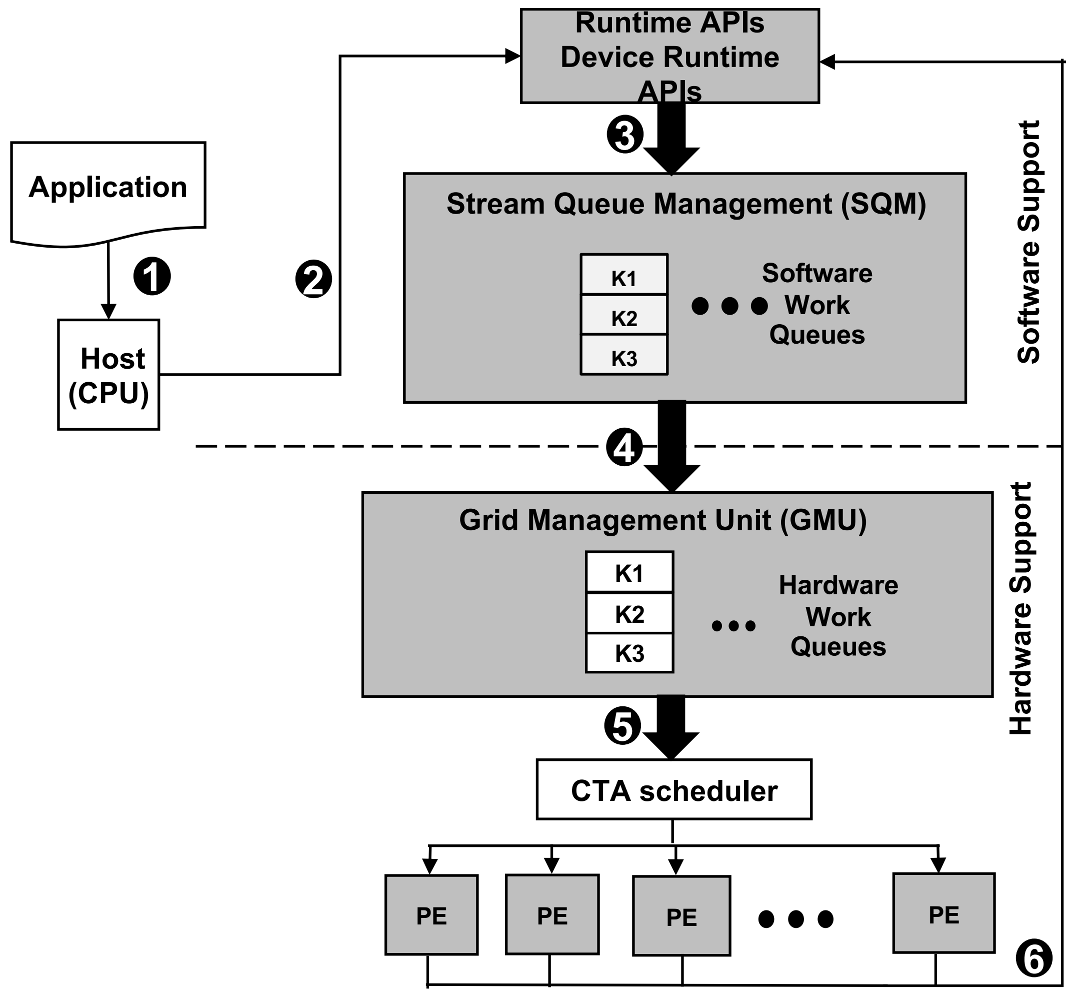
Parent kernel

```
parent (type *workload){
    …
    if (local_workload > THRESHOLD){
        child<<< c_grid, c_cta, shmem, c_stream>>>
    else
        while(local_workload){…}
    …}
```

Child kernel

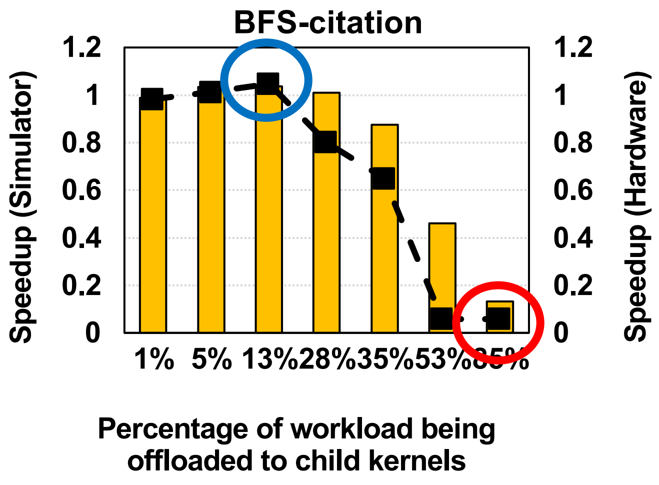
```
    __global__ void child(*c_workload){
        …
    }
```

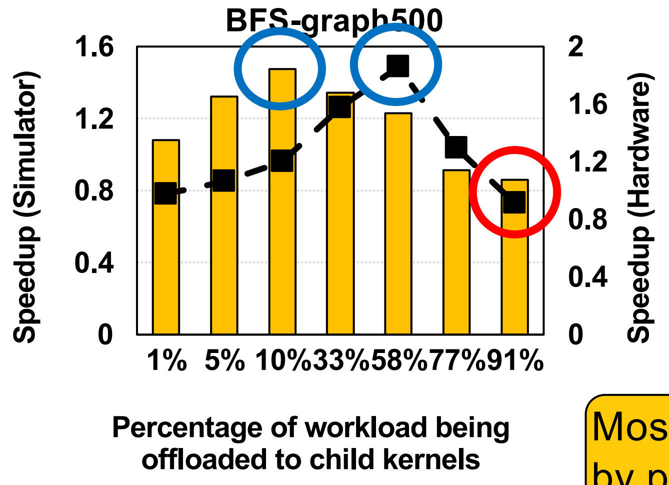
Hardware support for dynamic parallelism

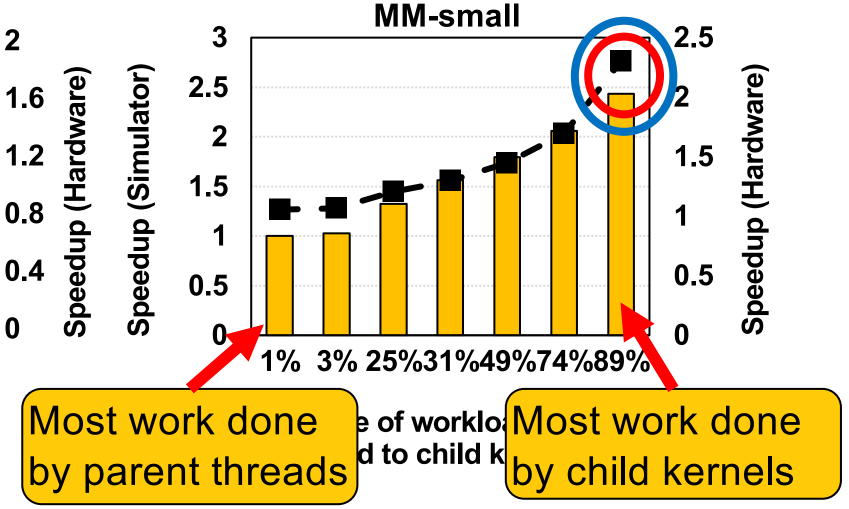


Application characterization

- Three key parameters:
  - Workload distribution ratio, child kernel dimension, child kernel sequence
  - The first two can be controlled at the user-level
- Observations:
  - Workload distribution is very important.
  - Preferred workload distribution ratio varies for different applications and inputs.
- **Best offline-search:** peak performance through manually varying the workload distribution ratio (THRESHOLD).
- **Baseline-DP:** offload most computations by launching child kernels.

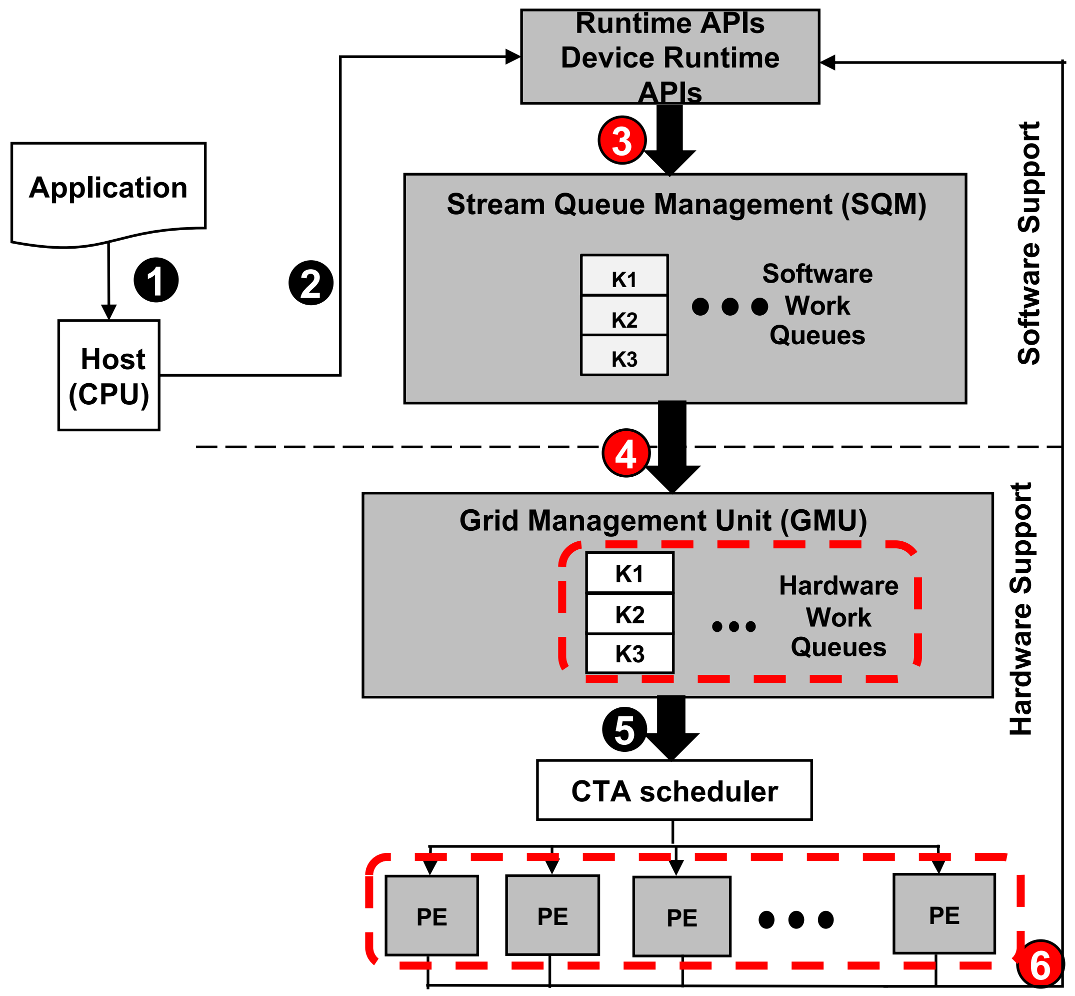




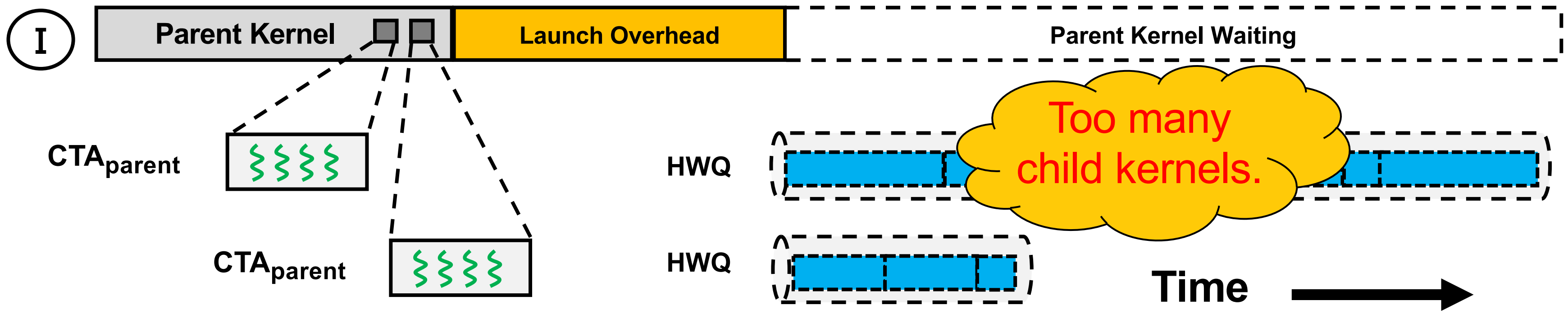


Overheads

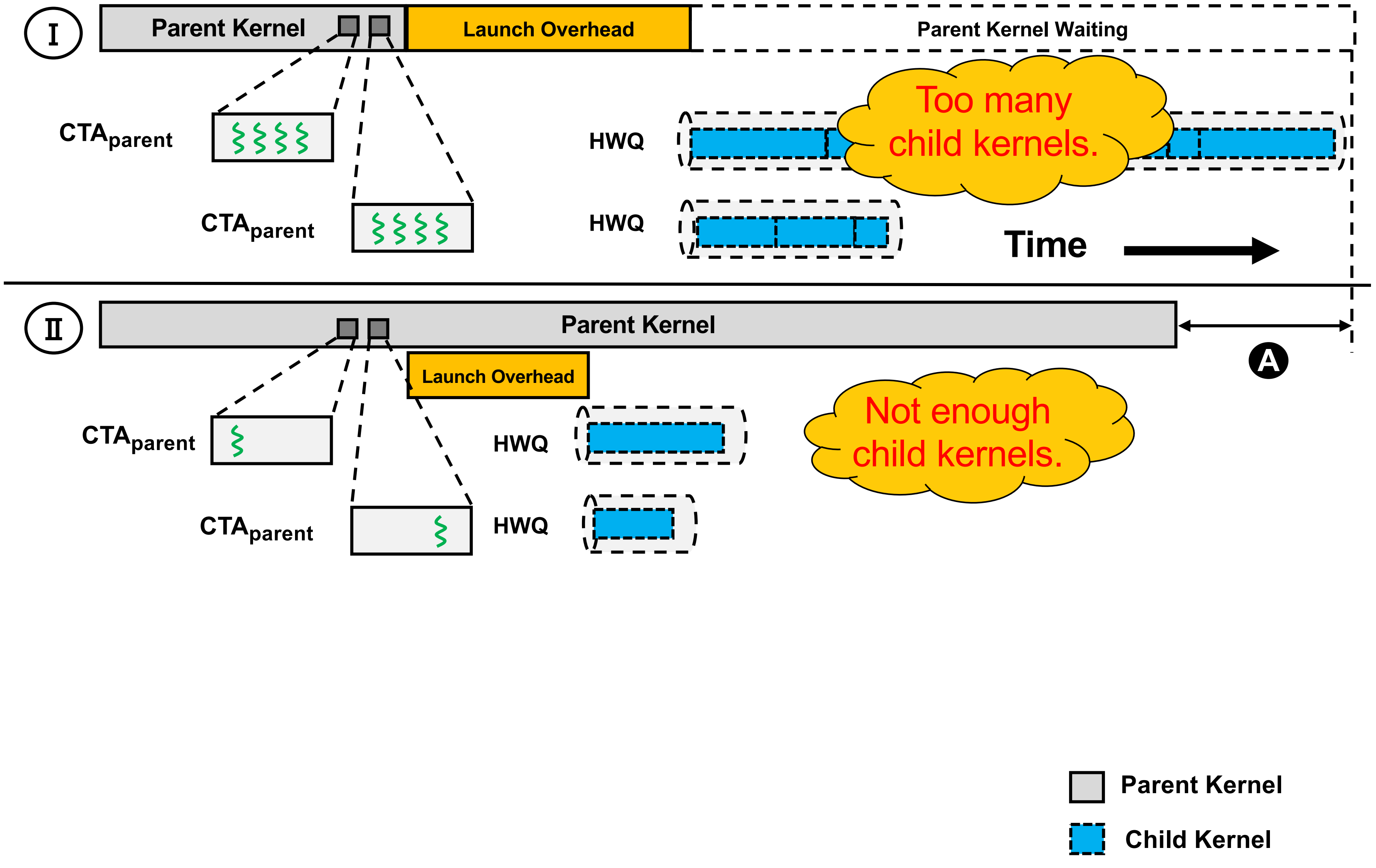
- **Launch overhead:** from invoking API to kernels being pushed into GMU and ready to start execution.
  = 6 + 3 + 4
- **Queuing latency:** Kernels waiting in GMU due to the hardware limitations.
  - Maximum number of concurrently running kernels.
  - Maximum number of concurrently running CTAs (thread blocks).
  - Available hardware resources.



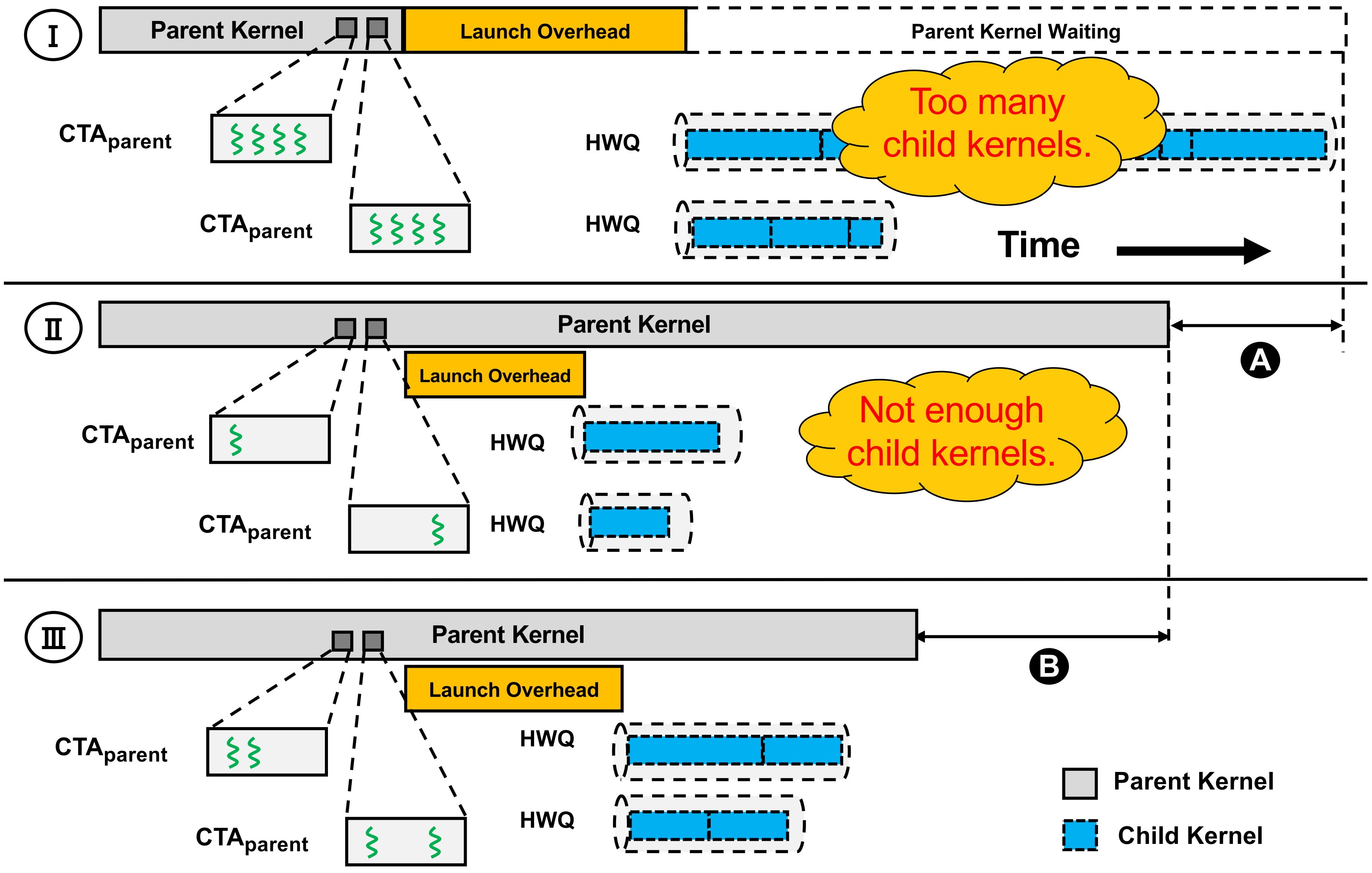
Workload distribution



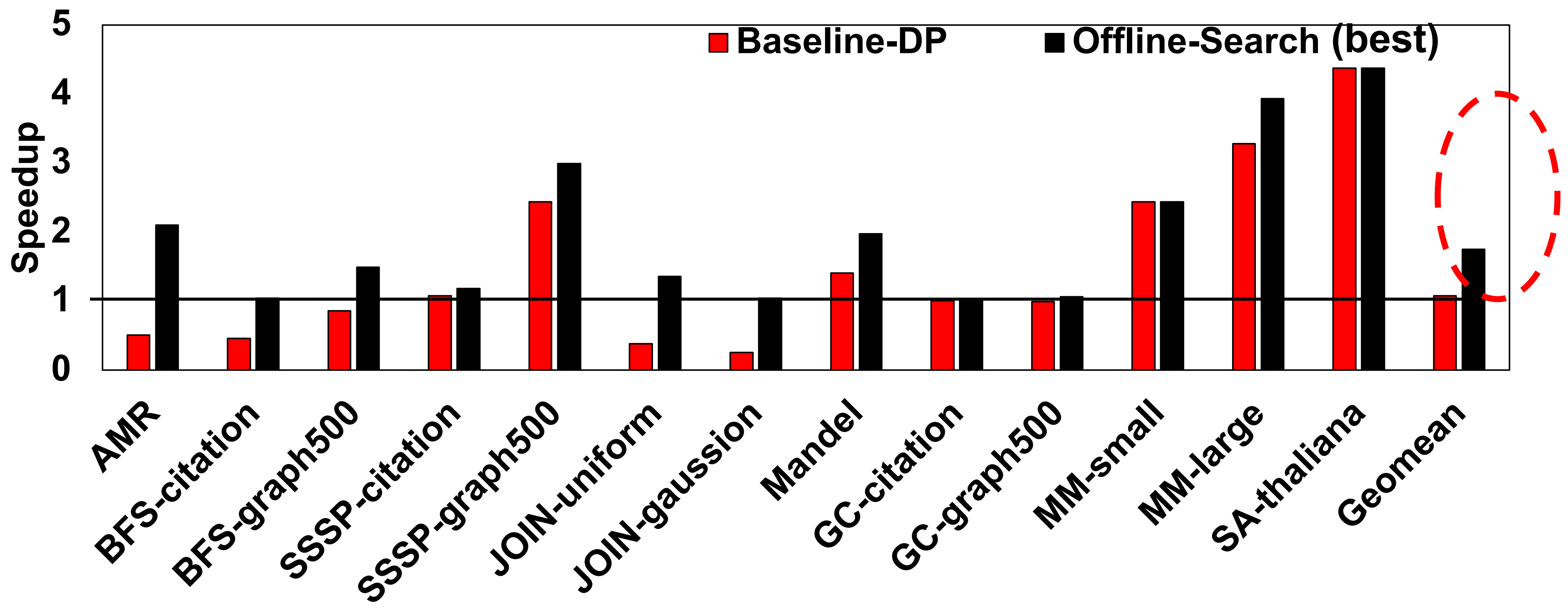
Workload distribution



Workload distribution



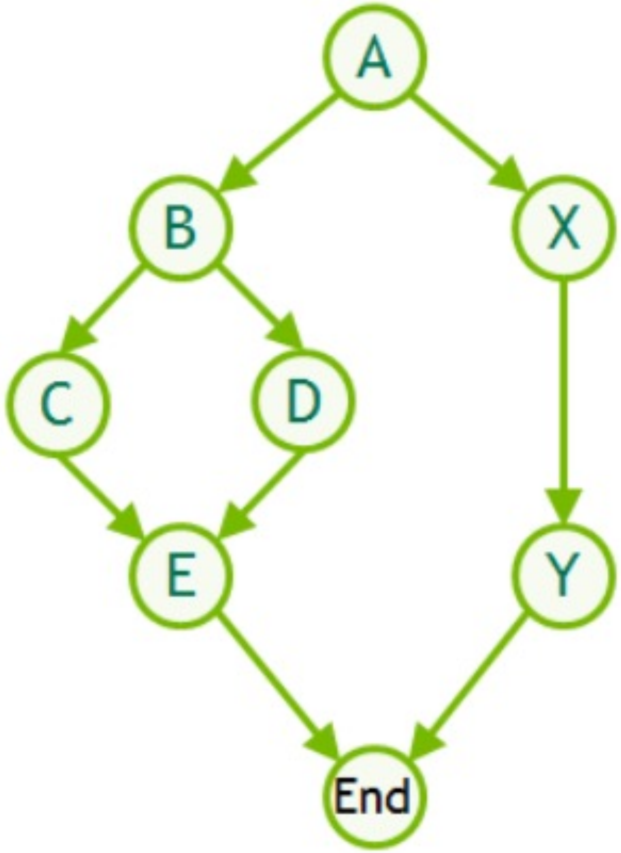
Quantification of the opportunity



CUDA graphs

Most kernels are quite small

- So, the launching overhead becomes more significant
- If many small kernels are executed consecutively, we need to overlap kernel launching and execution
- CUDA Graphs allow us to **pre-record** the sequence (graph) of kernel launches (and their synchronizations) and execute them ***all at once***


$$\begin{tikzcd} & & A \arrow[dl] \arrow[dr] & \\ & B \arrow[dl] \arrow[dr] & & X \arrow[dd] \\ C \arrow[dr] & & D \arrow[dl] & \\ & E \arrow[dr] & & Y \arrow[dl] \\ & & \text{End} & \end{tikzcd}$$

CUDA graphs

Example

And assuming there are many epochs (e.g., thousands)

```
constexpr int steps = 20;
for (int epoch = 0; epoch < epochs; ++epoch) {
  for (int i = 0; i < steps; ++i) {
    stencilKernel<<<..., stream>>>(...);
  }
  smoothKernel<<<..., stream>>>(...);
  cudaStreamSynchronize(stream);
  ...
}
```

Assuming the kernels will take microseconds to execute

CUDA graphs

```
cudaGraph_t graph;

cudaGraphExec_t instance;

bool graphCreated = false;

for (int epoch = 0; epoch < epochs; ++epoch) {

  if (!graphCreated) {
    cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
    for (int i = 0; i < steps; ++i) {

      stencilKernel<<<..., stream>>>(...);

    }

    smoothKernel<<<..., stream>>>(...);

    cudaStreamEndCapture(stream, &graph);

    cudaGraphInstantiate(&instance, graph, NULL, NULL, 0);

    graphCreated = true;

  }

  cudaGraphLaunch(instance, stream);

  cudaStreamSynchronize(stream);

}
```

Graph holds the structure

Not actually executed, only captured for the graph

All captured kernels launched (in order) at once

CUDA graphs

- A **CUDA graph** is a representation of a nonlinear sequence of operations that are supposed to take place on a GPU, along with their *dependencies*.
- These operations include *both* kernel launches and memory transfers.
- The graph *nodes* correspond to the *operations* and the *edges* to their *dependencies*.
- The benefit of using a CUDA graph is:
  - Cleaner code
  - Once the graph is defined, CUDA enables a number of optimizations that are not possible to be implemented otherwise.
- A CUDA graph can minimize the setup time required to perform a series of computations on a GPU, while maximizing the concurrency by overlapping non-competing or dependent operations.

CUDA graphs

- Working with a CUDA graph requires three steps:
  - **Definition:** Describing the nodes and edges connecting them.
  - **Instantiation:** Performing the setup work associated with launching the graph, producing an **executable graph**.
  - **Execution:** An executable graph can be launched into a stream like any other regular operation.
- Supported node types:
  - kernel,
  - host function call,
  - memory copy,
  - memory initialization (e.g., cudaMemset),
  - a child graph (graphs can be nested),
  - an empty node (placeholder for establishing dependencies).

How to create a CUDA graph

- The definition can be done in two ways:
  - by using the graph API or
  - by capturing a stream.
- Using the graph API is the most powerful and expressive of the two.
- Capturing a stream is the more convenient and fast way.

The graph API

- A graph object can be constructed via:

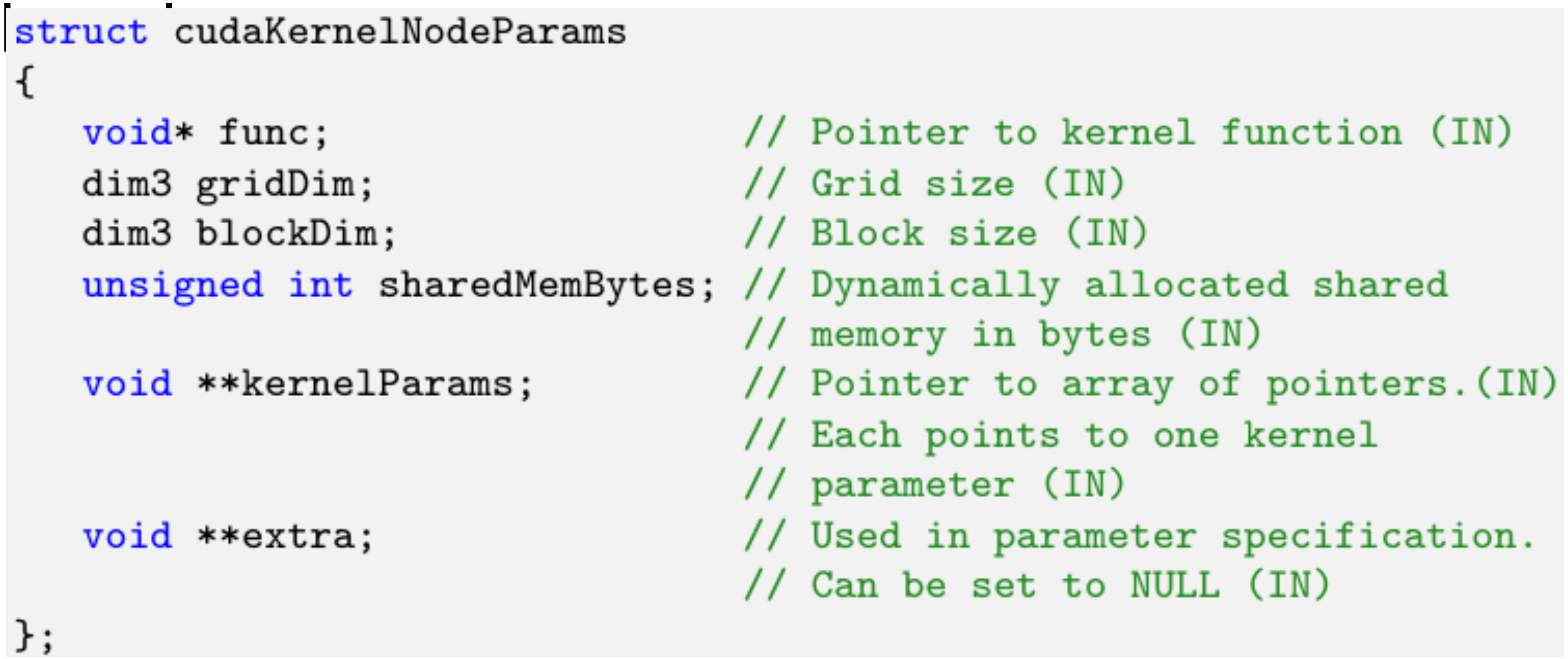
```
cudaError_t cudaGraphCreate (
    cudaGraph_t *pGraph,  // Graph object to be initialized (IN/OUT)
    flags );              // Creation flags, must be 0
```

- Subsequently, we can add a kernel node with:

```
cudaError_t cudaGraphAddKernelNode(
   cudaGraphNode_t *pGraphNode,   // Address of node object to be
                                  // created (IN/OUT)
   cudaGraph_t graph,             // Graph object to add the node
                                  // to (IN)
   const cudaGraphNode_t *pDepend,// Array of pointers to other node
                                  // objects that correspond to
                                  // operations that preceed this
                                  // node (IN)
   size_t numDependencies,        // Size of pDepend array (IN)
   const cudaKernelNodeParams *params); // Pointer to a structure
                                  // that specifies how the launch
                                  // will take place (IN)
```

The graph API (cont.)

- The last parameter to `cudaGraphAddKernelNode` encapsulates all information needed for a kernel



```
struct cudaKernelNodeParams
{
    void* func;                 // Pointer to kernel function (IN)
    dim3 gridDim;               // Grid size (IN)
    dim3 blockDim;              // Block size (IN)
    unsigned int sharedMemBytes; // Dynamically allocated shared
                                // memory in bytes (IN)
    void **kernelParams;        // Pointer to array of pointers.(IN)
                                // Each points to one kernel
                                // parameter (IN)
    void **extra;               // Used in parameter specification.
                                // Can be set to NULL (IN)
};
```

Adding a memory transfer node

- The corresponding function is called `cudaGraphAddMemcpyNode`:

```
cudaError_t cudaGraphAddMemcpyNode(
   cudaGraphNode_t *pGraphNode,  // Address of node object to be
                                 // created (IN/OUT)
   cudaGraph_t graph,            // Graph object to add the node
                                 // to (IN)
   const cudaGraphNode_t *pDepend,// Array of pointers to other
                                 // node objects that correspond
                                 // to operations that preceed
                                 // this node (IN)
   size_t numDependencies,       // Size of pDepend array (IN)
   const cudaMemcpy3DParms *params);// Pointer to a structure that
                                 // specifies how the transfer will
                                 // take place (IN)
```

Adding a memory transfer node

- The transfer information is placed in the last parameter:

```
struct cudaMemcpy3DParms {
   cudaArray_t srcArray;          // Source address
   struct cudaPos srcPos;         // Starting position
   struct cudaPitchedPtr srcPtr; // Source address alternative
   cudaArray_t dstArray;          // Destination address
   struct cudaPos dstPos;         // Starting position in destination
   struct cudaPitchedPtr dstPtr; // Destination address alternative
   struct cudaExtent extent;      // Size of memory to transfer
   enum cudaMemcpyKind kind;      // Direction of copy
};
```

Adding a memory transfer node (cont.)

- The components of the `cudaMemcpy3DParms` structure make the use of `cudaGraphAddMemcpyNode` tedious.
- CUDA provides a set of helper functions for creating these components:

```
static __inline__ __host__
 struct cudaPitchedPtr make_cudaPitchedPtr(
    void *d,     // Pointer to memory (IN)
    size_t p,    // Pitch of allocated memory in bytes (IN)
    size_t xsz, // Logical width of allocation in elements (IN)
    size_t ysz);// Logical height of allocation in elements (IN)
//****************************************
// Returns a cudaPos instance for specifying a location offset in 1D, 2D and 3↩
     D arrays
static __inline__ __host__
 struct cudaPos make_cudaPos(
    size_t x,    // Offsets in the x, y and z dimensions (IN)
    size_t y,
    size_t z);
//****************************************
// Returns a cudaExtend instance for specifying a 1D, 2D or 3D block
// of data. Parameters are in bytes if the block is not a cudaArray
static __inline__ __host__
 struct cudaExtent make_cudaExtent(
    size_t w,    // Block width (IN)
    size_t h,    // Block height (IN)
    size_t d);   // Block depth (IN)
```

Graph dependencies

- The dependencies can be added at a later stage using this function, which can be called multiple times:

```
// Connect graph nodes with dependency edges
cudaError_t cudaGraphAddDependencies(
   (cudaGraph_t graph,           // Target graph
   const cudaGraphNode_t *from, // Array of "source" nodes
   const cudaGraphNode_t *to,   // Array of "target" node
   size_t numDependencies);     // Size of "from" and "to" arrays
```

Graph instantiation and execution

- Instantiation is done with:

```
// Returns an instantiation of a graph, ready for execution
cudaError_t cudaGraphInstantiate(
   cudaGraphExec_t *pGraphExec, // Pointer to executable graph to be
                                // created (IN/OUT)
   cudaGraph_t graph,           // Graph (IN)
   cudaGraphNode_t *pErrorNode, // Pointer to a graph node object
                                // that will be made to point to the
                                // graph node that caused the error
                                // in case of failure (IN/OUT)
   char *pLogBuffer,            // Buffer for storing diagnostic
                                // messages(IN)
   size_t bufferSize);          // Size of pLogBuffer (IN)
```

- Launching requires a stream:

```
// Inserts an executable graph in a stream
cudaError_t cudaGraphLaunch(
   cudaGraphExec_t graphExec, // Exec. graph (IN)
   cudaStream_t stream);      // Stream in which to launch (IN)
```

Graph capture

- This is equivalent to "recording" the work submitted to one or more streams.
- Once a stream is placed in "capture" mode, all subsequent jobs requests are recorded but not executed.

```
cudaError_t cudaStreamBeginCapture(
    cudaStream_t stream,           // Stream to capture (IN)
    cudaStreamCaptureMode mode); // Capture mode (IN)
```

- Capture ends with:

```
cudaError_t cudaStreamEndCapture(
    cudaStream_t stream,   // Stream that is being captured (IN)
    cudaGraph_t *pGraph); // Pointer to graph object to receive the
                          // captured graph (IN/OUT)
```

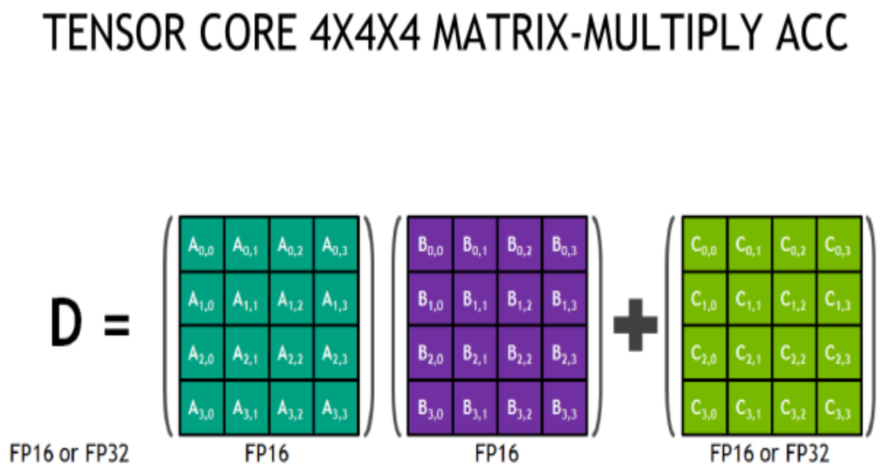
Tensor cores

The tensor core is a new type of processing core that performs a type of specialized matrix math, suitable for deep learning and certain types of HPC.

Tensor cores perform a fused multiply add, where two 4 x 4 FP16 matrices are multiplied and then the result added to a 4 x 4 FP16 or FP32 matrix. The result is a 4 x 4 FP16 or FP32 matrix.

NVIDIA refers to tensor cores as performing mixed precision math, because the inputted matrices are in half precision but the product can be in full precision.

As it so happens, the math that tensor cores do is commonly found in deep learning training and inferencing.



NVLINK

Increasing compute demands in AI and high-performance computing (HPC)—including an emerging class of trillion-parameter models—are driving a need for multi-node, multi-GPU systems with high-speed communication across GPUs.

For this, a fast, scalable interconnect is needed.