



ECE 508

Manycore Parallel Algorithms

Lecture 9: Dynamic Refinement Algorithms

Background

- load balancing
 - discussed briefly in 408
 - used padded and transposed CSR for SpMV
 - overflow non-zeroes done as COO on CPU
 - in that problem, row lengths known in advance
- saw more dynamic versions earlier in 508
 - dynamic extraction of frontiers for BFS
 - variable node degree for triangle counting

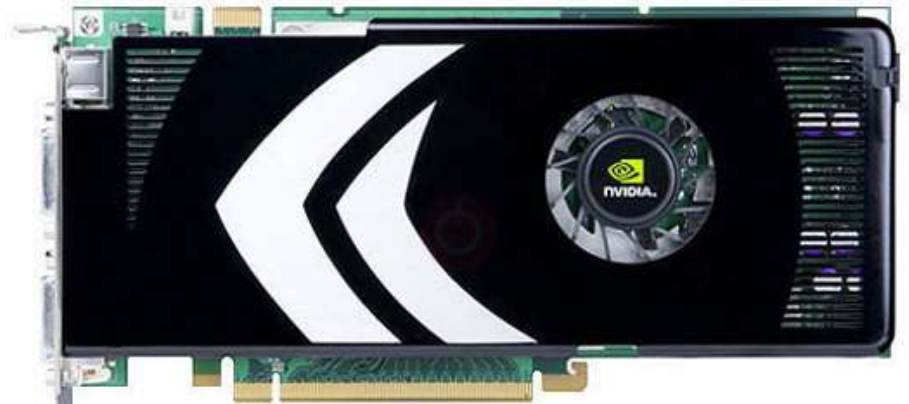
Objective

- to understand the need for dynamic refinement
- to understand dynamic parallelism in CUDA
 - fork-join style parallelism
 - device-side kernel launch semantics
 - performance considerations
 - aggregation and future trends

CUDA Did Not Drive GPU Development

The first version of **CUDA**

- **was** an interface
designed to leverage
existing GPU hardware
- by parallelizing iterations of a loop
with known trip count.



Even the idea of treating the GPU as a thread pool was absent from early explanations.

GPU Architecture Evolved for Graphics

- To the extent that the **hardware**
 - was modified at all to support CUDA,
 - it was only **to support execution of a single kernel.**

For example, the first GPGPUs didn't even fully support IEEE floating-point!

Algorithm Development Eliminates Regularity

Few applications offer such regularity.

Algorithms in most fields tend to **evolve**

- from regular approaches
- **to more control-intensive,
data-dependent approaches**
- until the constant factors
- outweigh the gains in complexity.

Dense Matrix Multiply, Anyone?

In the **1980s**, **Japanese supercomputers** regularly **outperformed Cray** supercomputers **on dense matrix multiplication**.

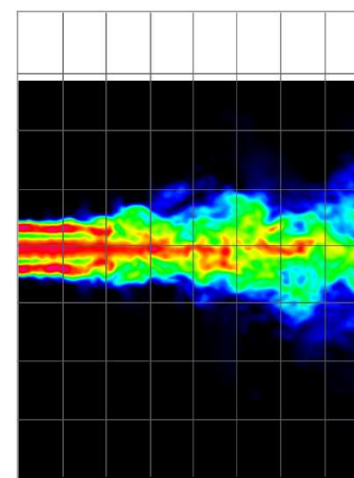
Neither Cray nor the US national labs that used Cray particularly cared, since most computational science codes no longer relied on dense matrices.

Cray-1



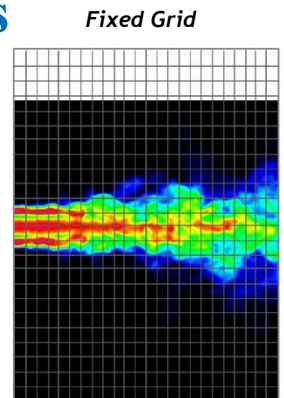
Adaptive Mesh Refinement for CFD Codes

For example, in turbulence simulation / computational fluid dynamics (CFD), **adaptive mesh refinement (AMR) codes** were invented in the mid-80s and **in** fairly wide **use by the 90s**.



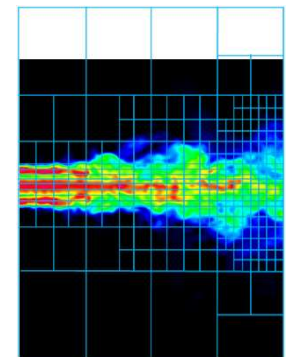
Initial Grid

*Statically assign
conservative
worst-case grid*



Fixed Grid

*Dynamically assign
performance where
accuracy is required*



Dynamic Grid

HPC Was Skeptical About GPU Use

**“Really good at
multiplying dense matrices”**

was not much of a selling point.

Heterogeneity and architecture-dependent code extremely unattractive.

That’s why **Wen-meï had
to work hard**

- to convince NSF **to consider**
- **adding GPUs** to Blue Waters
as an option.



Over Time, HPC Has Helped Evolve GPUs

Fortunately, the **HPC community**
behind Blue Waters

- **had a lot of experience**
 - in **using regular hardware**
 - **to execute irregular algorithms**
 - (such as sparse matrix formats),
- which **helped** to **shape** the **evolution**
of both CUDA and **GPUs**.

Early CUDA Made Libraries Difficult

The **early CUDA model** also **posed challenges for library code.**

- BLAS (Basic Linear Algebra Subroutines) has been a library for decades
- BLAS calls should execute natively.
- On a GPU, that means as kernels.

But ... how?

More Flexible Models Hard to Adapt to CUDA

**Many more flexible models existed,
but few worked well in CUDA.**

For example, work queues:

- drop work (continuations) into a queue.
- Processors pull out work and execute it,
- perhaps making more work in the process.

Work Queues Don't Work with CUDA

- Work queue operations require **global atomics**, which are **slow**, particularly under contention.
- Weak GPU processors take **significant time for** dynamic context / **code selection**.
- **Can't adapt** the **parallelism profile** to the work:
 - synch. only within thread blocks, but
 - may need a different number of threads, and
 - no way to restrict efficiently to a subset.
- **Poor resource use profile**, since register count and shared memory per block is max over all possible codes.

Another Idea: Heterogeneous Work Queue

Another: **continuations launched from CPU.**

Insert “kernel launch” into work queue.

- CPU manages new launches, so
- **arguments move back and forth across PCIe.**

Similar to event-based distributed systems:

- **complex and error-prone** state machines, with
- all state packed into continuation (sometimes by reference).

Ping-Pong GPU-CPU Also Painful

- Also **need to break code** across “launch”:
 - do pre-launch work,
 - “launch” kernel (by writing into queue), then
 - do post-launch work (as a separate kernel).
- **No shared context**
 - **between pre- and post-launch kernels**
 - (registers, shared memory, or even thread mapping).

GPU Hardware Also Designed for One Kernel

Early GPU hardware optimized for a single kernel.

Supporting **multiple** concurrent **kernels**

- may **lead to fragmentation** in register file and shared memory, **and**
- **undermines resource-based strategies.**

For example,

- a fixed pool of threads per SM
- is badly imbalanced if one of the SMs
- is used by another kernel.

GPU Software Offloaded onto CPU

Kernel/memory management functions

- **executed by CPU** (in driver code), so
- **any** type of **dynamic activity**,
 - such as allocating memory
 - to hold results with data-dependent size
 - also **required CPU support**.

Need Fast CPUs Just to Keep GPUs Busy

As a result, **executing** many **kernels requires** significant support from **the CPU**.

The Mont-Blanc supercomputer combined

- **low-power ARM cores**
- with GPUs.

The CPUs **could not feed the GPUs** fast enough!



More Dynamic Models Must be Self-Contained

To enable more dynamic models,

- these **functions had to be duplicated** on the GPU,
- sometimes **adding hardware support** for efficiency,
- and sometimes developing ways to **address serialization issues** (such as memory allocation).

CUDA and GPU Hardware Will Continue to Evolve

Not surprisingly, many **researchers**

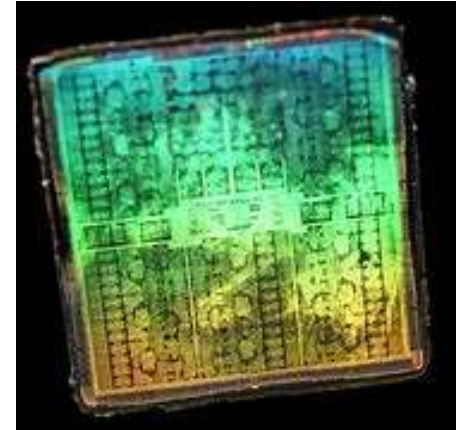
- **declared** CUDA and **GPUs to be non-competitive**
- **for** a range of **irregular algorithms**.

If you find yourself **in such a position**,

- **think about how CUDA/GPU** hardware
- **could evolve** to support your needs.

Kepler+ Generation Supports Fork-Join Parallelism

- **Kepler generation added support** for dynamic parallelism, including
 - necessary **changes to GPU hardware,**
 - **support in** NVIDIA's **runtime** and
 - **extension of** many CUDA **APIs to kernel code.**
- Similar to fork-join parallelism,
 - “fork” is a kernel launch, and
 - “join” is a stream or device synchronization call.



Kernels are Kernels: Same Syntax, Same Rules

NVIDIA did a fairly **good job**,

- **extending** the **CUDA** abstractions
- **to** support **launching kernels from kernels**.

Syntax is identical!

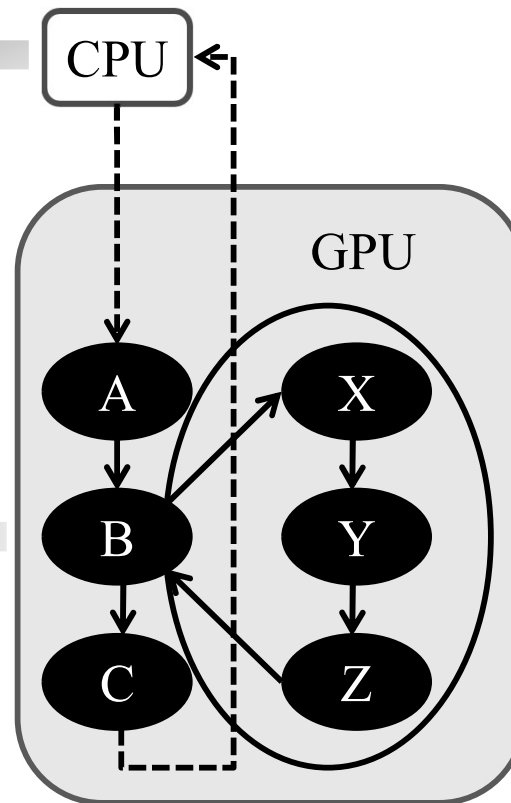
Thread blocks **share only global memory**,

- so while arguments are passed from parent to child,
- other data must go through global memory, and
- scheduling is easier to manage
(and as hard to reason about as before).

Kernel Launches Same in Host and Device Code

```
int main() {  
    float *data;  
    setup(data);  
  
    A <<< ... >>> (data);  
    B <<< ... >>> (data);  
    C <<< ... >>> (data);  
    cudaDeviceSynchronize();  
  
    return 0;  
}
```

```
__global__ void B(float *data)  
{  
    do_stuff(data);  
  
    X <<< ... >>> (data);  
    Y <<< ... >>> (data);  
    Z <<< ... >>> (data);  
    cudaDeviceSynchronize();  
  
    do_more_stuff(data);  
}
```



Note: Each thread launches X, Y, and Z!

Recall Full Kernel Launch Syntax

kernel_name<<< Dg, Db, Ns, S >>> ([kernel arguments]);

- **dim3 Dg** specifies **grid dimensions**,
- **dim3 Db** specifies **thread block dimensions**,
- **size_t Ns** specifies the number of **bytes of shared memory** per thread block (needed for runtime sizing of tiles), and
- **cudaStream_t S** specifies **stream used** for launch.

We'll come back to using streams later.

As Before, Wait for Kernel to Finish

What about memory consistency?

Historically, a **kernel's results**

- were guaranteed to be **in global memory**
- **only after** the kernel **terminated**.

This choice allows use of GPU **caches**

- to **hold modified results**
- that may or may not be visible
- to other thread blocks in the kernel
- and to the CPU.

Consistency Guaranteed at Fork and at Join

To support mid-kernel launches

- without forcing all data into kernel arguments,
- a **consistent view** of global memory is **guaranteed**
 - **when** a **child kernel** is **launched** (fork) **and**
 - **when** a child kernel is **synchronized** (join).

Either of these actions thus
implies flushing cached data.

(Children implicitly synchronized at end of kernel.)

Do Not Reason About Block Scheduling

What about scheduling?

As with other thread block scheduling issues, CUDA allows for **no reasoning about inter-block scheduling**.*

- In particular, a **child** kernel **may not execute at all until** the **parent** kernel **synchronizes** with it.
- **Or it may**. Is there a prioritization scheme? Probably.

*I think this may have been weakened slightly for individual kernels launched under tight controls to ensure that blocks are scheduled concurrently, but NVIDIA manual makes the above limitations clear for dynamic parallelism.

No Save/Restore of Resources, So Locked Down

What about resources?

Currently, **parent kernel resources** are

- effectively **locked down**
- **while** any **descendant** kernels **execute**.

Why? For explicit mid-parent-kernel synchronization,

- **reclaiming** the parent's **resources means**
- **saving and restoring** registers and shared memory
- used by the parent kernel
- across the synchronization point.

Always Need to Synchronize with Ancestors

For implicit **end-parent**-kernel **synchronization**,

- **need for** protecting **state vanishes**, and
- requires merely designing a method to enforce synchronization with ancestor kernels/host code.

Our Bézier code, for example,

- launches a child kernel that must be complete
- before the host code collects the results.

Resource Reclamation May Be Done in Future

My guess is that it hasn't happened

- because launches occupy slots in hardware queues, and
- the structures are not flexible enough to accommodate
 - insertion of a large number of orphaned descendants
 - when a child terminates with live children.

Perhaps resource reclamation (or even preemption, in the case of mid-kernel synchronization) **will be available in future generations.**

New Kernel Launches May Fail, So Always Check

Can resource constraints lead to deadlock?

No, that's what runtime errors are for!*

In an overburdened system, **new launches may fail** due to

- **too many kernels at once** (now fails over to software),
- **too deep a nesting level** (to avoid infinite recursion, but actual limit can be adjusted), or
- **all resources in use** and parent tries to wait for child (which would otherwise be a deadlock).

*I haven't tested this case, but presumably NVIDIA chose giving a runtime error over a deadlock.

Let's Examine a Few Examples

Let's do some examples:

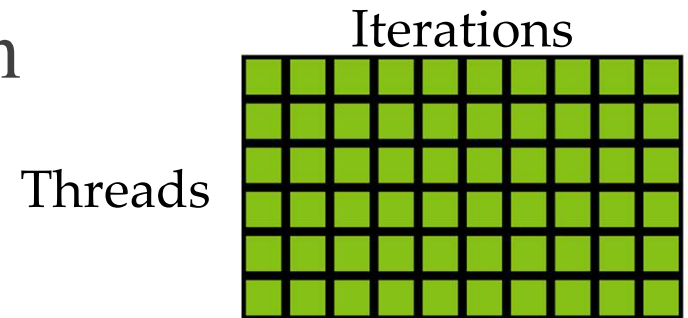
- generic Code Similar to SpMV,
- Bézier curves, and
- building a quadtree.

Uniform Load and No Dynamic Execution

- fixed work per thread

```
__global__ void kernel  
(int ct, int stride, float* data, float* more)  
{  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    doSomeWork (data[i]);  
    for (int j = 0; j < count; ++j) {  
        doMoreWork (more[i + j * stride]);  
    }  
}
```

- similar to ELL format execution
(padded and transposed CSR)

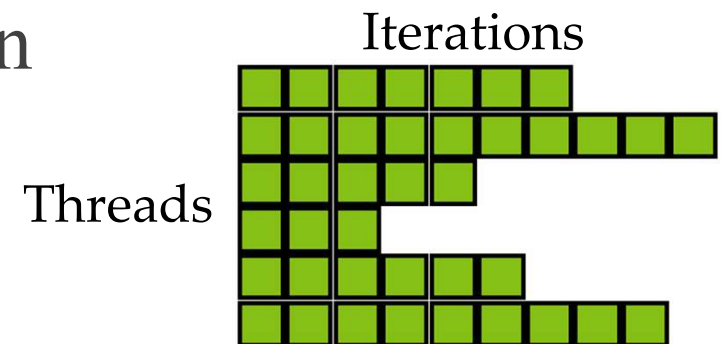


Non-Uniform Load and No Dynamic Execution

- variable work per thread

```
__global__ void kernel  
(int* start, int* end, float* data, float* more)  
{  
    int i = blockIdx.x * blockDim.x + threadIdx.x;  
    doSomeWork (data[i]);  
    for (int j = start[i]; j < end[i]; ++j) {  
        doMoreWork (more[j]);  
    }  
}
```

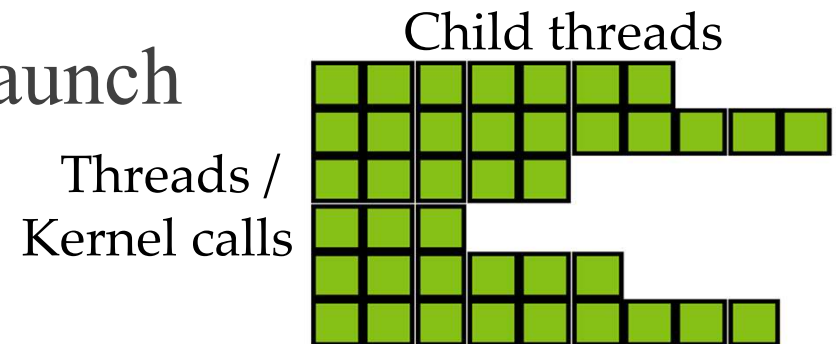
- similar to CSR format execution



Non-Uniform Load with Dynamic Execution

```
__global__ void kernel
(int* start, int* end, float* data, float* more)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    doSomeWork (data[i]);
    kernel_child <<<((end[i]-start[i]+255)/256),256>>>
                (start[i], end[i], more);
}
```

- replace loop with kernel launch



Child Kernel for Dynamic Execution

```
__global__ void kernel_child
(int start, int end, float* more)
{
    int j = start + blockIdx.x * blockDim.x + threadIdx.x;
    if (j < end) {
        doMoreWork(moreData[j]);
    }
}
```

Pros and Cons of Dynamic Parallelism

Without dynamic parallelism, suffers from

- **idle resources** (imbalance within warps) **and**
- **load imbalance** (imbalance across warps).

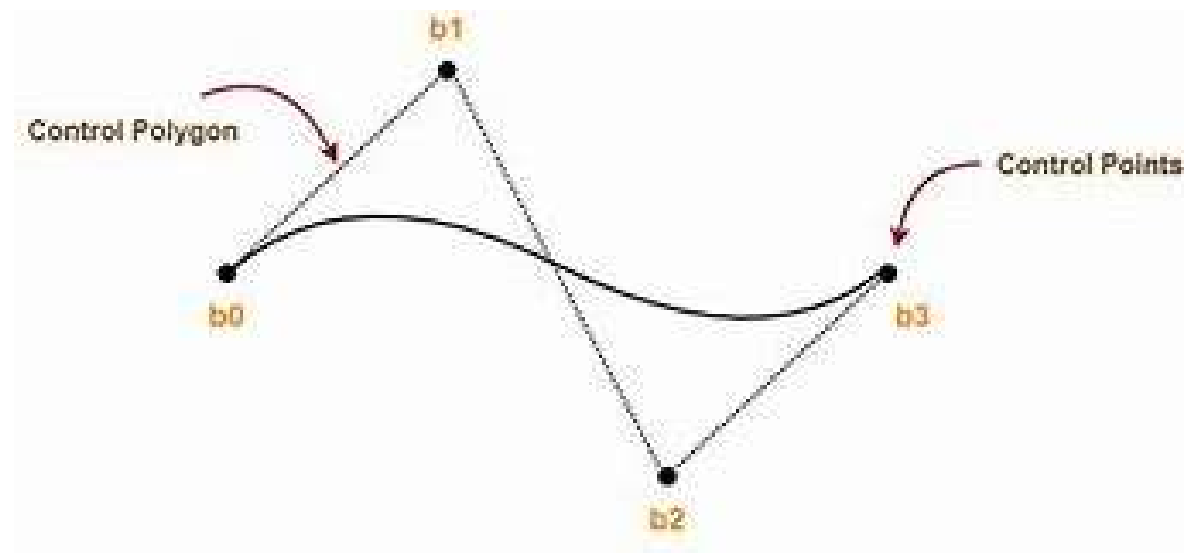
With dynamic parallelism,

- has more parallelism and **less load imbalance**, but
- needs **more index calculations**,
- costs **more** for **kernel launch overhead**, and
- still has **some idle resources** (boundary warps).

Drawing Bézier Curves a More Complex Example

Bézier curves **used to draw smooth curves**:

- based on control points, **compute curvature**, then
- **compute tessellation points** for rendering
(number of points depends on curvature).



Two Points Give a Line (First Order)

Number of control points
defines the **order** of the curve.

Two points (P_0 and P_1)
gives **first order**, a line:

$$\text{For } t \in [0, 1], B(t) = (1 - t)P_0 + tP_1.$$

Three Points Give a Quadratic Curve (Second Order)

Three points (P_0 , P_1 , and P_2)
gives **second order**, a quadratic curve.

First, interpolate linearly to find endpoints,
then interpolate endpoints to find curve:

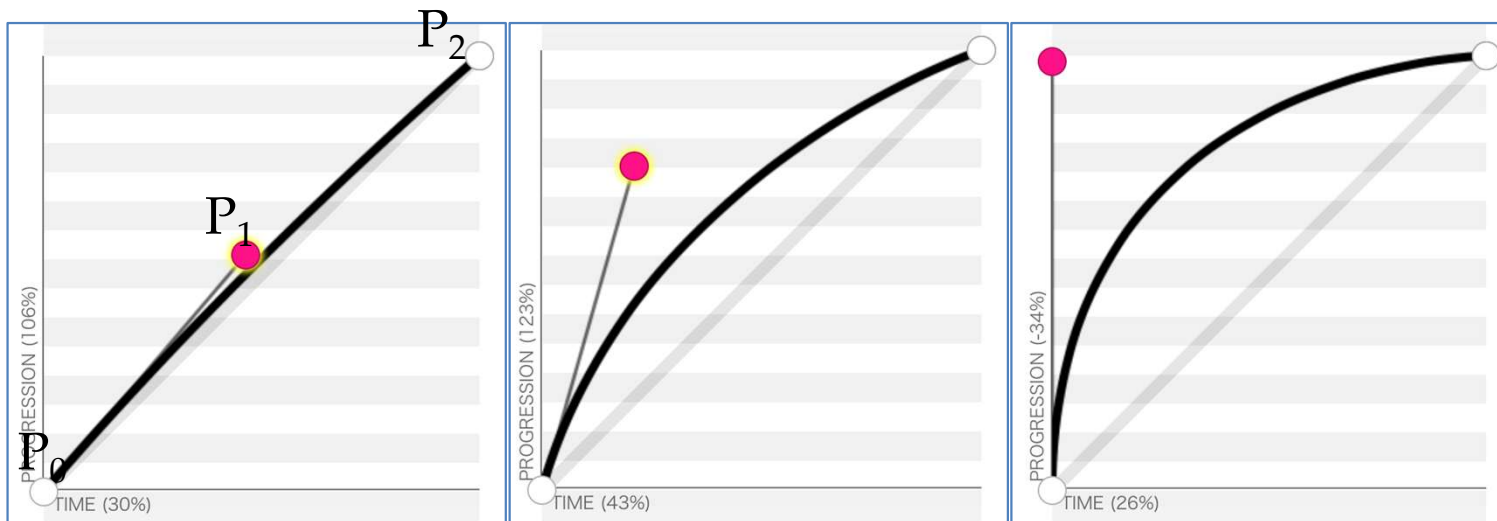
For $t \in [0, 1]$,

$$\begin{aligned} B(t) &= (1 - t)[(1 - t)P_0 + tP_1] + t[(1 - t)P_1 + tP_2] \\ &= (1 - t)^2 P_0 + 2t(1 - t)P_1 + t^2 P_2. \end{aligned}$$

More Curvature Requires More Points to Draw Well

What is curvature?

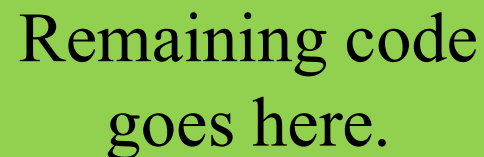
Related to distance from P_1 to P_0 - P_2 line.



Computing Curves without Dynamic Parallelism

Without dynamic parallelism, **each thread block computes** tessellation **points for one Bézier curve**.

```
__global__ void computeCurves  
    (Curve* curves, int nCurves)  
{  
    int cidx = blockIdx.x;  
    if (cidx < nCurves) {  
        ...  
    }  
}
```



Remaining code
goes here.

Find Curvature and Choose Number of Points

```
int curvature =
```

Compute curvature.

```
16.0f * findCurvature (curves + cidx);
```

```
int nPoints =
```

Choose and record number of
tessellation points.

```
min (max (curvature, 4) , 32) ;
```

```
curves[cidx].nPoints = nPoints;
```

Compute Tessellation Points for Each Curve

```
for (int inc = 0; inc < nPoints;  
    inc += blockDim.x)
```

Use all threads to
compute points.

```
{
```

```
    int idx = inc + threadIdx.x;
```

Use unique
index.

```
    if (idx < nPoints) {
```

```
        ...
```

Point computation
goes here.

```
    }
```

```
}
```

Point Computation With C++ Syntax

```
float t = (float)idx / (float)(nPoints - 1);  
float omt = 1.0f - t;
```

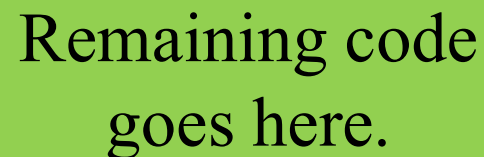
Find t and $(1 - t)$.

```
// Each point is a 2-tuple of floats  
// with appropriate operators defined.  
curves[cidx].vertexPos[idx] =  
    omt * omt * curves[cidx].CP[0] +  
    2.0f * t * omt * curves[cidx].CP[1] +  
    t * t * curves[cidx].CP[2]
```

Computing Curves with Dynamic Parallelism

With dynamic parallelism, **each thread** (not block) **computes** tessellation **points for one Bézier curve**.

```
__global__ void computeCurves  
    (Curve* curves, int nCurves)  
{  
    int cidx = blockDim.x * blockIdx.x +  
                threadIdx.x;  
    if (cidx < nCurves) {  
        ...  
    }  
}
```



Remaining code
goes here.

Find Curvature and Choose Number of Points

```
int curvature  
    16.0f * fi  
  
int nPoints  
    min (max (  
curves[cidx].nPoints = nPoints,
```

Same as
before.

are.
+ cidx) ;
ord number
n points.

Replace Tessellation Loop with Kernel Launch

cudaMalloc

```
( (void**) &curves[cidx].vertexPos,  
  curves[cidx].nPoints *  
  sizeof (*curves[cidx].vertexPos) );
```

curveChild

```
<<< ( (curves[cidx].nPoints+31) / 32 ) , 32 >>>  
(cidx, curves, curves[cidx].nPoints);
```


Allocating Device Memory in a Kernel

cudaMalloc in a kernel?

Yes, we want device memory.

The implementation

- uses a scan across threads
- to improve allocation speed.

Free Occurs After Data Created and Consumed

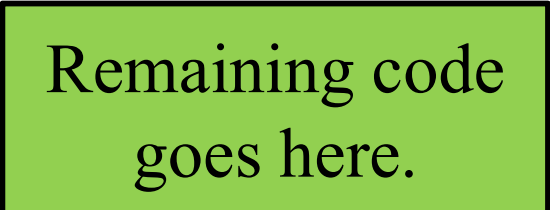
When is the memory freed?

By the host (CPU)

- using the stored pointer
- after **computeCurves** terminates, which is
- **after all curveChild kernels terminate.**

Child Kernel Computes One Tessellation Point

```
__global__ void curveChild
(Curve* curves, int cidx, int nPoints)
{
    int idx = blockDim.x * blockIdx.x +
              threadIdx.x;
    if (idx < nPoints) {
        ...
    }
}
```



Point Computation With C++ Syntax

```
float t = (float)idx / (float)(nPoints - 1);  
float omt = 1.0f - t;  
  
// Each point is a vector of floats  
// with approx 3 floats defined.  
Vec curves[cidx] = Vec(0.0f, 0.0f, 0.0f);  
Vec p = omt * curves[cidx].CP[0] +  
        2.0f * t * curves[cidx].CP[1] +  
        t * t * curves[cidx].CP[2];
```

Same as before.

“Launch Pool” Limits Pending Kernel Launches

That’s a lot of kernels!

Yes, so **be careful.**

CUDA supports with a **hardware queue**:

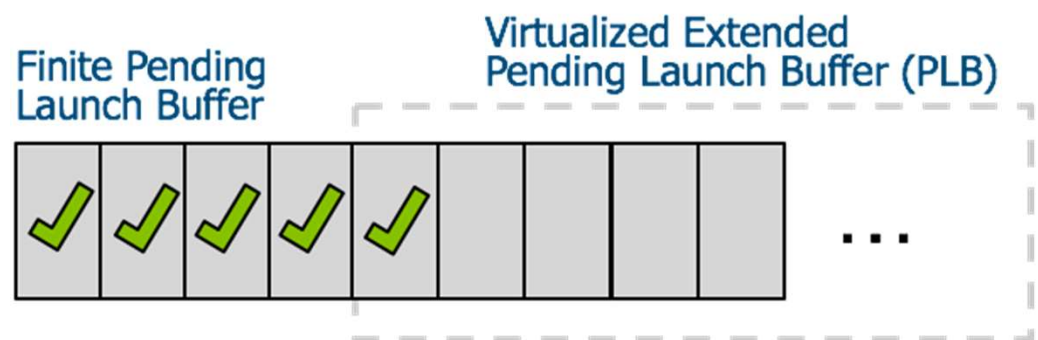
- **2048 entries**, called the launch pool.
- Before CUDA 6.0, overflows caused runtime errors.



“Launch Pool” Overflows into Software

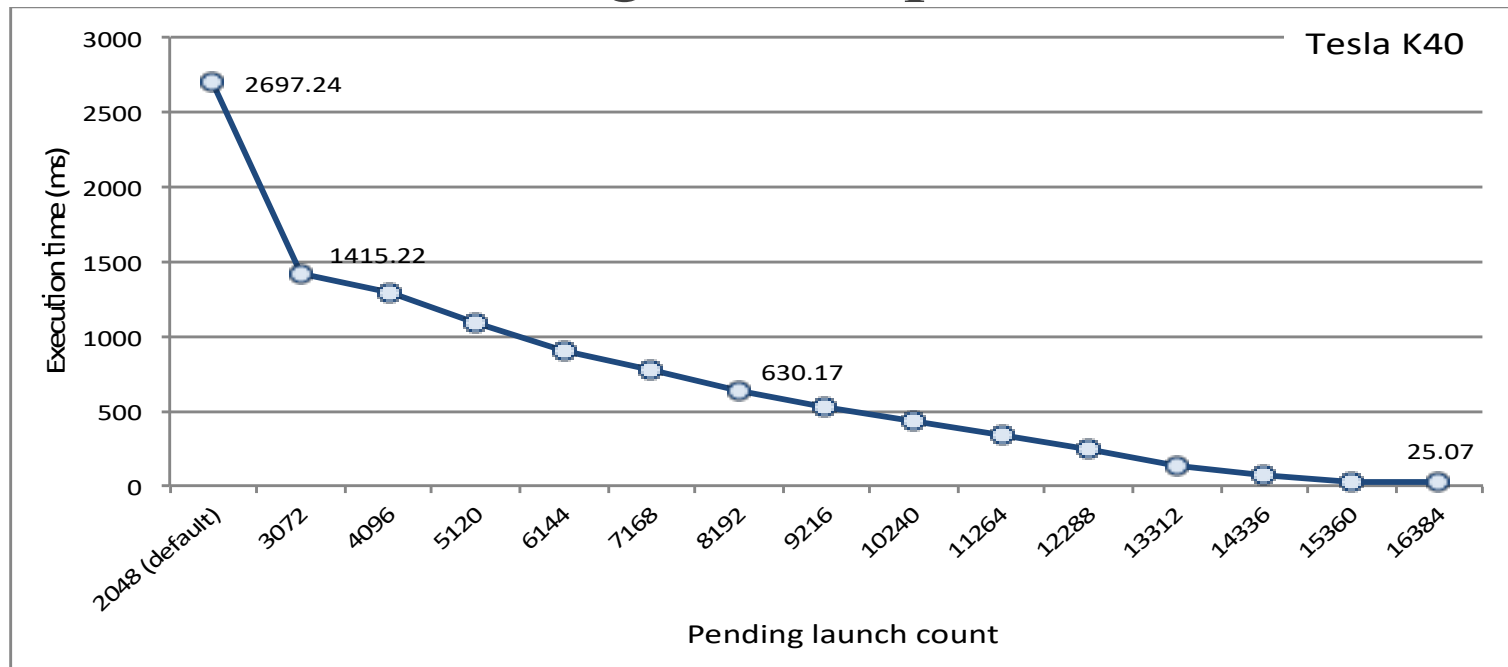
In CUDA 6.0,

- can increase **limit** by calling `cudaDeviceSetLimit` to raise `cudaLimitDevRuntimePendingLaunchCount`.
- Additional launches **overflow into software** queue.



Larger Launch Pool May Allow More Parallelism

Bézier kernel performance (16k lines) improves with increasing launch pool size.



Streams Must Track Many Kernels to Reclaim

Numbers in that example help us understand

- **why reclaiming resources is** a bit **challenging**:
- 16,384 kernels launched by **computeCurves**.

If **computeCurves** “finishes,” **stream on host**

- **must inherit 16,384 kernels** and
- wait for completion

to prevent host code

- **from reading** results **and freeing data**
- **before** child **kernels finish**.

Mid-Kernel Synchronization Prohibited at Depth

Another `cudaDeviceSetLimit` control

- **suggests** that **CUDA** does attempt
- (or at least **is planning**) to **reclaim resources**:
`cudaLimitDevRuntimeSyncDepth`
- limits the **depth at which a kernel**
- **can explicitly synchronize** with children.
- Kernels at deeper levels may **ONLY**
synchronize implicitly after finishing their code.

The limit **defaults to 2!**

(Hardware also limits max depth to around 24.)

Streams Can Limit Dynamic Parallelism

Streams are **also important** to dynamic parallelism. In 408, streams were introduced

- to parallelize GPU computation
- with PCIe communication.

In more sophisticated heterogeneous applications,

- many concurrent kernels may be launched
- to occupy the GPU.

**With dynamic parallelism,
streams can be a limiting factor.**

Kernels Launched in Default Stream Serialized

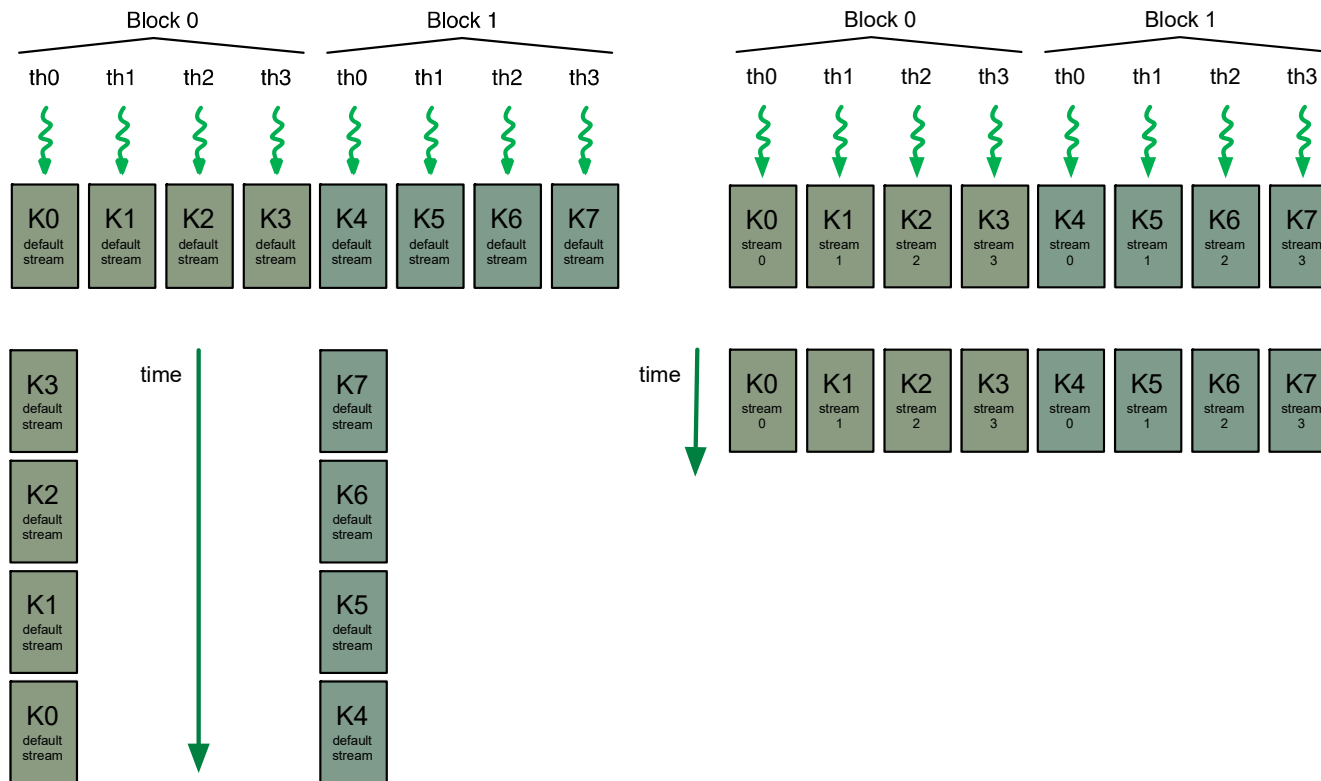
In particular,

- the **“default” stream is shared**
- **across** each **thread block**
- with dynamic parallelism.

Any **kernels** launched

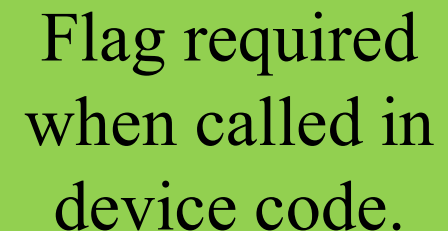
- by threads within a block
- are **implicitly serialized**
- by the default stream!

Illustration of Impact of the Default Stream



Child Kernel Launch Using an Explicit Stream

```
cudaStream_t stream;  
cudaStreamCreateWithFlags  
    (&stream, cudaStreamNonBlocking) ;  
  
... // cudaMalloc  
  
curveChild  
    <<< ((curves[cidx].nPoints+31)/32), 32,  
        0, stream>>>  
    (cidx, curves, curves[cidx].nPoints) ;
```



Flag required
when called in
device code.

After Use, Destroy the Stream

```
curveChild
```

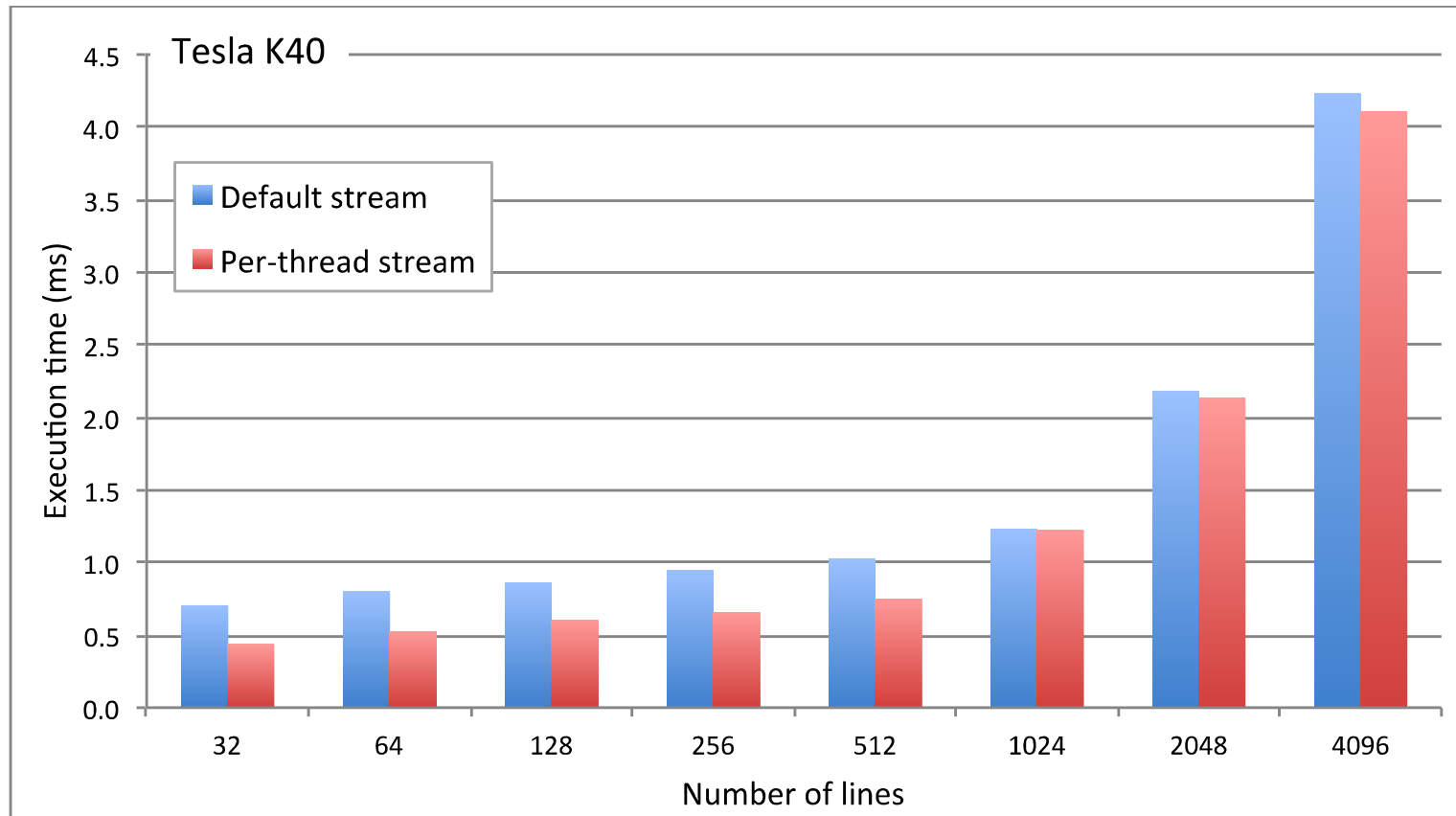
```
<<< ((curves[cidx].nPoints+31)/32), 32,  
      0, stream>>>  
(cidx, curves, curves[cidx].nPoints);  
cudaStreamDestroy(stream);
```



Don't panic!

Remember that **CUDA delays stream destruction until the last operation** on the stream has **completed**.

Streams Matter Most if GPU Underutilized



Or Use Compiler Flag for One Stream per Thread

Alternatively, tell the compiler!

Pass

- **--default-stream=per-thread**
- to NVCC to change to
- one default stream per thread
- (instead of a default stream per block).

Other Aspects of CUDA Dynamic Parallelism*

1. Children kernel thread blocks **may or may not run on same SM** as parent.
2. Each **thread** executing `cudaDeviceSynchronize` **waits for (at least) its own children** kernels to complete.
3. **Can order kernels** in different streams **using `cudaStreamSynch`**, but cannot use for timing or other purposes.
4. Currently, device code **can only launch to same GPU**.

*See Appendix D of the CUDA Programming Guide for more.

Dynamic Parallelism ... May Take Some Effort

Sadly, I wasn't able to

- get dynamic parallelism working
- in the context of the RAI labs using CMake.
- I haven't figured out why...

The error message from NVIDIA is “unknown error” (#30) iff my kernel launches a kernel.

Maybe the default params are set badly, but the kernel was trivial (1 block, 1 thread, empty code).

I Will Give You Example Code

Izzat sent me working code, which I was able to run by scrapping use of Cmake.

Pass these flags when compiling:

-O3 -dc --default-stream=per-thread

These flags when compiling AND when linking:

-gencode arch=compute_70,code=sm_70

And these flags when linking:

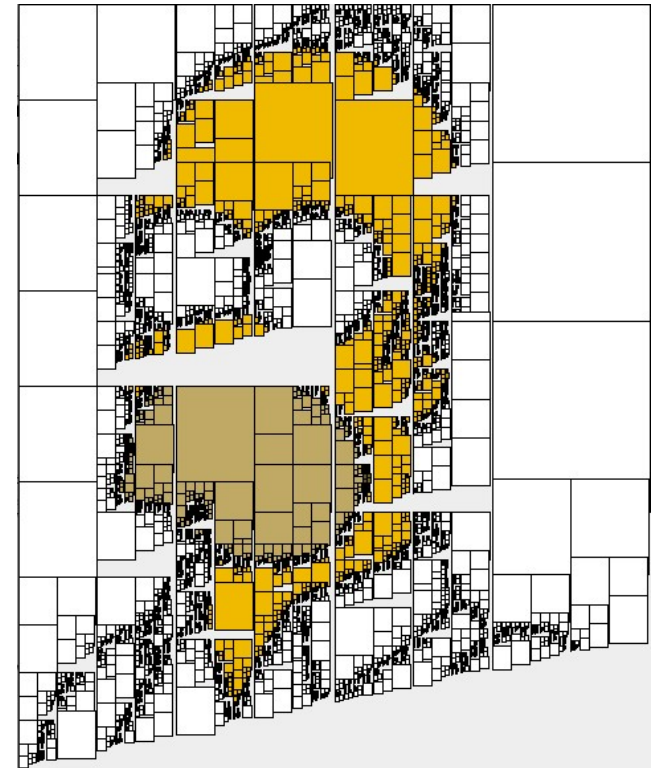
-lcudadevrt

(That's for Titan V—be sure that you use the EXACT values for your GPU. I'll add example to lab repo sometime soon.)

Next Example: Building a Quadtree

And now for something completely different!

- **Building quadtrees!**
- Quadtrees are **widely used in both scientific computing and in gaming.**
- For many years, they were created on CPUs and only used by GPU code.



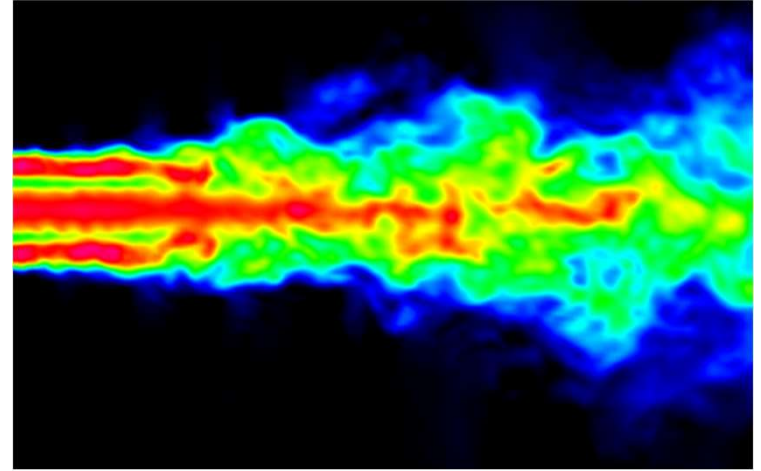
Quadrees Important for Spatial Sorting

As you probably know,

- a quadtree **serves to spatially sort 2D data**,
- **enabling** fast and simple **range-based computation**
- to support cutoff or other range-dependent formulae, including collision detection,
- **as well as spatially localized computation**, such as ray tracing.

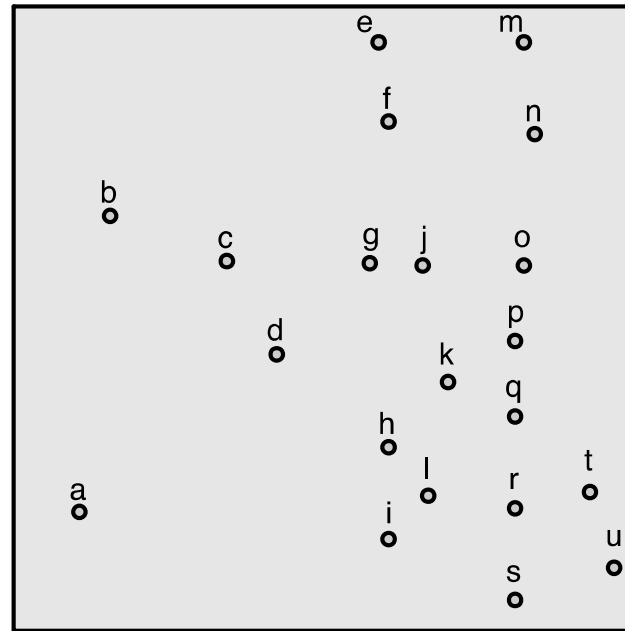
CFD Requires Repeated Spatial Sorting

- Being able to build quadtrees (and octrees, their 3D cousins) quickly on a GPU is **especially useful in** computational fluid dynamics (**CFD**) computations, **in which particles move** and spatial sorting must be performed repeatedly.
- Moving data over PCIe to build on the CPU and then moving the tree back to the GPU is particularly slow.



A Recursive Example: Quadtree Formation

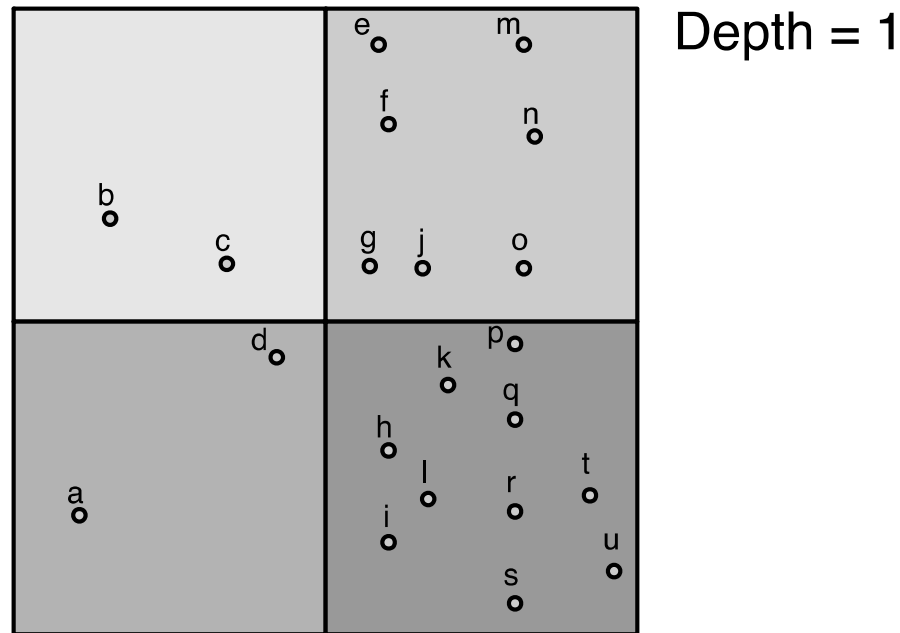
Partition 2D space by recursive division into quadrants.



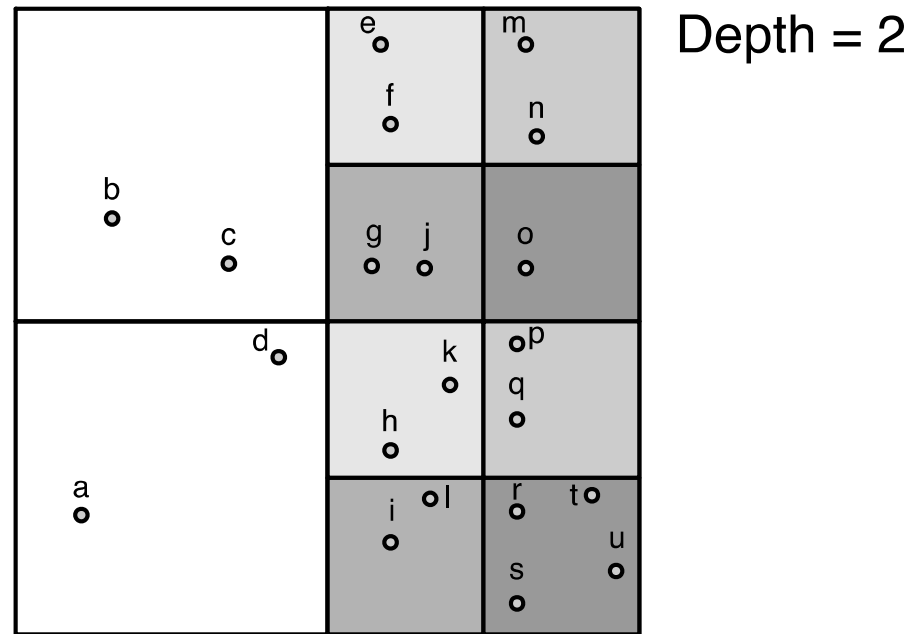
Depth = 0

Want maximum
of two nodes per
quadrant.

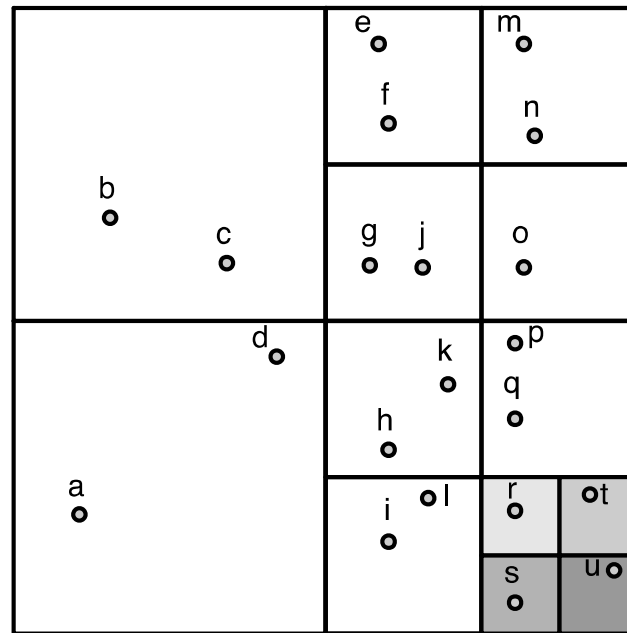
More than Two Points? Subdivide Around Center!



Each Quadrant May Need Further Subdivision



No Leaf Quadrant Has More than Two Nodes



Depth = 3

Current NVIDIA hardware limits recursive depth to 24.

Top-Down Dynamic Parallelism Used to Build

How do we build a quadtree in CUDA?

- **Top down, using** a complete set of nodes with fixed depth (a **complete 4-ary tree**).
- **Lower levels** of the tree are **built by separate**, dynamically-launched **kernels**.

Why use a complete tree structure?

Doing so allows **independent array access** to nodes at each level.

Arrays of Points are Double-Buffered

- At each level, points are reorganized into four subgroups (by quadrant).
- **Points are double-buffered.**

Why?

Same reason as we saw in 408 for scan algorithms:

- **eliminates** logical **write-after-read dependence**
- implied by reorganizing an array in place.

Subdivision Stops Based on Density

The **second buffer is** a **temporary** array:

- we want all points in one buffer,
- which sometimes means copying from second buffer back into first.

Once quadrants contain few enough points,

- **no further subdivision** is needed,
- no kernels are launched for those sections,
- and the data stay in place in the first buffer.

Should Use a Hierarchical Organization

For a large number of points, we want

- **multiple thread blocks**
- to repartition the points into quadrants,
- **to keep the GPU occupied.**

However, once we have

- enough concurrent kernels or few enough points,
- we want a single thread block.
- So the **kernel should be hierarchical.**

We Focus on the One-Block Kernel

But you saw such a thing with triangle counting.

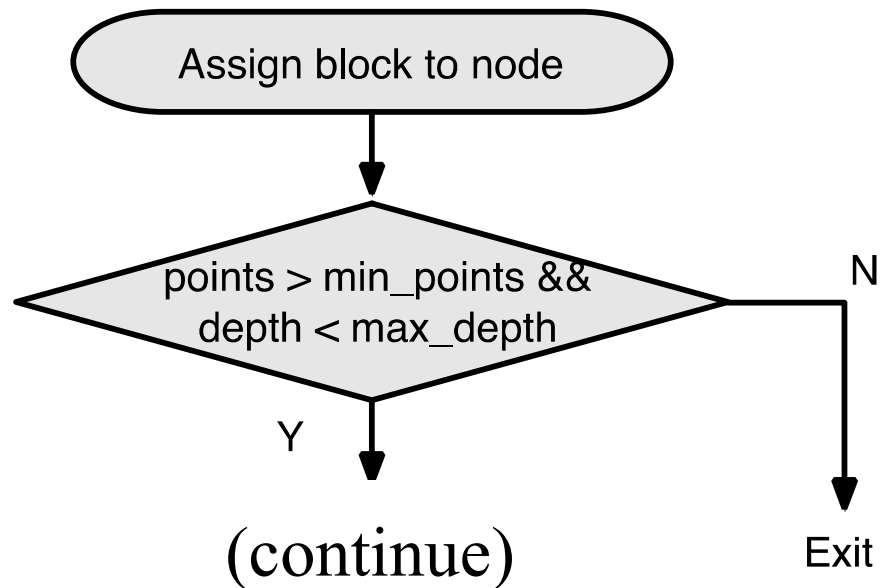
For simplicity,

- we'll **examine a one-block kernel**
- **that** recursively **launches itself with four blocks**
- (one per quadrant).

Recurse Until Few Enough Nodes or Max Depth

Let's **start with a flow chart** for the kernel.

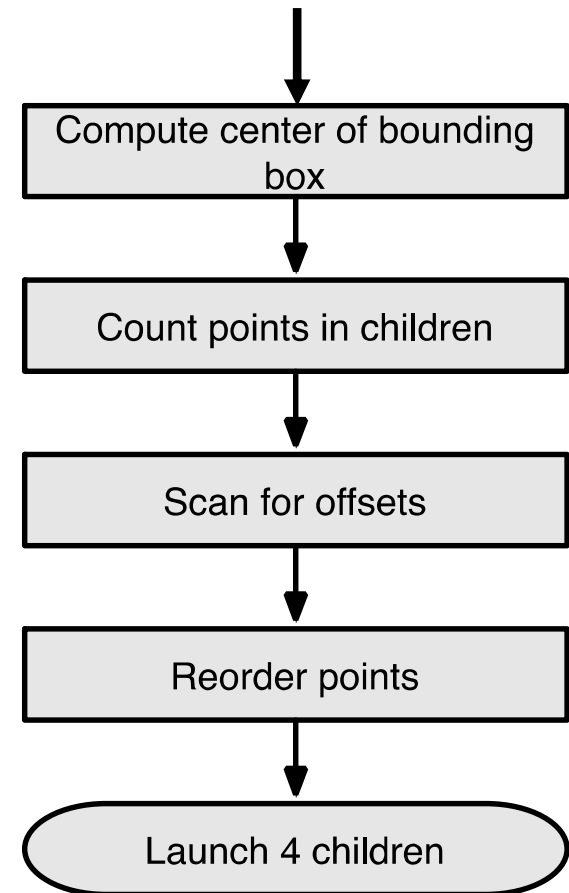
- A **block** is responsible for **building a node** in our complete quadtree.
- Recursion **stops when**:
 - we have **few enough points, or**
 - we reach **max depth**.



Recursion Requires Work to Repartition Points

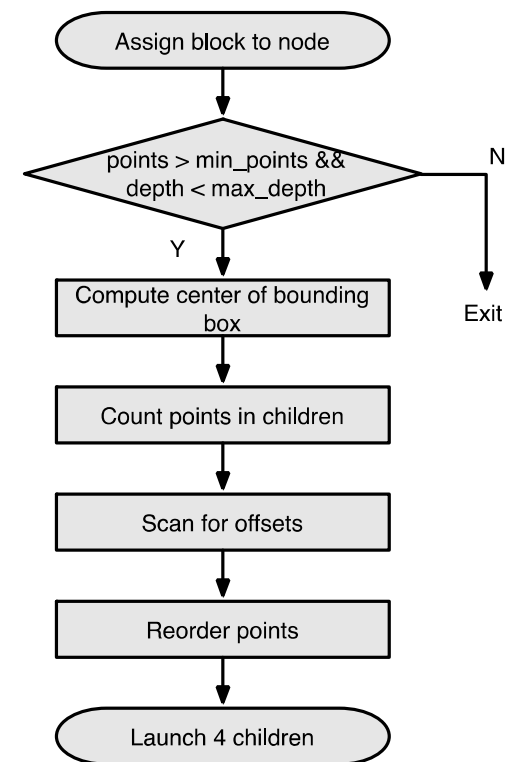
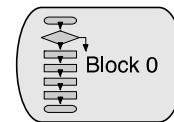
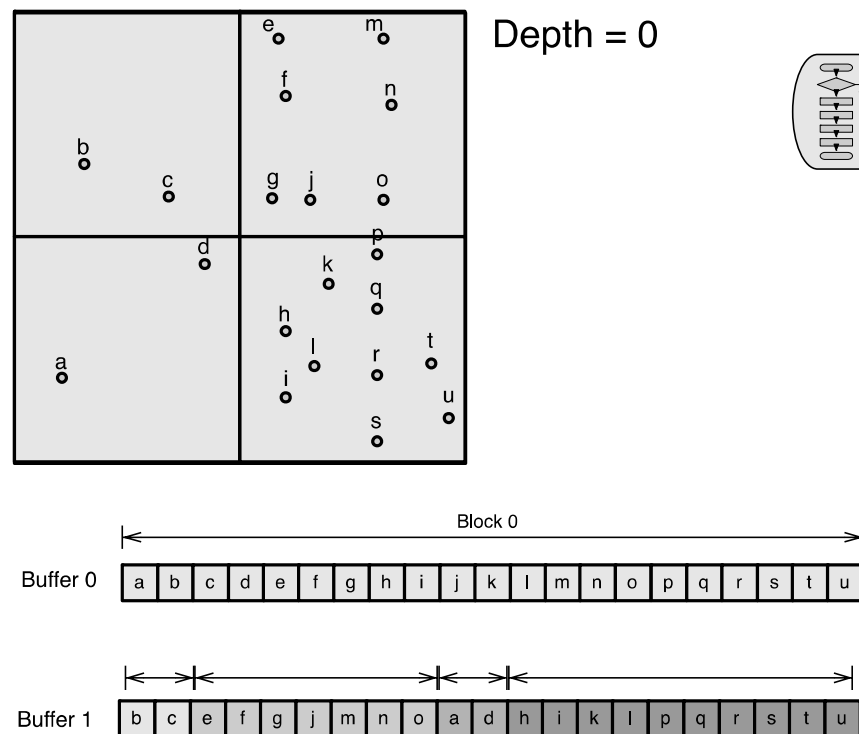
If recursion is necessary,

- **find center** of current node's bounding box
- **count** current node's **points** within each quadrant,
- **scan for** remapping **offsets** into the output buffer,
- **remap** the **points**, and
- **recurse with four** new **blocks**.



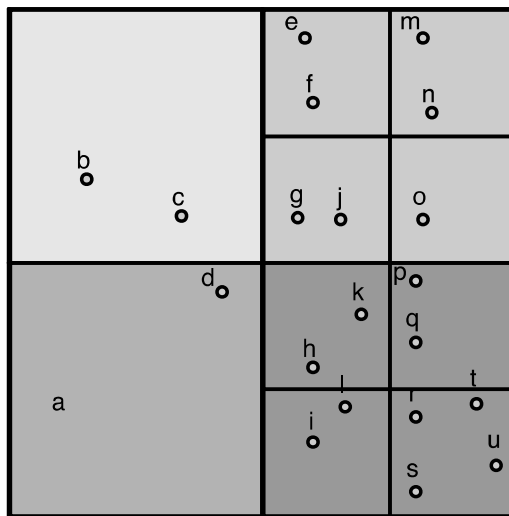
First Step: One-Block Kernel to Build Node 0

Block 0 launched from host.

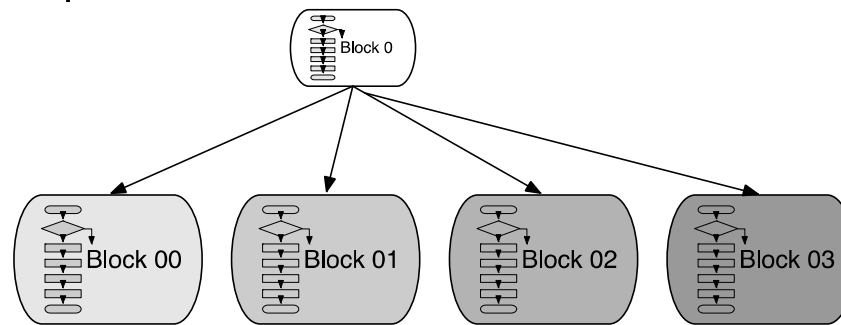


Node 0 has Four Children, Each with a Block

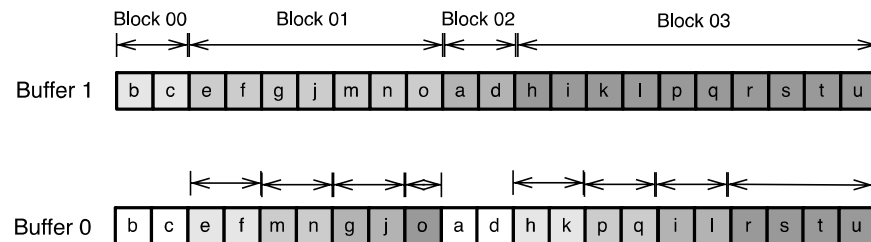
Block 0 launches a child grid of 4 blocks.



Depth = 1

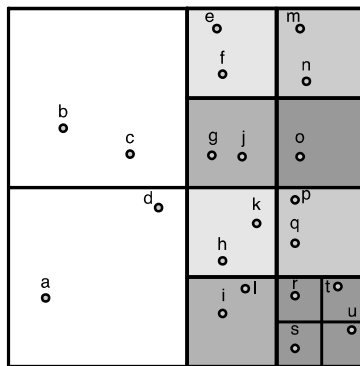


**Blocks 00 and 02
copy back to Buffer 0.**

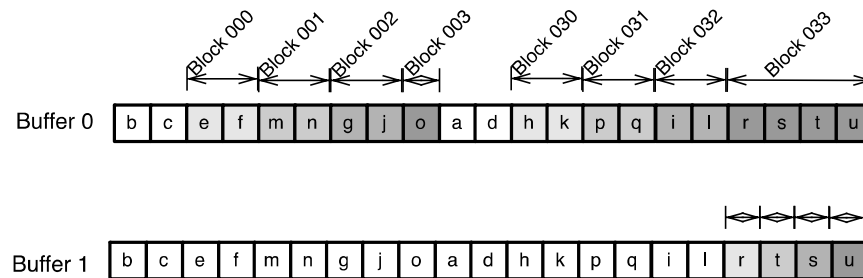
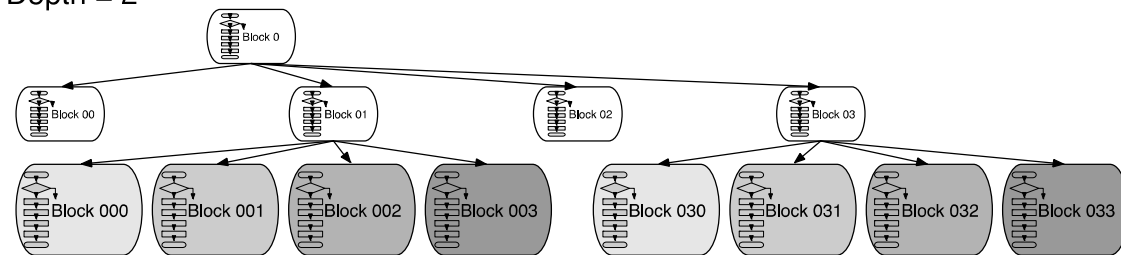


Two Second-Level Nodes Have Children

Blocks 01 and 03 launch child grids with 4 blocks.

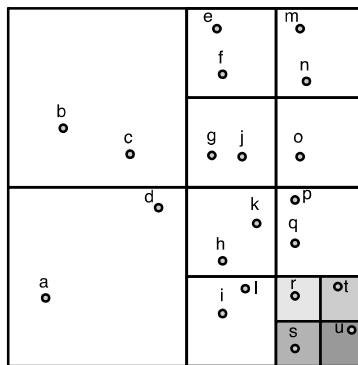


Depth = 2

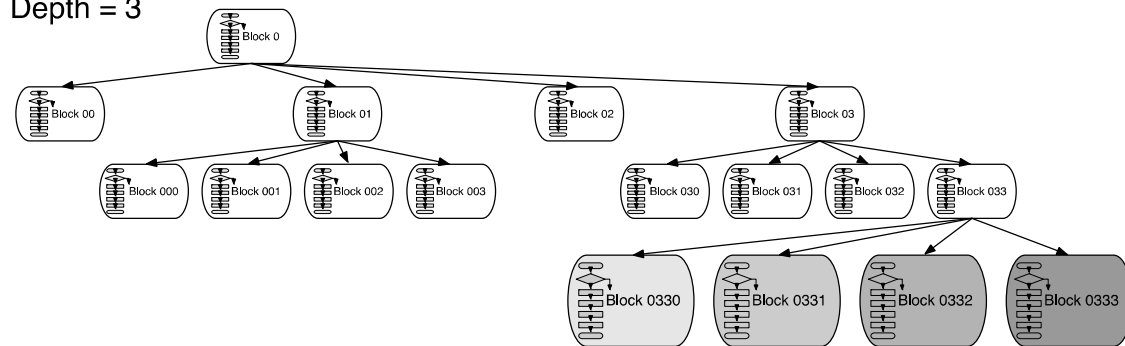


One Third-Level Node Has Children

Block 033 launches a child grid with 4 blocks.



Depth = 3



All four blocks copy back to Buffer 0.

Buffer 1

b	c	e	f	m	n	g	j	o	a	d	h	k	p	q	i	l	r	t	s	u
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

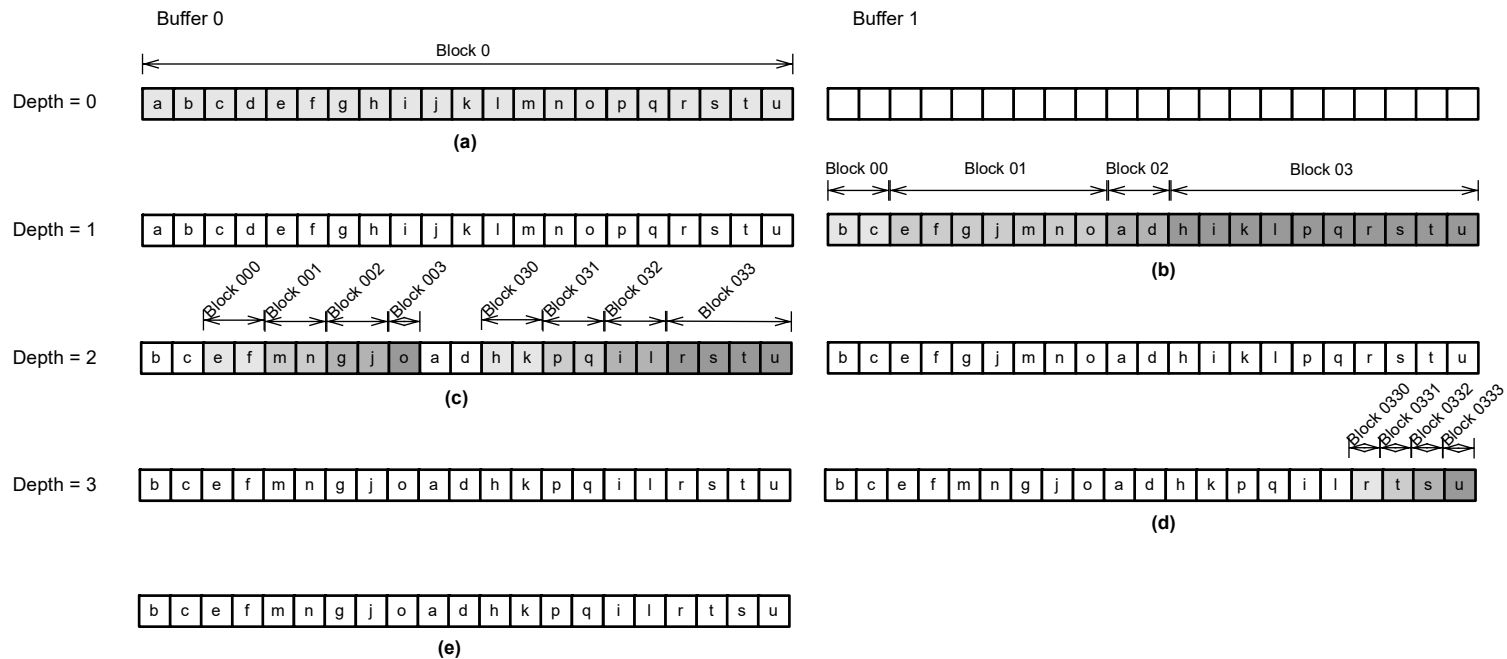
Buffer 0

b	c	e	f	m	n	g	j	o	a	d	h	k	p	q	i	l	r	t	s	u
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Block 0330
Block 0331
Block 0332
Block 0333

When Done, Each Quadrant is a Contiguous Set

Points in the same quadrant are grouped together.



Code is in the Textbook

- Code is Ch. 13 of the textbook (split between the text and an Appendix at the end of the chapter).
- It's a fair bit of code, and hides a lot of activity with C++ semantics, although it does use structures of arrays instead of array of structures.
- Please read it for further details.

Think Back to Our Earlier Discussions

Let's go back briefly to our discussions of

- library calls and
- Mont-Blanc,
 - the Barcelona supercomputer
 - based on ARM cores
 - coupled with high-end GPUs.

Library Calls Stay within the GPU

Library calls such as cuBLAS can now stay within a GPU.

```
__global__ void libraryCall(float *a,
                           float *b,
                           float *c)
{
    // All threads generate data
    createData(a, b);
    __syncthreads();

    // The first thread calls library
    if (threadIdx.x == 0) {
        cublasDgemm(a, b, c);
        cudaDeviceSynchronize();
    }

    // All threads wait for results
    __syncthreads();

    consumeData(c);
}
```

CPU launches
kernel

Per-block
data
generation

Call of 3rd
party library

3rd party
library
executes

Parallel
use
of result

Mont-Blanc Could Not Compete on Lattice QCD

Mont-Blanc machine at Barcelona Supercomputer Center

- used same GPUs as contemporary supercomputers, but
- replaced x86 CPU chips with low-power ARM cores.

Lattice QCD is a code

- used by nearly every computational scientist
- interested in quantum chromodynamics.

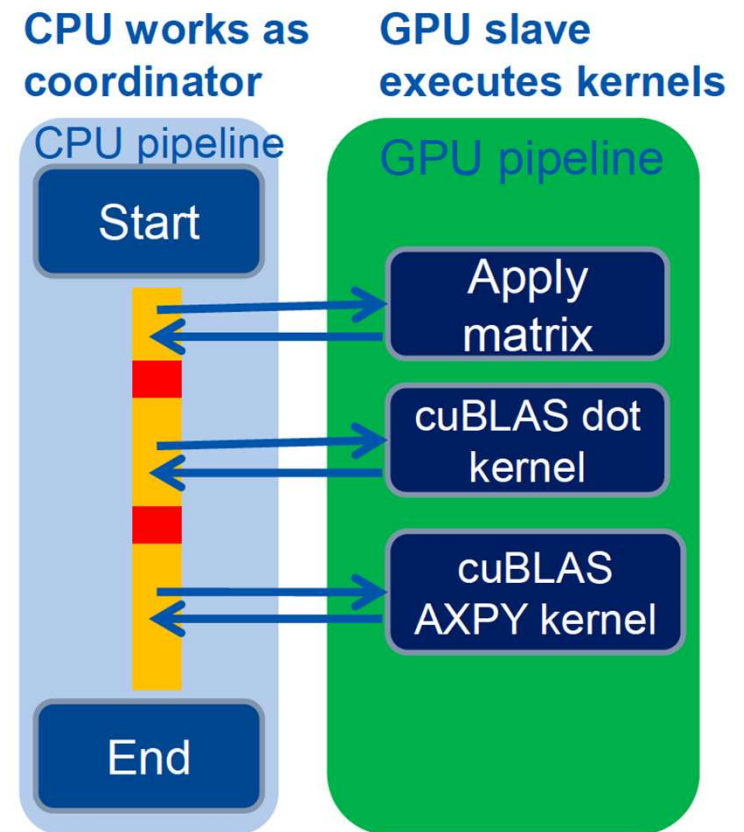
Performance was poor.

CPUs Could Not Keep GPUs Busy!

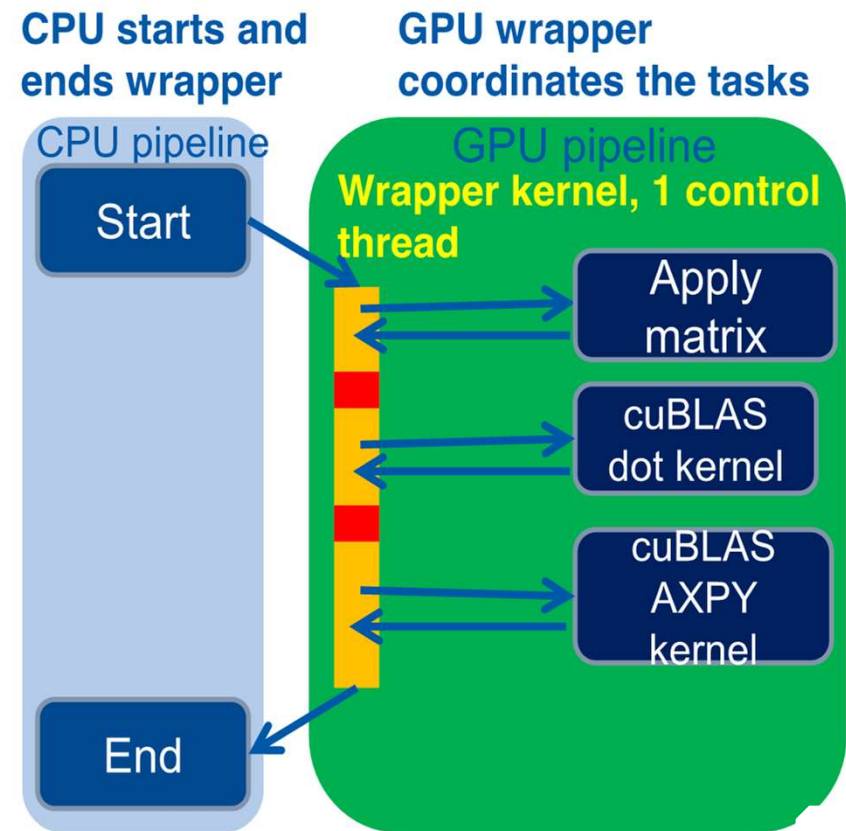
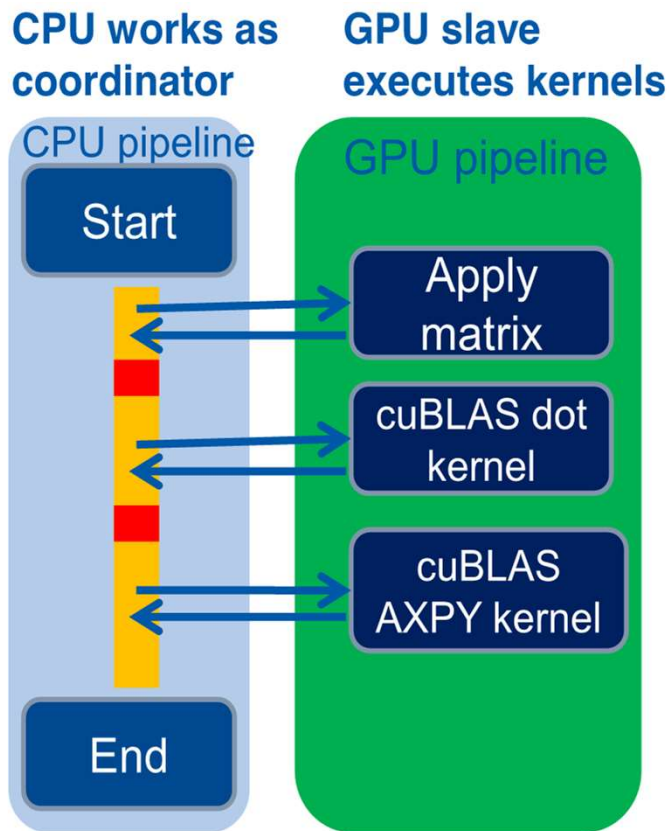
Lattice QCD bottlenecks*

- **Many calls** to cuBLAS.
- Dominated by CPU's ability to launch cuBLAS kernels.
- ARM **CPU not fast enough**, so **GPUs underutilized**.

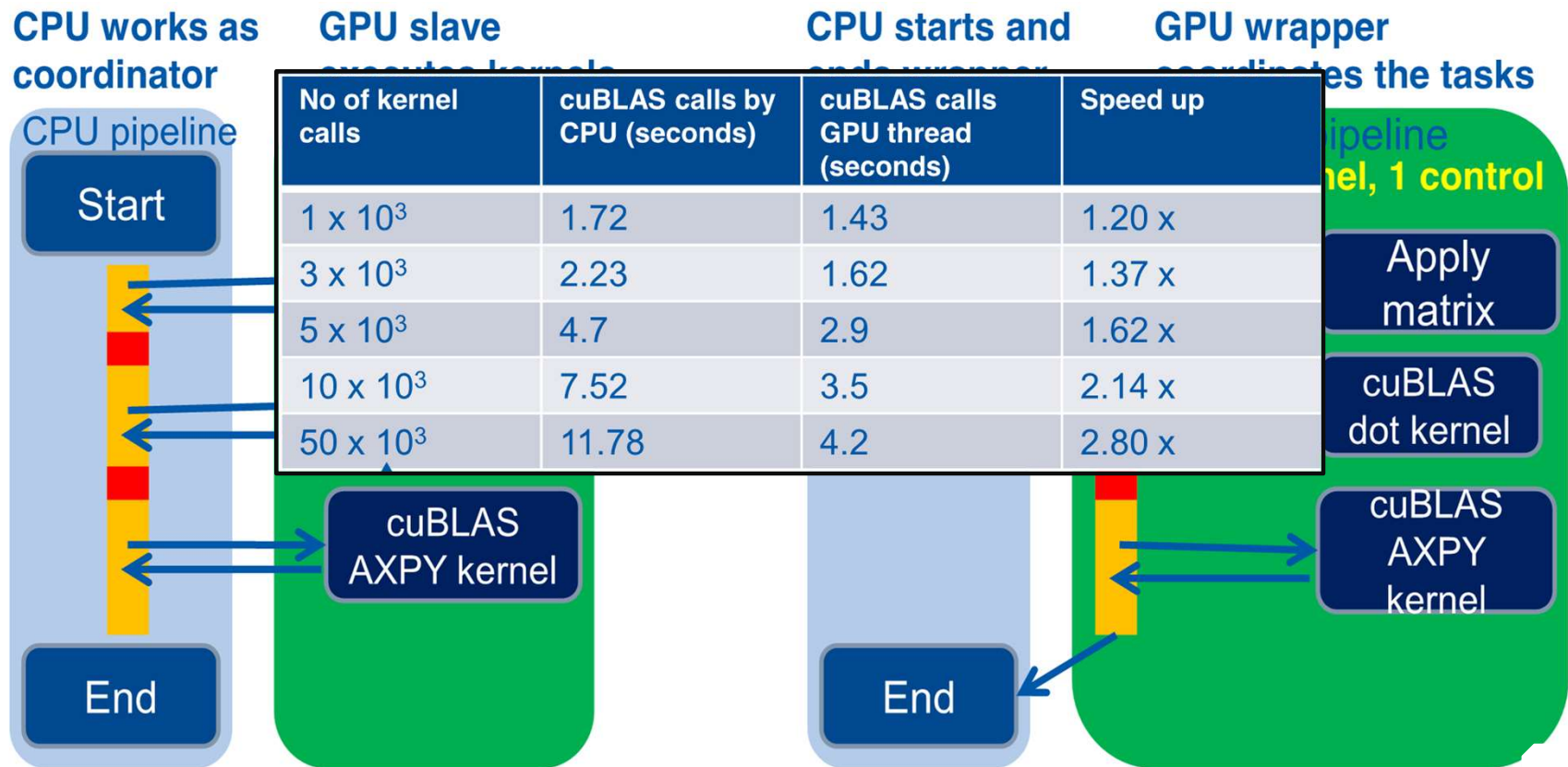
*V. Mehta, "Exploiting CUDA Dynamic Parallelism for Low Power ARM Based Prototypes," GTC, 2015.



With Dynamic Parallelism, Comparable Performance



With Dynamic Parallelism, Comparable Performance



Boons of Dynamic Parallelism

Dynamic parallelism

- **helps load balance** by exposing parallelism, and
- sometimes **improves programmability** by avoiding undesired dependence on the CPU.

In fact, we can

- **move “wrapper kernels” onto the GPU,**
- as we saw with Lattice QCD on Mont-Blanc;
- a BFS wrapper kernel can launch successive frontiers from GPU instead of coordinating from the CPU.

Banes of Dynamic Parallelism

But grids with **small numbers of threads**

- can **severely underutilize GPU resources**, as
- each SM limits the number of concurrent thread blocks.

Launching each **kernel**

- also **takes time and**
- **has** other **resource implications** for the GPU.

Strategy for Success with Dynamic Parallelism

Generally, try to **ensure** that

- grids have at least **a couple thousand threads**,
- preferably spread over **enough blocks**
- **to** also **leverage** more than one or two **SMs**.

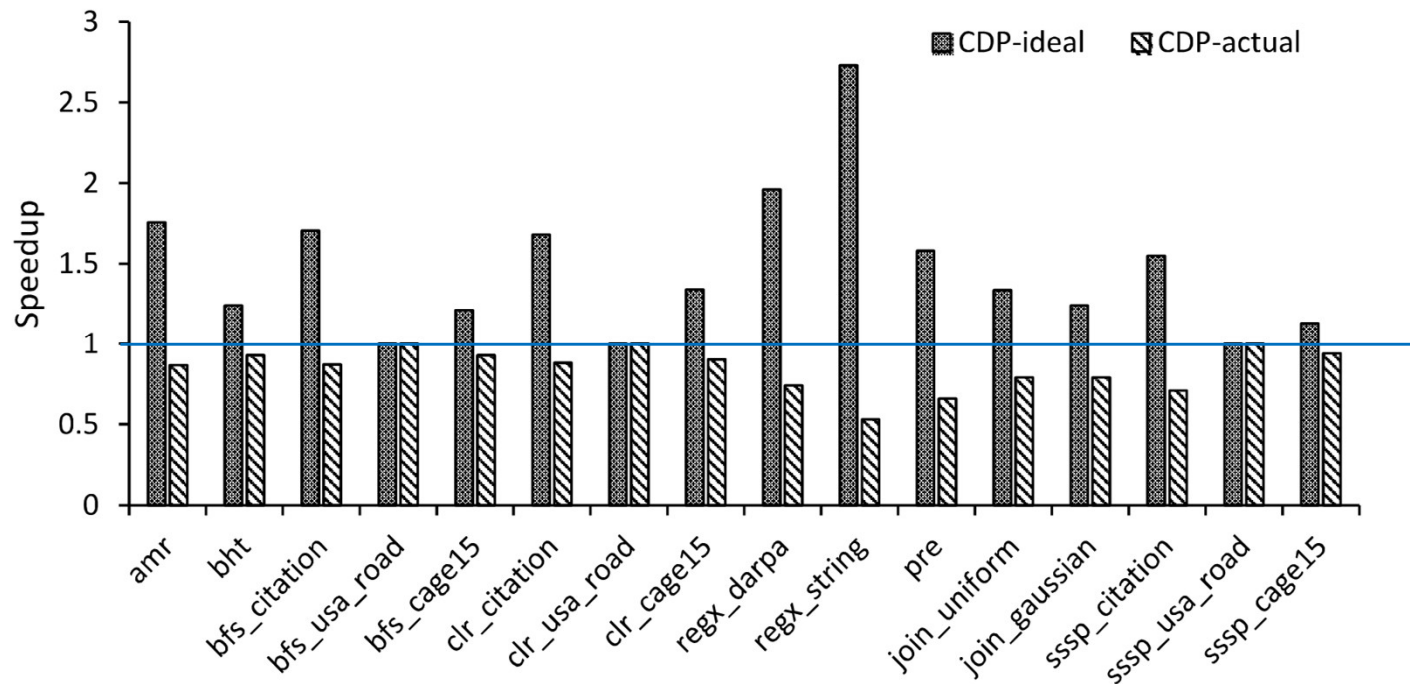
Nested Parallelism Works Well

Nested parallelism (as in quadtree) is **a good fit**:

- thick tree nodes (each node deploys many threads)
- and/or branch degree is large (each parent node has many children).
- Hardware limits nesting depth, so only shallow trees can be implemented efficiently.

For other problems, one may need to **use aggregation** (to be discussed).

Launch Overhead Can Dominate the Results



(graph from J. Wang, S. Yalamanchili, “Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications, IISWC, 2014.)

Where Has My Launch Time Gone?

- **parameter allocation**
- **launch command**
 - from SM (streaming multiprocessor)
 - to KMU (kernel management unit, a central resource),
- **dispatch kernel** from KMU to KD (kernel distributor, which distributes blocks to SMs)
- **pending kernels, suspended parents**

Lots of Work Addressing Launch Overhead

Software solutions

- CUDA-NP, PPOPP'2014
- Parallelization templates, ICPP'2015
- Free launch, MICRO'2015

Hardware improvements

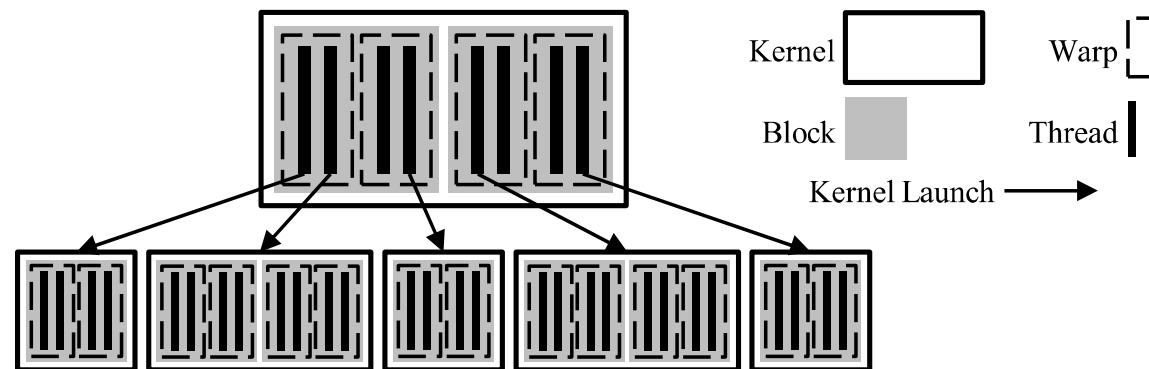
- DTBL, ISCA'2015
- LaPerm, ISCA'2016

What Can We Do with a Bunch of Launches?

Imagine a set of threads launching grids.

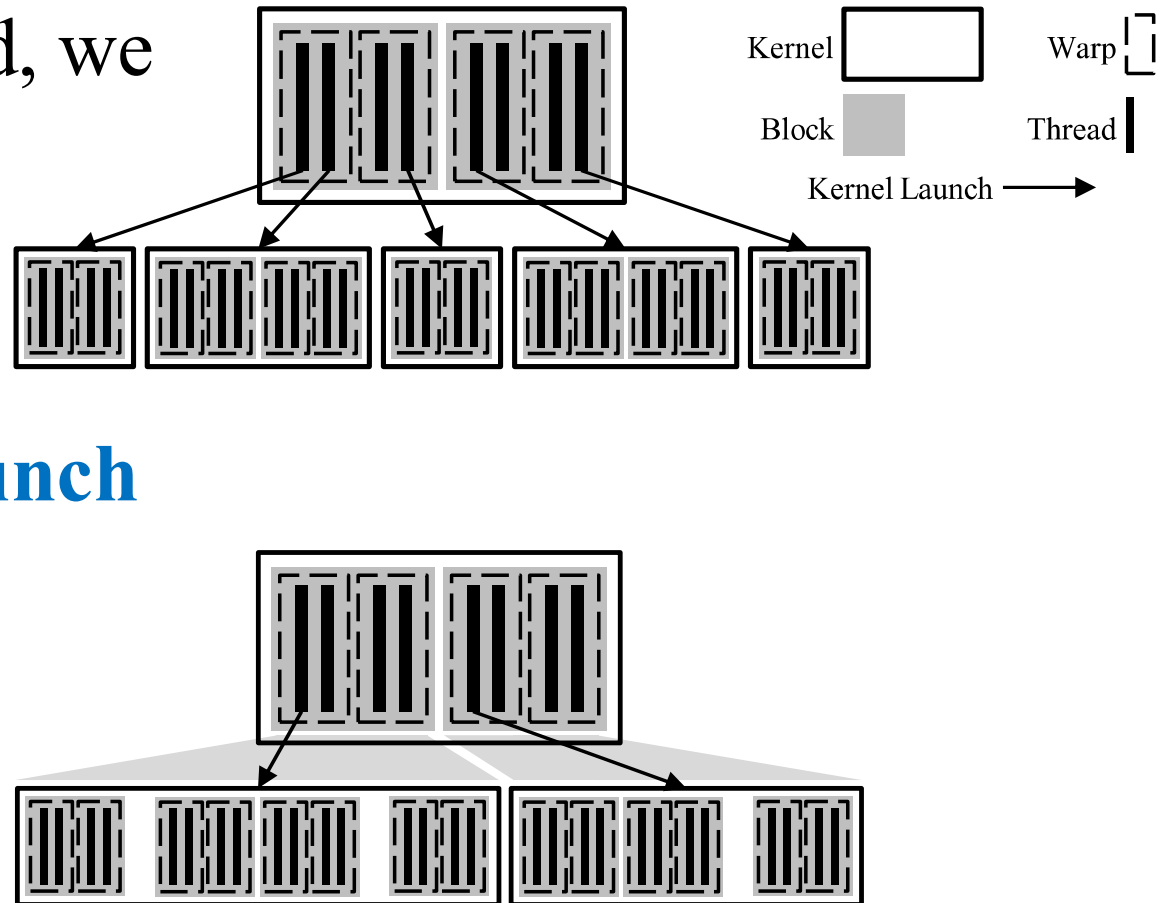
Assume that the new grids

- all use the **same kernel** (as each other—
- not necessarily the same as the threads doing the launching).



Aggregation at Block Level

To reduce overhead, we might **aggregate launches** from within each block **into one larger launch** per thread block.



Aggregation Automation Studied by I. El Hajj


Other types of aggregation are also **possible**.

Details on **automation and evaluation** of the idea are in **Izzat El Hajj's Ph.D. Thesis**,

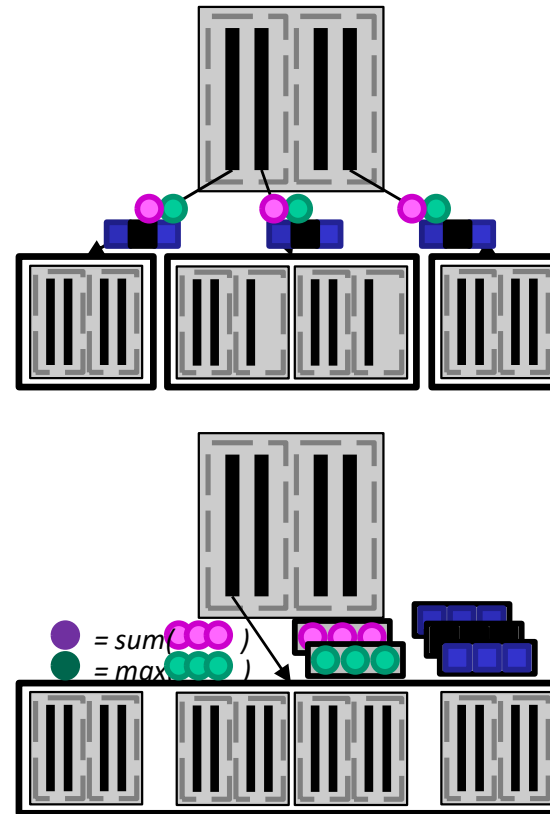
“Techniques for Optimizing Dynamic Parallelism on Graphics Processing Units,” **2018**.

kernel<<<gD, bD>>> (arg1, arg2, arg3)

Original Kernel Call

allocate arrays for args, gD, and bD 
 store args in arg arrays
 store gD in gD array, and bD in bD array
 ● new gD = sum of gD array across warp/block
 ● new bD = max of bD array across warp/block
 if(threadIdx == launcher thread in warp/block) {
 kernel_agg<<<new gD, new bD>>>
 (arg arrays, gD array, bD array)
 }

Transformed Kernel Call
 (block-granularity aggregation example)

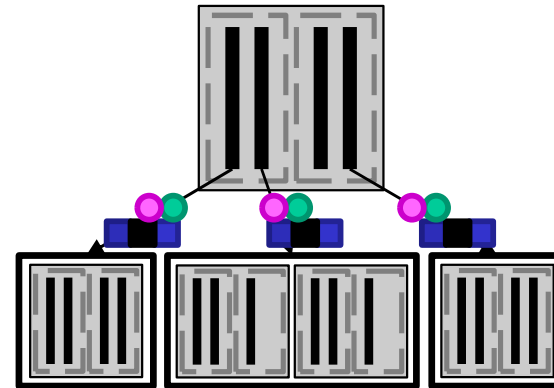



```

__global__ void kernel(params) {
    kernel body
}

```

Original Kernel

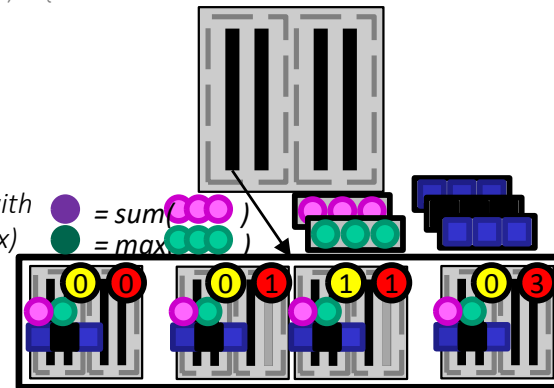



```

__global__ void kernel_agg(param arrays, gD array, bD array) {
    calculate index of parent thread ( original kernel index) #
    load params from param arrays
    load actual gridDim/blockDim from gD/bD arrays
    calculate actual blockDim #
    if(threadIdx < actual blockDim) {
        kernel body (with kernel launches transformed and with
                    using actual gridDim/blockDim/blockIdx)
    }
}

```

Transformed Kernel
(block-granularity aggregation example)




$gD =$  $= \{ 1, 2, 0, 1 \}$


$gDs = \text{scan}(\text{array of 4 purple circles}) = \{ 0, 1, 3, 3, 4 \}$ (performed by parent thread as optimization)

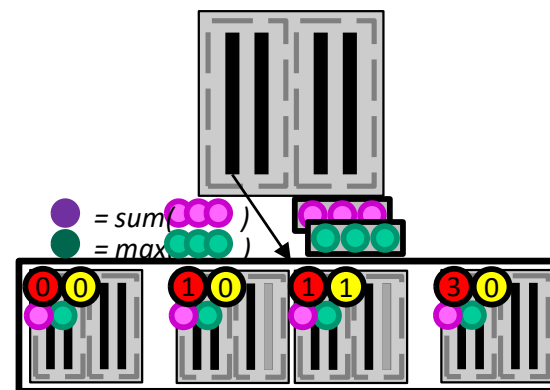
$\text{blockIdx.x} = 0 \quad 1 \quad 2 \quad 3$

$\text{red circle \#} = p \mid gDs[p] \leq \text{blockIdx.x} < gDs[p+1]$ (n-arry search on gDs for p satisfying this condition)
 $= 0 \quad 1 \quad 1 \quad 3$

$\text{yellow circle \#} = \text{blockIdx.x} - gDs[p]$
 $= 0 \quad 0 \quad 1 \quad 0$

calculate index of parent thread 

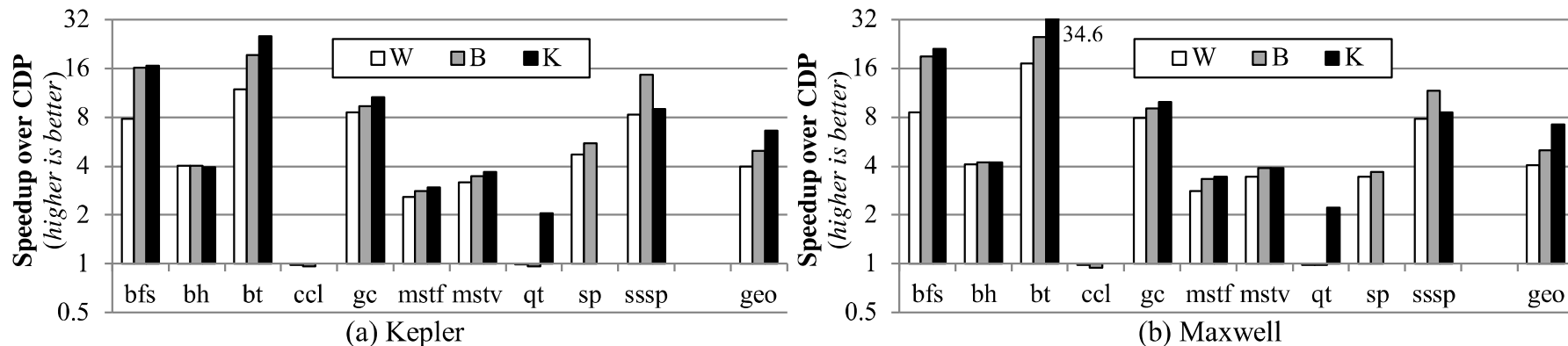
calculate actual blockIdx 



Aggregation

Results from Izzat's early work.*

W = warp-level, B = block-level, K = kernel-level



*I. El Hajj, J. Gómez-Luna, C. Li, L.-W. Chang, D. Milojicic, W.-m. Hwu, “KLAP: Kernel Launch Aggregation and Promotion for Optimizing Dynamic Parallelism,” MICRO, 2016.



ANY QUESTIONS?