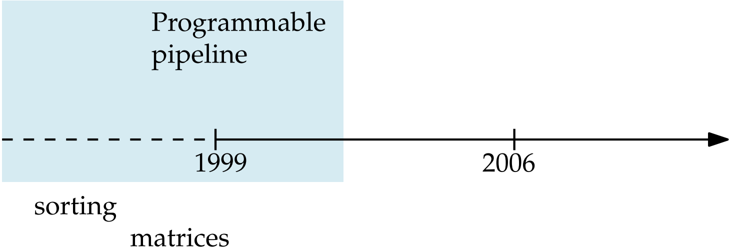
GPU Algorithms II
Models and CUDA

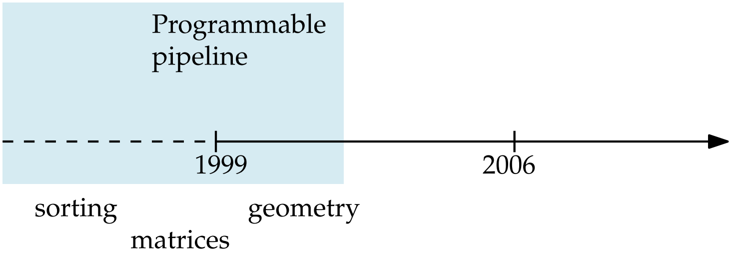
MADALGO Summer School on Algorithms for Modern Parallel and Distributed Models

Suresh Venkatasubramanian
University of Utah

Previously...

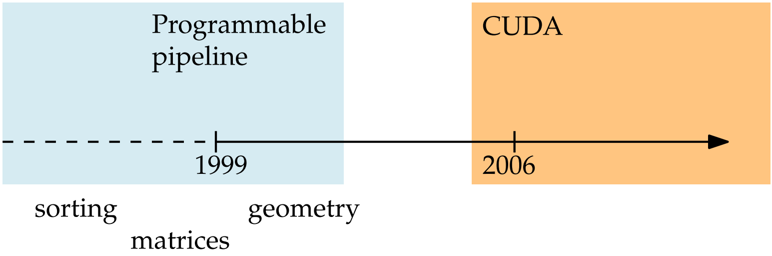


Outline



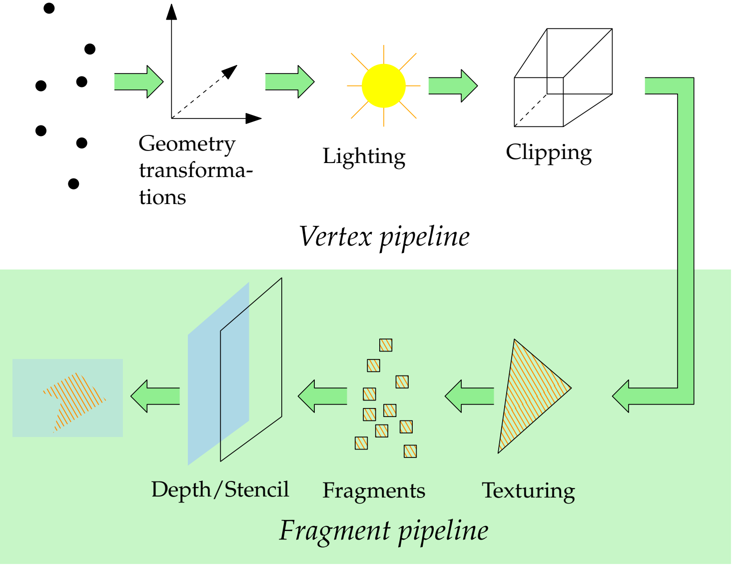
A streaming model

Outline



A streaming model

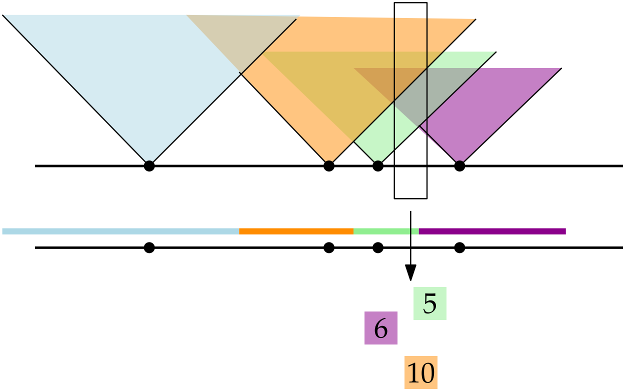
The GPU Pipeline



Fragment shader operations

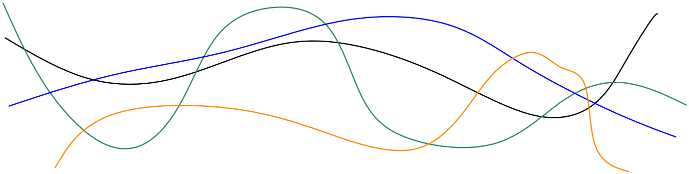
- *Every pixel* acts like a *streaming* SIMD processor
  - actually, some fixed number are processed in parallel
- Fragment processor could perform simple *straight-line operations* and conditionals (no looping)
- (limited) texture memory for local storage
- Each pixel processor could do a simple `reduce` (`add`, `blend`)
- Computation proceeds in *passes*: output could be rendered or stored in memory for next pass.
- All computation on GPU from start of the vertex pipeline

Algorithms via Lower Envelopes[AKMV03]



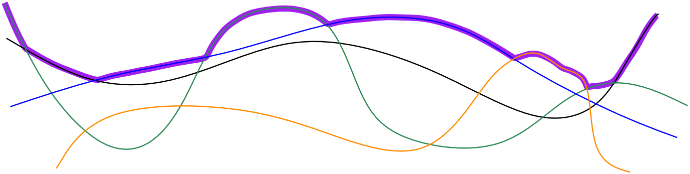
Voronoi diagram is *lower envelope* of collection of distance functions

General Envelope Extents



- Let $f_1, \ldots f_n$ be a set of functions from $\mathbb{R}^2 \rightarrow \mathbb{R}$

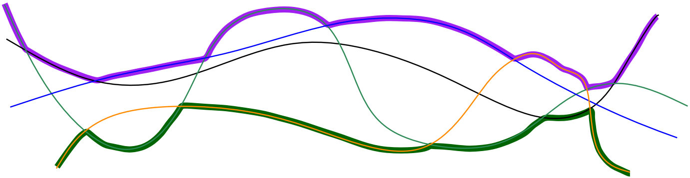
General Envelope Extents



- Let $f_1, \ldots f_n$ be a set of functions from $\mathbb{R}^2 \to \mathbb{R}$

$$U(x) = \max_i f_i(x)$$

General Envelope Extents

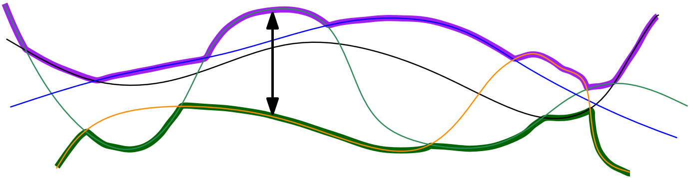


- Let $f_1, \ldots f_n$ be a set of functions from $\mathbb{R}^2 \rightarrow \mathbb{R}$

$$U(x) = \max_i f_i(x)$$

$$L(x) = \min_i f_i(x)$$

General Envelope Extents



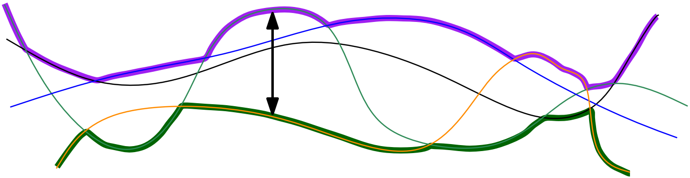
- Let $f_1, \ldots f_n$ be a set of functions from $\mathbb{R}^2 \rightarrow \mathbb{R}$

$$U(x) = \max_i f_i(x)$$

$$L(x) = \min_i f_i(x)$$

$$E(x) = U(x) - L(x)$$

General Envelope Extents



- Let $f_1, \ldots f_n$ be a set of functions from $\mathbb{R}^2 \to \mathbb{R}$

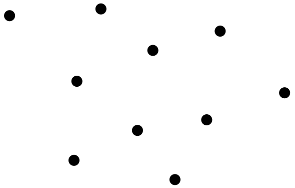
$$U(x) = \max_i f_i(x)$$

$$L(x) = \min_i f_i(x)$$

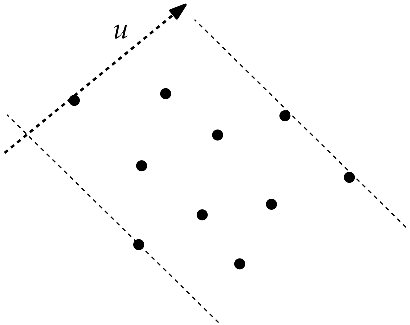
$$E(x) = U(x) - L(x)$$

Compute $(\min, \max)_x E(x), L(x), U(x)$

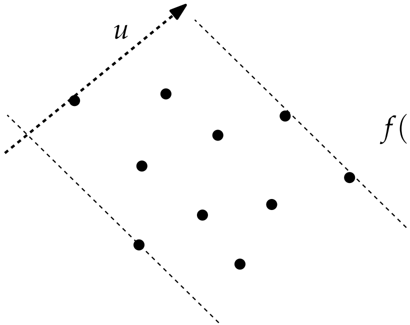
Examples



Examples

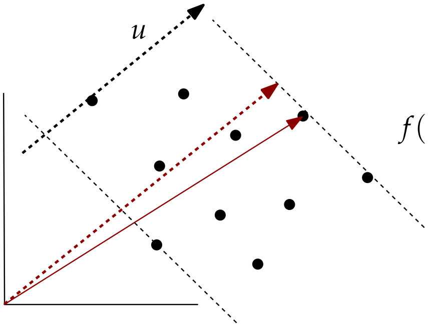


Examples



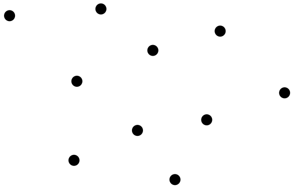
$f(u) = \max_{p \in P} \langle u, p \rangle$

Examples



$$f(u) = \max_{p \in P} \langle u, p \rangle$$

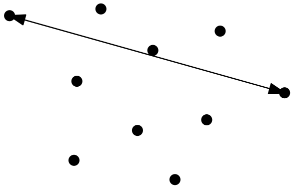
Examples



$$f(u) = \max_{p \in P} \langle u, p \rangle$$

$$\Delta(P) = \max_u f(u) - f(-u)$$

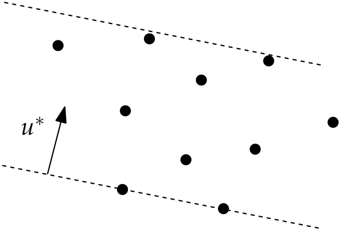
Examples



$f(u) = \max_{p \in P} \langle u, p \rangle$

$\Delta(P) = \max_u f(u) - f(-u)$

Examples

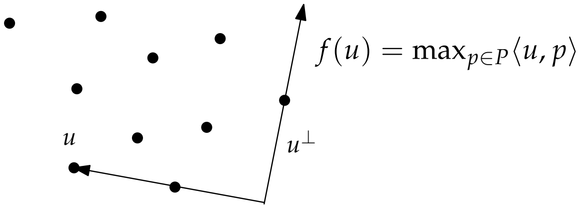


$f(u) = \max_{p \in P} \langle u, p \rangle$

$\Delta(P) = \max_u f(u) - f(-u)$

Width(P) = $\min_u W(u) \triangleq f(u) - f(-u)$

Examples

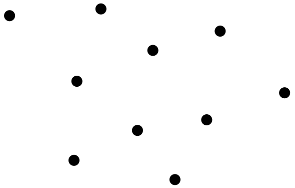


$$\Delta(P) = \max_u f(u) - f(-u)$$

$$\text{Width(P)} = \min_u W(u) \triangleq f(u) - f(-u)$$

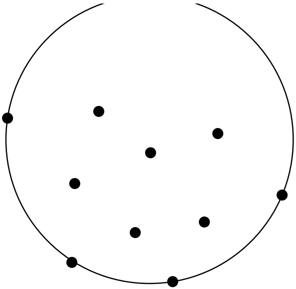
$$\text{Min-Bbox(P)} = \min_u W(u) \cdot W(u^{\perp})$$

Examples



$$d_p(x) = \|p - x\|$$

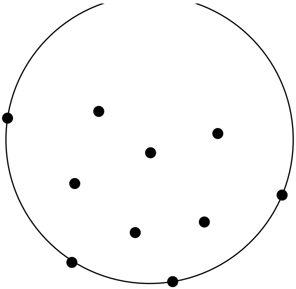
Examples



$d_p(x) = \|p - x\|$

MEB(P) = $\min_c \max_p d_p(c)$

Examples

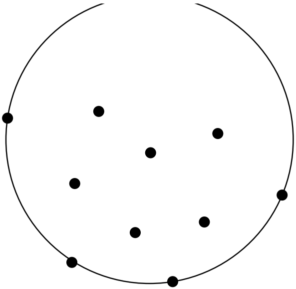


$d_p(x) = \|p - x\|$

$g(x) = \sum_p \|p - x\|$

$\text{MEB(P)} = \min_c \max_p d_p(c)$

Examples



$d_p(x) = \|p - x\|$

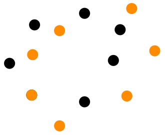
$g(x) = \sum_p \|p - x\|$

MEB(P) = $\min_c \max_p d_p(c)$

1-median(P) = $\min_x g(x)$

Examples

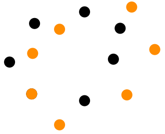
Hausdorff Distance



$$d_H(P, Q) = \max_{p \in P} \min_{q \in Q} \|p - q\|$$

Examples

Hausdorff Distance

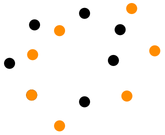


$$d_H(P, Q) = \max_{p \in P} \min_{q \in Q} \|p - q\|$$

- Penetration depth between two surfaces
- Best-fit circle, minimum width annulus

Examples

Hausdorff Distance

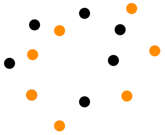


$$d_H(P,Q) = \max_{p \in P} \min_{q \in Q} \|p - q\|$$

- Penetration depth between two surfaces
- Best-fit circle, minimum width annulus
- Discretize the space to get an appropriate approximation

Examples

Hausdorff Distance



$$d_H(P,Q) = \max_{p \in P} \min_{q \in Q} \|p - q\|$$

- Penetration depth between two surfaces
- Best-fit circle, minimum width annulus
- Discretize the space to get an appropriate approximation
- Some computations happen in dual space

Development Support

Language Support

- `Brook`[BFH+04]
- `Cg`[MGAK03]
- `GLSL/HLSL`[Fou, Mic]
- `LibSh`[MDT04]

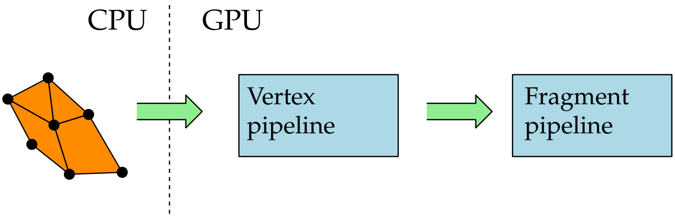
Coding Support

- Cg debugger
- Code optimizer

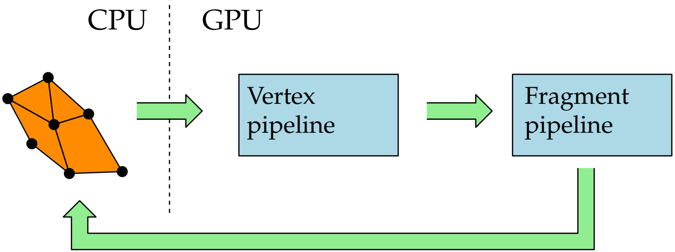
GPU programming moves closer to "real" programming

- API is still delinked from hardware, and this is deliberate
- Parallelism is still partly a fiction.

Geometry Shaders

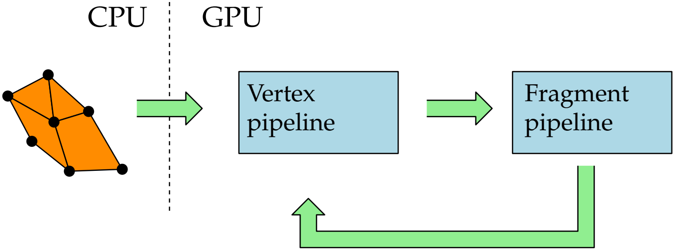


Geometry Shaders



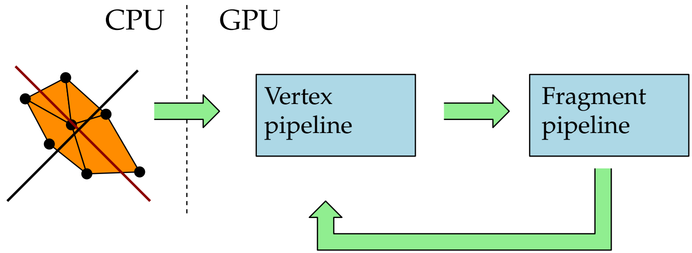
- A multipass GPU computation requires crossing the CPU-GPU interface

Geometry Shaders



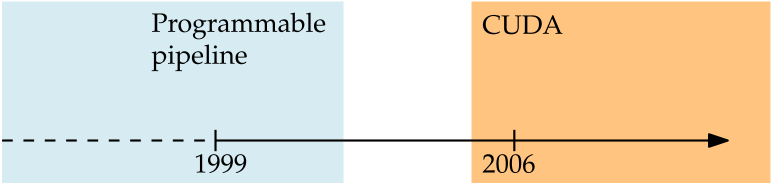
- A multipass GPU computation requires crossing the CPU-GPU interface
- Geometry shaders allow programming the vertex pipeline

Geometry Shaders



- A multipass GPU computation requires crossing the CPU-GPU interface
- Geometry shaders allow programming the vertex pipeline
- Geometry can be generated inside the vertex pipeline

Outline



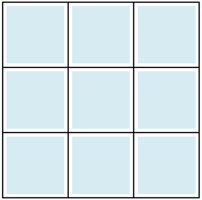
sorting

matrices

geometry

A streaming model

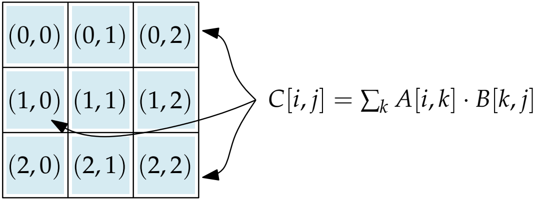
Revisiting the GPU model



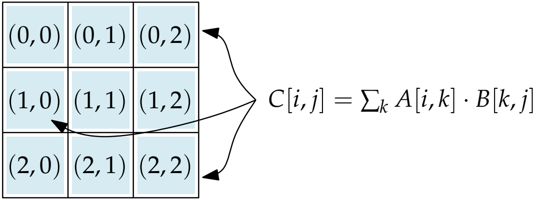
Revisiting the GPU model

$(0,0)$	$(0,1)$	$(0,2)$
$(1,0)$	$(1,1)$	$(1,2)$
$(2,0)$	$(2,1)$	$(2,2)$

Revisiting the GPU model



Revisiting the GPU model

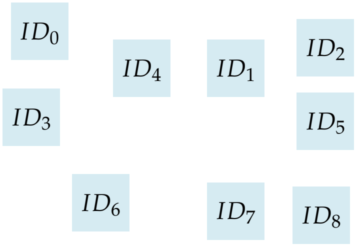


SIMD execution across all elements of grid

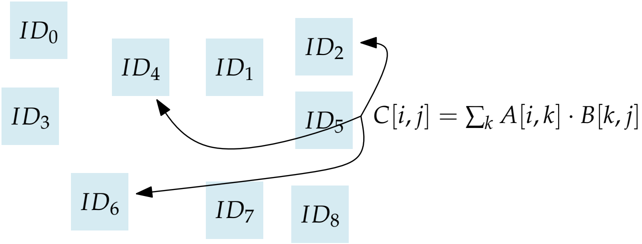
Revisiting the GPU model

ID_0	ID_1	ID_2
ID_3	ID_4	ID_5
ID_6	ID_7	ID_8

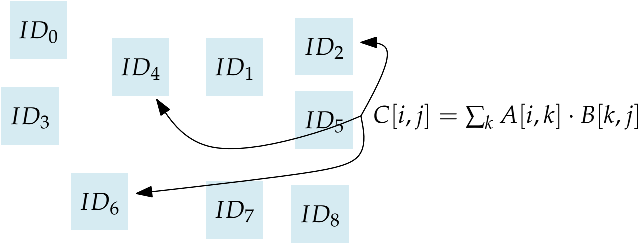
Revisiting the GPU model



Revisiting the GPU model



Revisiting the GPU model



A "block" of threads executing in SIMD

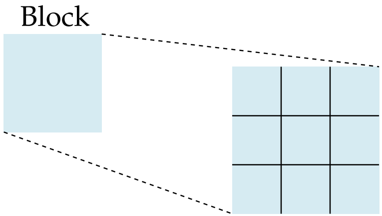
CUDA: Compute Unified Device Architecture[NVI]

- Lightweight threads that run SIMD (SIMT) in "blocks"
- Blocks run in "SPMD" mode (single program, multiple data)
- Memory at multiple levels (thread, blocks, global)
- Threads are very lightweight, and there are many of them.
- Two views: programmer-centric and hardware-centric

CUDA Model: Blocks

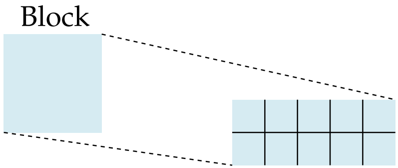
Block

CUDA Model: Blocks



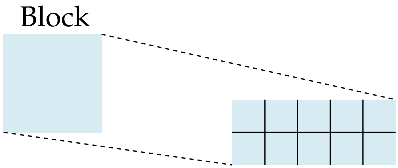
- A block is a collection of threads

CUDA Model: Blocks



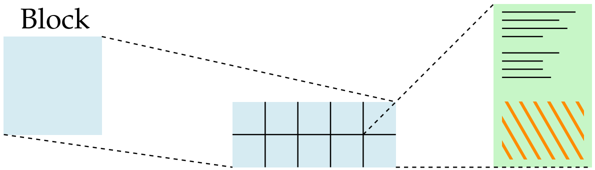
- A block is a collection of threads
- A block can have different ”shapes”

CUDA Model: Blocks



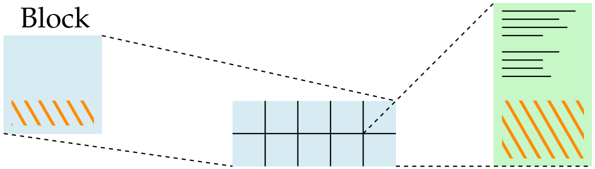
- A block is a collection of threads
- A block can have different "shapes"
- All threads run the same instructions and can synchronize

CUDA Model: Blocks



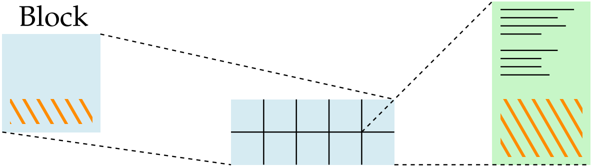
- A block is a collection of threads
- A block can have different "shapes"
- All threads run the same instructions and can synchronize
- Theads have local memory

CUDA Model: Blocks



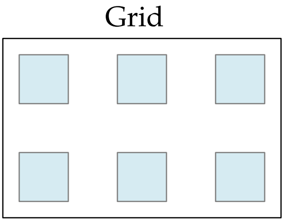
- A block is a collection of threads
- A block can have different ”shapes”
- All threads run the same instructions and can synchronize
- Theads have local memory (and so do blocks)

CUDA Model: Blocks



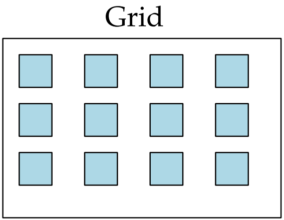
- A block is a collection of threads
- A block can have different "shapes"
- All threads run the same instructions and can synchronize
- Theads have local memory (and so do blocks)
- Block memory is low-latency and shared among threads

CUDA Model: Grids



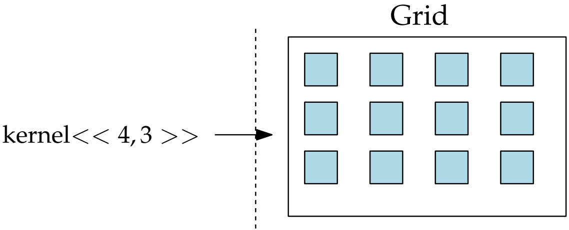
- A grid is a collection of blocks

CUDA Model: Grids



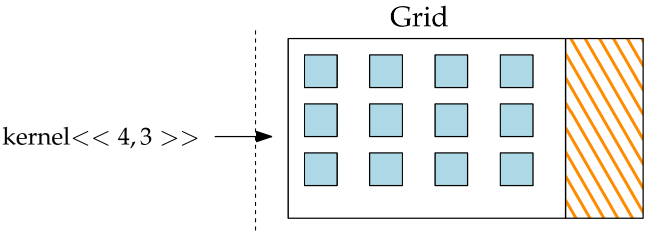
- A grid is a collection of blocks
- A grid can have different shapes

CUDA Model: Grids



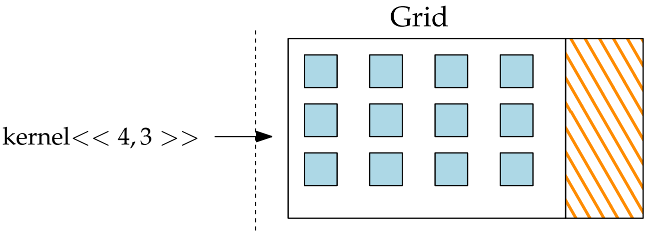
- A grid is a collection of blocks
- A grid can have different shapes
- A grid of blocks is initiated by a request from the host

CUDA Model: Grids



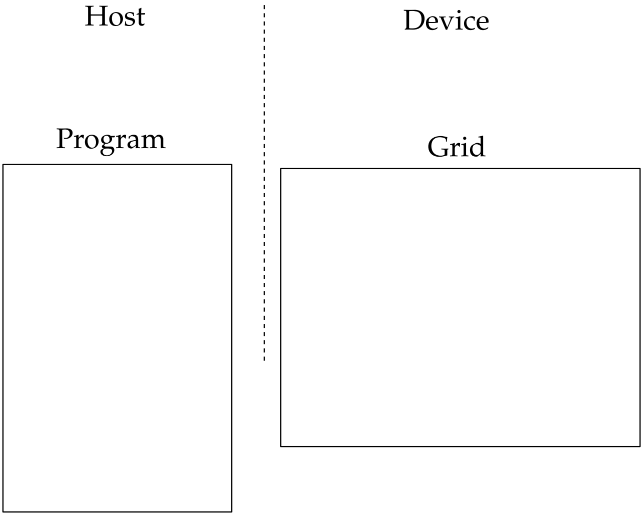
- A grid is a collection of blocks
- A grid can have different shapes
- A grid of blocks is initiated by a request from the host
- A grid has shared memory

CUDA Model: Grids

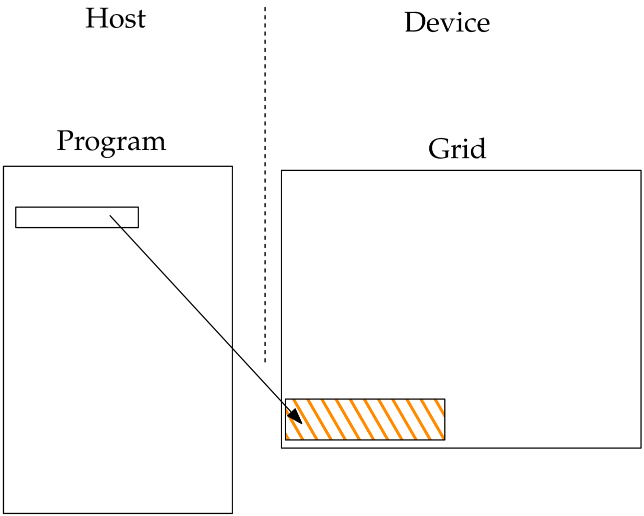


- A grid is a collection of blocks
- A grid can have different shapes
- A grid of blocks is initiated by a request from the host
- A grid has shared memory
- Blocks cannot coordinate with each other and are run independently

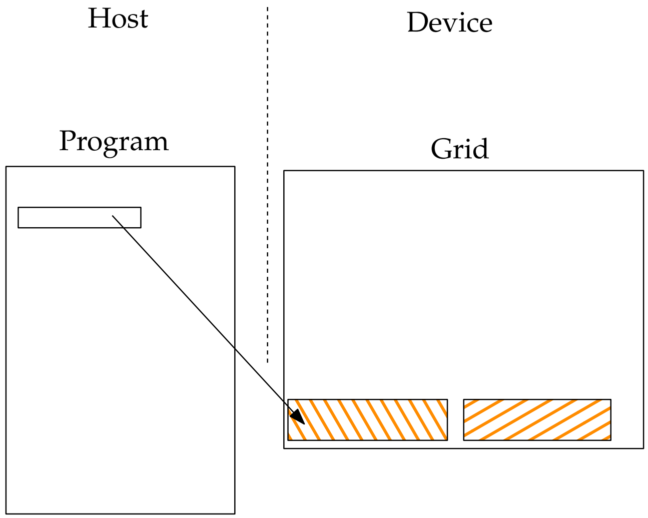
CUDA Model: Overview



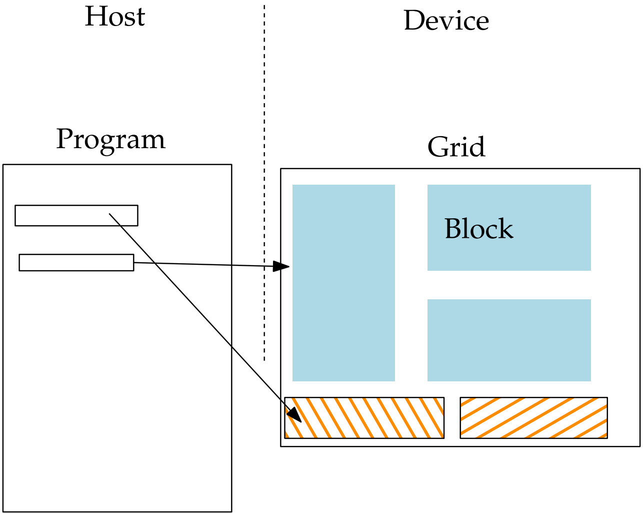
CUDA Model: Overview



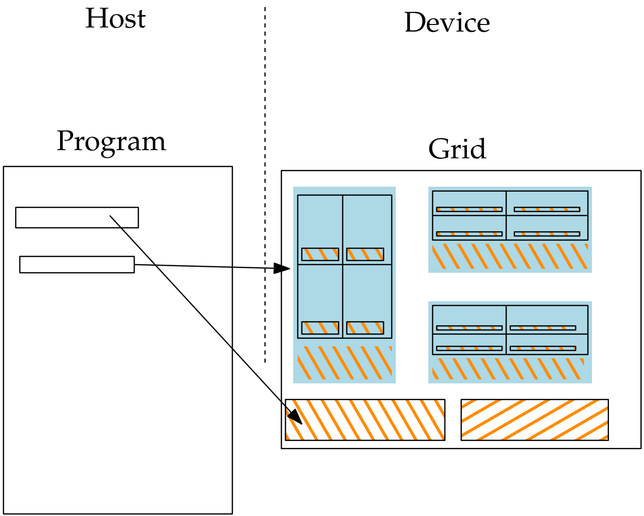
CUDA Model: Overview



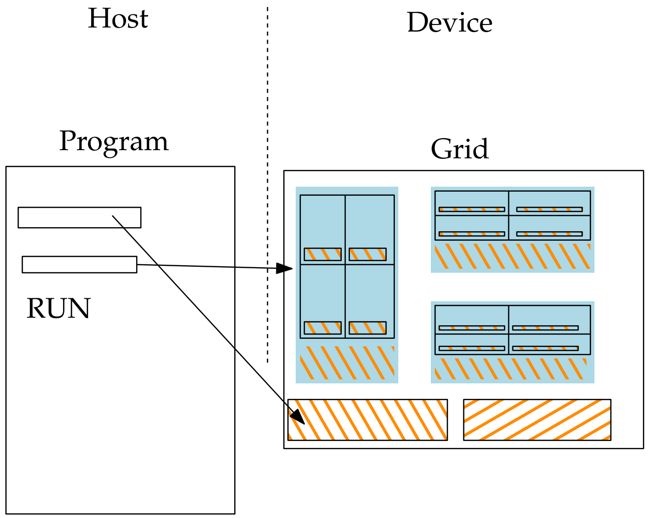
CUDA Model: Overview



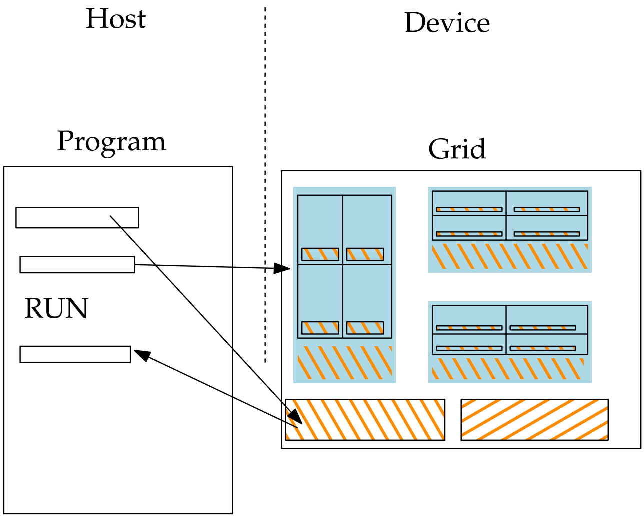
CUDA Model: Overview



CUDA Model: Overview



CUDA Model: Overview



“Hello World”: CUDA Matrix Multiplication

Problem

Multiply two 64×64 *matrices.*

`CUDAalloc(Md,` 64×64`)` {Allocate device memory}
`CUDAalloc(Nd,` 64×64`)`
`CUDAalloc(Pd,` 64×64`)`

`CUDAcopy(Md, M)` {Transfer matrices to device memory}
`CUDAcopy(Nd, N)`

Initiate Kernel

`CUDAcopy(P, Pd)` {Retrieve result from device}

“Hello World”: CUDA Matrix Multiplication

- Thread (i, j) will compute the dot product of row i of M and column j of N
- All threads will be in a single block of a single grid

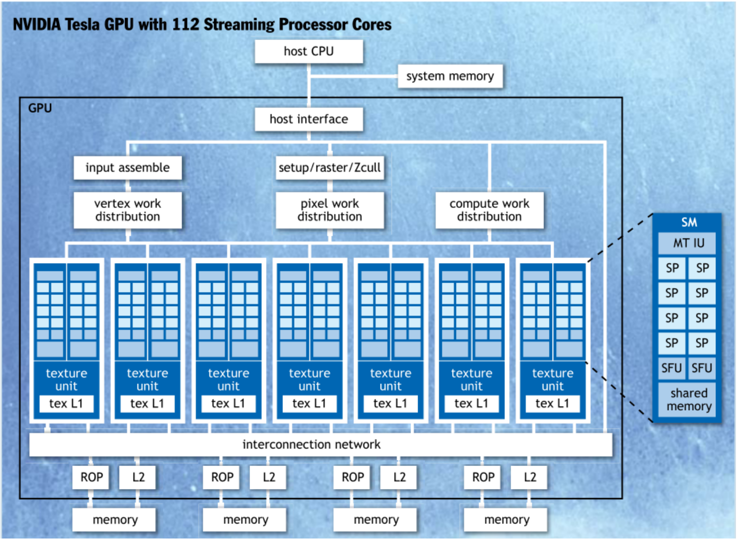
$(tx, ty) \leftarrow (\texttt{threadIdx.x}, \texttt{threadIdx.y})$
$P \leftarrow 0$ {Local thread storage}
for $i = 1 \ldots 64$ **do**
  $\mathrm{P} += \mathrm{Md}[64 \cdot ty + i] \cdot \mathrm{Nd}[64 \cdot i + tx]$
end for
$\mathrm{Pd}[tx, ty] \leftarrow P$ {Write to global memory}

- Kernel is invoked as `matmult` $\langle\langle (1, 1), (64, 64) \rangle\rangle$
- Blocks can only allocate a maximum of 512 threads

CUDA execution model

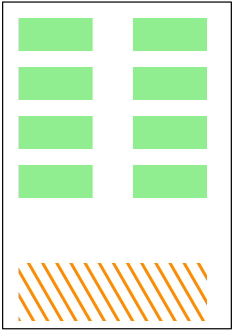
- CUDA looks different to programmer and hardware (like MapReduce)
- Understanding the execution model helps with design of algorithms

CUDA Execution Model



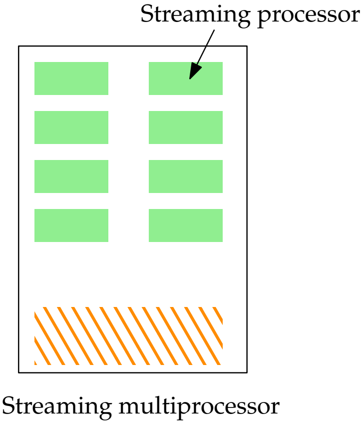
Nickolls, Buck, Garland, Skadron, ACM Queue, Mar 2008[NBGS08]

CUDA Execution Model

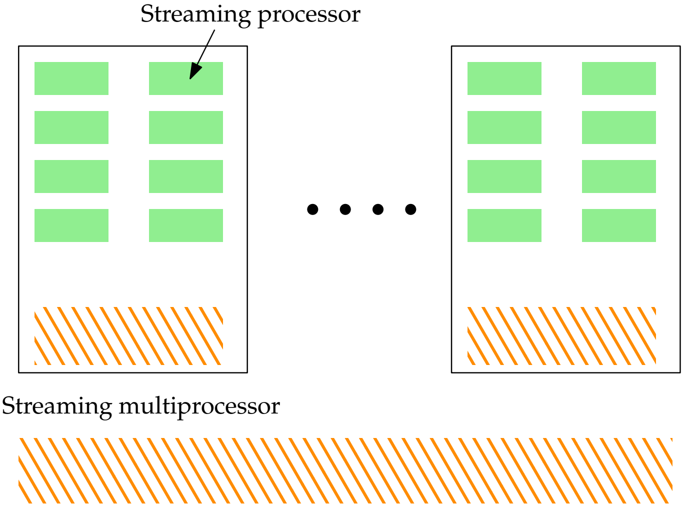


Streaming multiprocessor

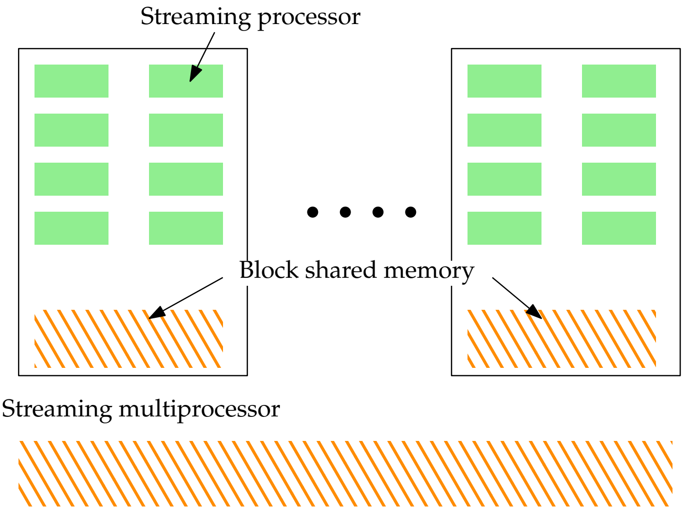
CUDA Execution Model



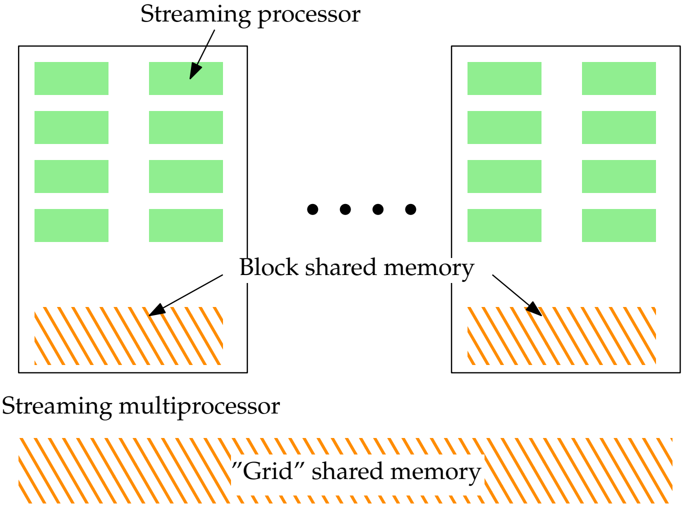
CUDA Execution Model



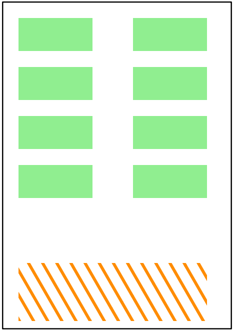
CUDA Execution Model



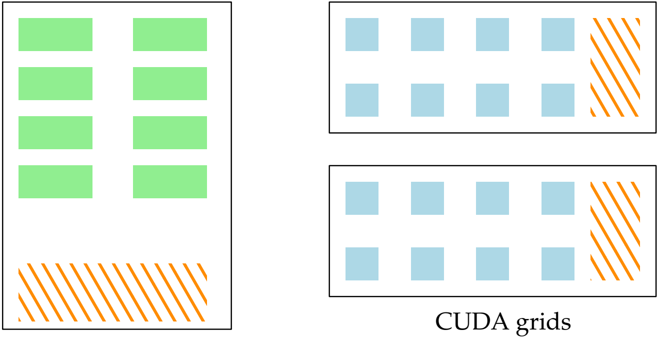
CUDA Execution Model



CUDA Execution Model

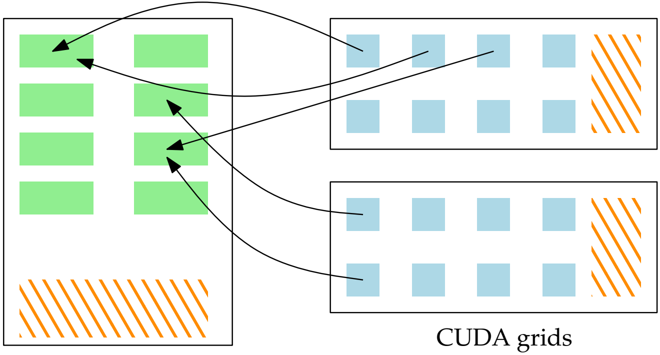


CUDA Execution Model



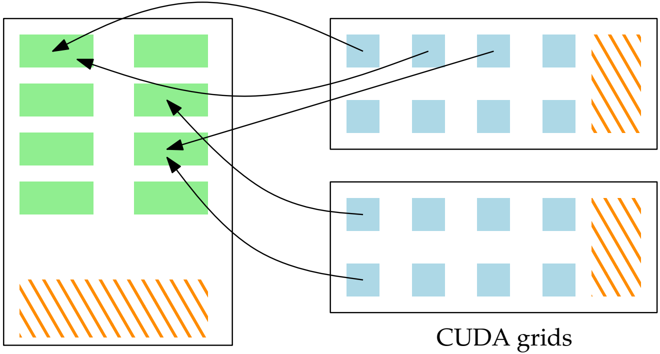
CUDA grids

CUDA Execution Model



- Each block is assigned to a single SP

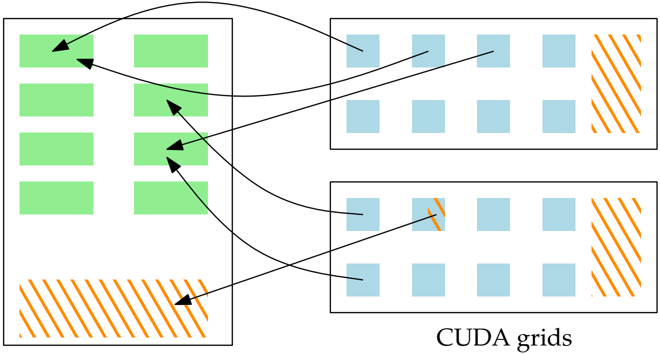
CUDA Execution Model



CUDA grids

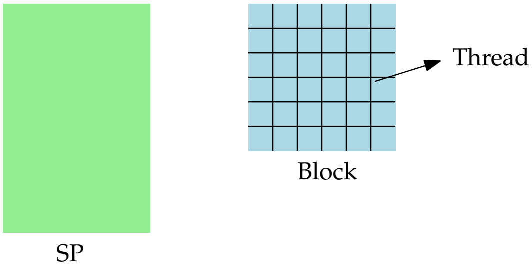
- Each block is assigned to a single SP
- Grid is a software construct

CUDA Execution Model

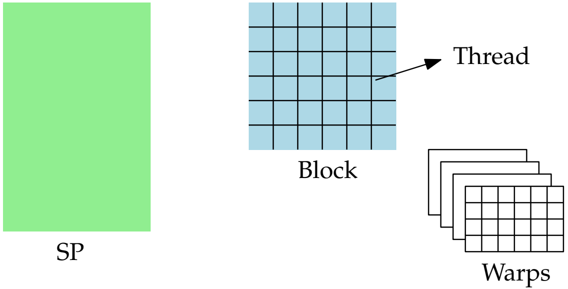


- Each block is assigned to a single SP
- Grid is a software construct
- Block memory managed by SM

CUDA Execution Model

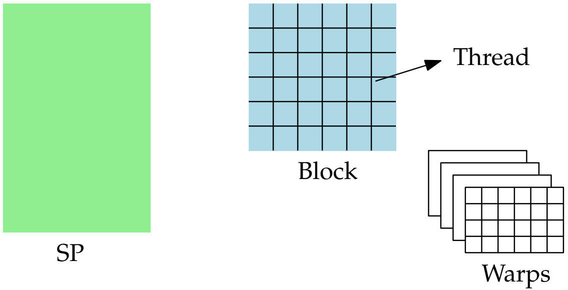


CUDA Execution Model



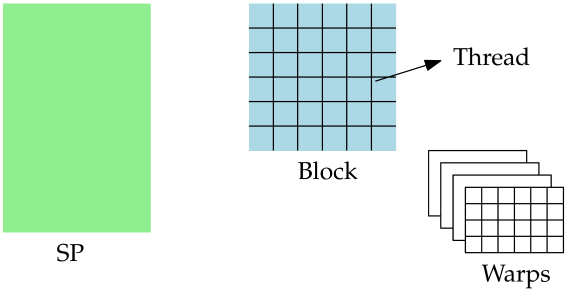
- Each block is divided into groups of 32 threads called ”warps”

CUDA Execution Model



- Each block is divided into groups of 32 threads called "warps"
- Warp threads are scheduled SIMD on the processor

CUDA Execution Model



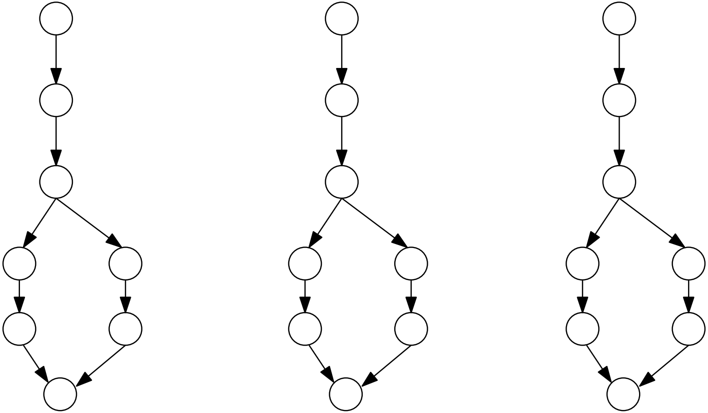
- Each block is divided into groups of 32 threads called "warps"
- Warp threads are scheduled SIMD on the processor
- Warps are scheduled concurrently

CUDA Execution Model: Warps

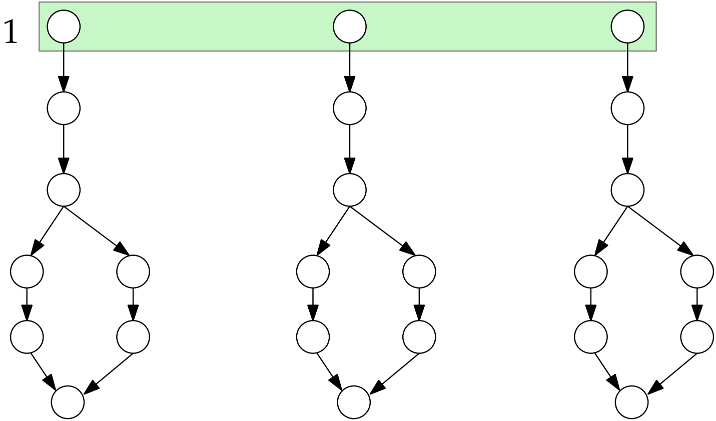
- Each warp consists of at most 32 threads taken from a single block
- All threads in a warp are executed in parallel with zero overhead
- In each clock cycle, a GO command is issued to all threads of warp to execute same command
- If there's branching, branches are executed sequentially – non-executing threads are inactive.
- Maximize throughput by minimizing branching

This is Single Instruction Multiple Threads (SIMT)

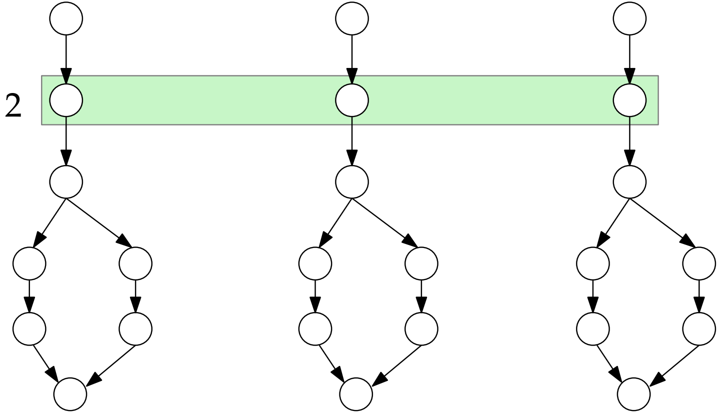
CUDA Execution Model: Warps



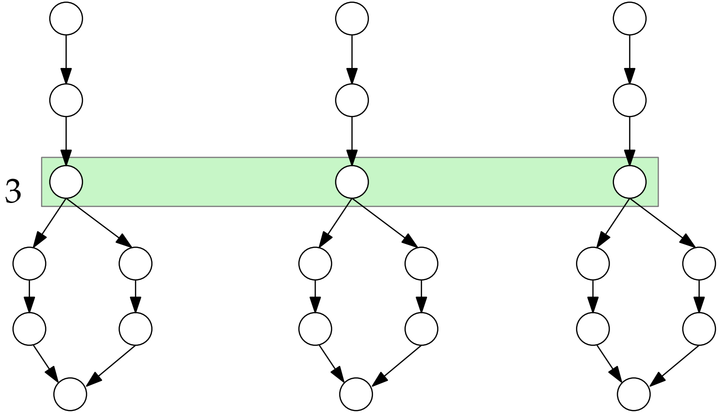
CUDA Execution Model: Warps



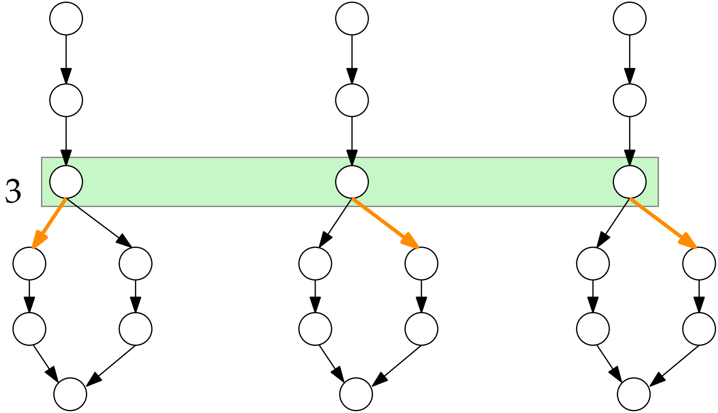
CUDA Execution Model: Warps



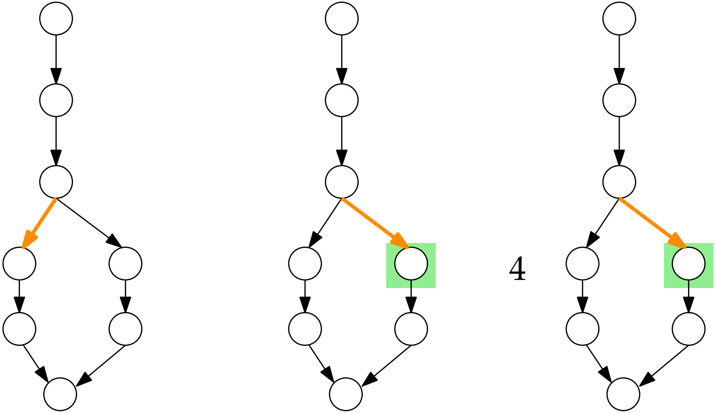
CUDA Execution Model: Warps



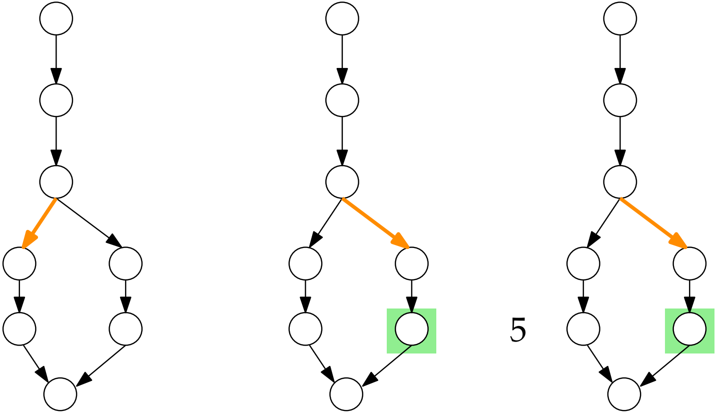
CUDA Execution Model: Warps



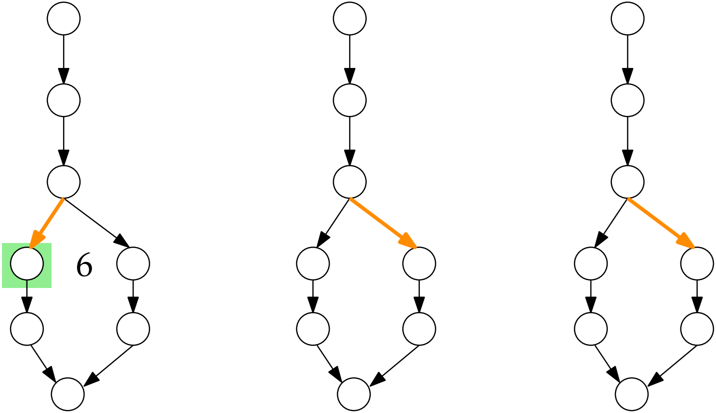
CUDA Execution Model: Warps



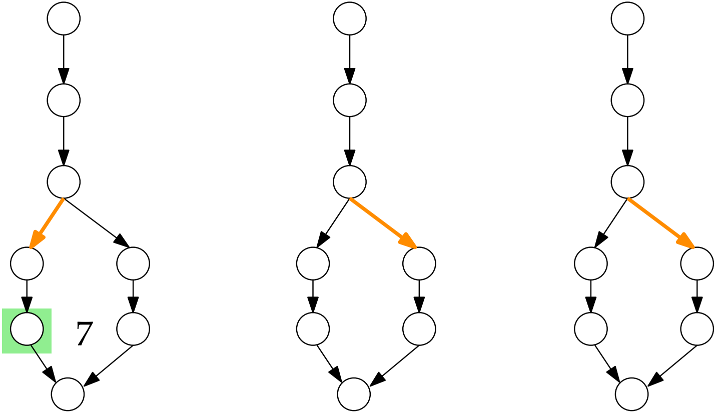
CUDA Execution Model: Warps



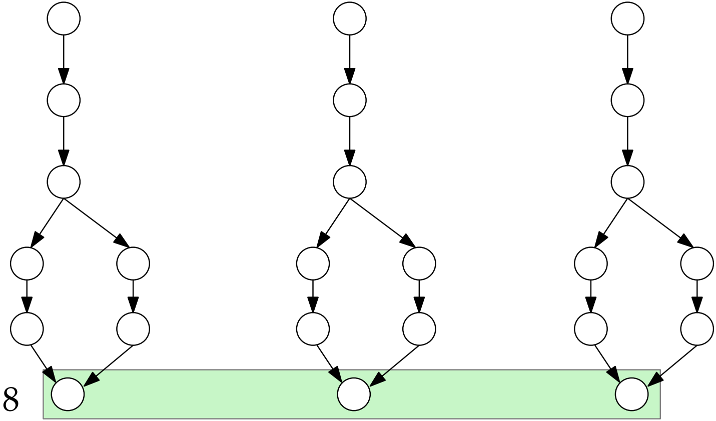
CUDA Execution Model: Warps



CUDA Execution Model: Warps



CUDA Execution Model: Warps



CUDA Execution Model: Scheduling Warps

- At each clock tick, SM determines which warp is ready to execute
- This is done by “scoreboarding”: hardware table that tracks
  - instructions
  - resources
  - which instructions use which registers
- Using scoreboard, SM can figure out who’s ready for execution next.

Sorting with CUDA

Many different implementations of sorting algorithms

- Radix sort
- Merge sort
- Quick sort
- Sample sort
- Bitonic sort
- Hybrid sorting methods

For fixed keys, radix sort is fastest

Radix Sort[MG10]

7	1	1	1
1	0	0	1
3	0	1	1

Radix Sort[MG10]

7	1	1	1
1	0	0	1
3	0	1	1

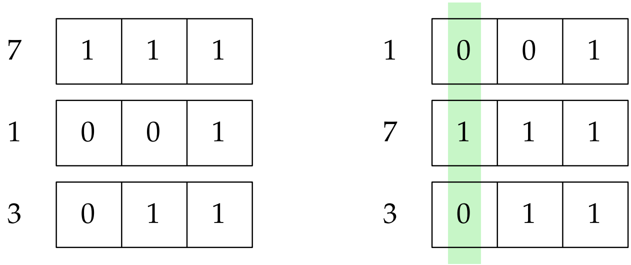
Radix Sort[MG10]

7	1	1	1
1	0	0	1
3	0	1	1

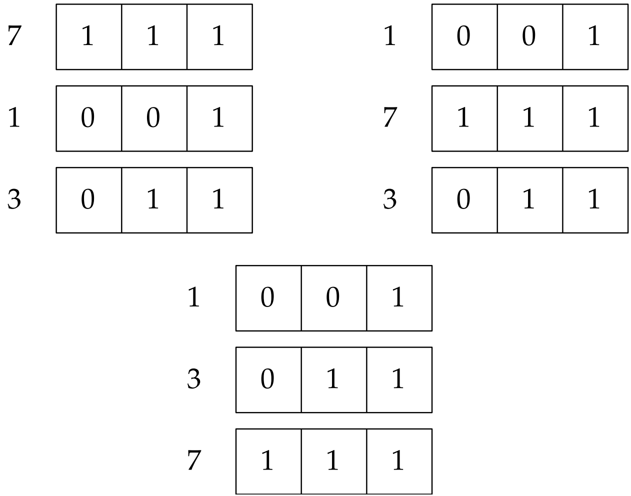
Radix Sort[MG10]

7	1	1	1		1	0	0	1
1	0	0	1		7	1	1	1
3	0	1	1		3	0	1	1

Radix Sort[MG10]



Radix Sort[MG10]



Prefix Counting

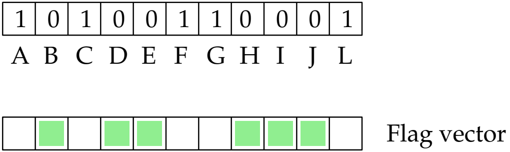
1	0	1	0	0	1	1	0	0	0	1
A	B	C	D	E	F	G	H	I	J	L

Prefix Counting

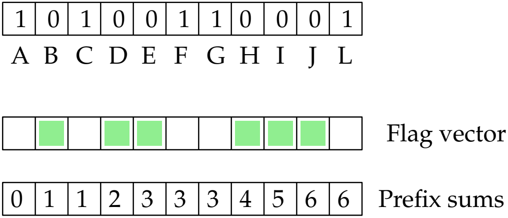
1	0	1	0	0	1	1	0	0	0	1
A	B	C	D	E	F	G	H	I	J	L

0	0	0	0	0	0	1	1	1	1	1
B	D	E	H	I	J	A	C	F	G	L

Prefix Counting



Prefix Counting



Prefix Counting

Prefix Sums

For each digit

1. Construct flag vector locally and write to shared memory
2. Do parallel reduce on flag vector to find offsets
3. Move items to correct locations in global array
4. Repeat

Parallel Reduce

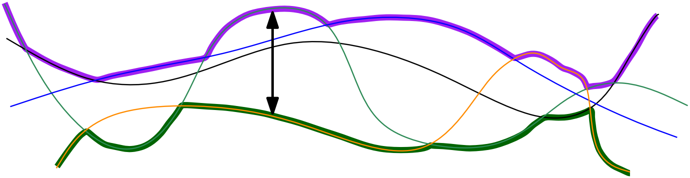
- If all reducers in one block, easy to synchronize
- If not, need to use global memory to communicate: Expensive !
- Create multiple kernels for different levels of the reduce tree (kernel creates sync)

Optimizations

- Distribute reduce operations to blocks
- Factor out branches to reduce divergence penalty
- Unroll operations in reduce when possible.

Overall 1 GKeys/second, 3-4x over Larrabee

Level Selection and Two-Sided Tests[GKMV03]



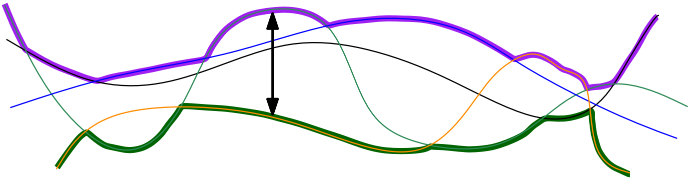
- Let $f_1, \ldots f_n$ be a set of functions from $\mathbb{R}^2 \to \mathbb{R}$

$$U(x) = \max_i f_i(x)$$

$$L(x) = \min_i f_i(x)$$

$$E(x) = U(x) - L(x)$$

Level Selection and Two-Sided Tests[GKMV03]



- Let $f_1, \ldots f_n$ be a set of functions from $\mathbb{R}^2 \rightarrow \mathbb{R}$

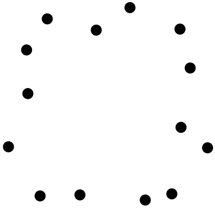
$$U(x) = \max_i f_i(x)$$

$$L(x) = \min_i f_i(x)$$

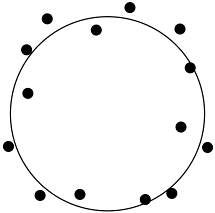
$$E(x) = U(x) - L(x)$$

$$M_k(x) = \text{kth-smallest } (f_1(x), f_2(x), \ldots f_n(x))$$

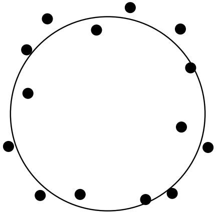
Level Selection and Two-Sided Tests[GKMV03]



Level Selection and Two-Sided Tests[GKMV03]

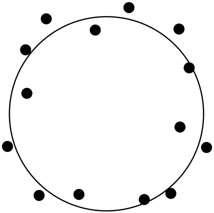


Level Selection and Two-Sided Tests[GKMV03]



Best-Fit(P) = $\min_{c,R} \sum |\|p - c\| - R|$

Level Selection and Two-Sided Tests[GKMV03]



$$\text{Best-Fit(P)} = \min_{c,R} \sum \big| \|p - c\| - R \big|$$

Solution is to minimize over the *median* layer of an associated arrangement

QuickSelect

Fragment Program: takes input $x_1, x_2, \ldots x_n$ and k

```
(lo, hi) ← arg min x_i, arg max x_i
while hi - lo > 1 do
    Pick random mid between lo,hi
    c = number of elements x such that x_lo ≤ x ≤ x_mid {two-sided test}
    if c ≥ k then
        hi ← mid
    else
        lo ← mid
    end if
end while
Return lo
```

QuickSelect

- QuickSelect as a fragment program extracts the k^{th} level of the arrangement.
- It uses three conditionals (one for the branching, and two for the two-sided test
- Two-sided test evaluated many times.
- Overall complexity is $O(\log n)$ passes on average

Lemma

A fragment procesor that only uses a one-sided test, or is not randomized, must take n passes.

Tradeoff between penalty of more conditional branching and number of passes

CUDA Execution Model: Design Choices

- To make maximum use of SIMT, minimize branching
- Memory bank conflicts have to be dealt with
- If there are too many blocks, you pay switching overhead on an SM
- Two-level model allows for flexibility: CUDA program can be adapted to different hardware configurations easily
  - (or even run on a single core machine!)

This Lecture

- Examples of the streaming SIMD view of the GPU
  - Lower envelope computations
  - Multipass streaming median
- The CUDA model:
  - The programmer's view
  - Matrix multiplication
  - The hardware view
  - Radix Sorting

Next Lecture(s)

Solving different problems using CUDA:

- Multipole methods
- Sparse Matrix Operations
- Graphs I: BFS
- Graphs II: Coloring

Questions?

References I

P. Agarwal, S. Krishnan, N. Mustafa, and S. Venkatasubramanian.
Streaming geometric optimization using graphics hardware.
Algorithms-ESA 2003, pages 544–555, 2003.

I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan.
Brook for gpus: stream computing on graphics hardware.
In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004.

OpenGL Foundation.
Opengl shading language.
http://www.opengl.org/documentation/glsl/.

S. Guha, S. Krishnan, K. Munagala, and S. Venkatasubramanian.
Application of the two-sided depth test to csg rendering.
In *Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 177–180. ACM, 2003.

References II

- M.D. McCool and S. Du Toit.
  Metaprogramming GPUs with Sh.
  AK Peters Wellesley, 2004.

- D.G. Merrill and A.S. Grimshaw.
  Revisiting sorting for gpgpu stream architectures.
  In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 545–546. ACM, 2010.

- W.R. Mark, R.S. Glanville, K. Akeley, and M.J. Kilgard.
  Cg: A system for programming graphics hardware in a c-like language.
  In *ACM Transactions on Graphics (TOG)*, volume 22, pages 896–907. ACM, 2003.

- Microsoft.
  Hlsl.
  `http://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx`.

References III

J. Nickolls, I. Buck, M. Garland, and K. Skadron.
Scalable parallel programming with cuda.
Queue, 6(2):40–53, 2008.

NVIDIA.
Parallel programming and computing platform | CUDA.
http://www.nvidia.com/object/cuda_home_new.html.