GPGPU

General Purpose Graphics Processing Unit

Thomas Swift
Sean Brennan
Andrew Wong

Outline

- GPGPU Overview
  - Terminology
  - Graphics Pipeline
  - CPU vs GPU
- CUDA
- NVIDIA's Kepler & AMD's GCN Architectures
- PTask

GPUs in Action

(product placement)

Graphics Processing Unit

- Traditionally used for 3D rendering, but now also used for large computations
  - Originally special function units with specialized HW & ISAs
  - Good at applying same operation to large number of independent elements, in parallel
- Why GPUs?
  - High performance/throughput for massively parallel computations
  - Much higher arithmetic capability and memory bandwidth than even high-end CPUs

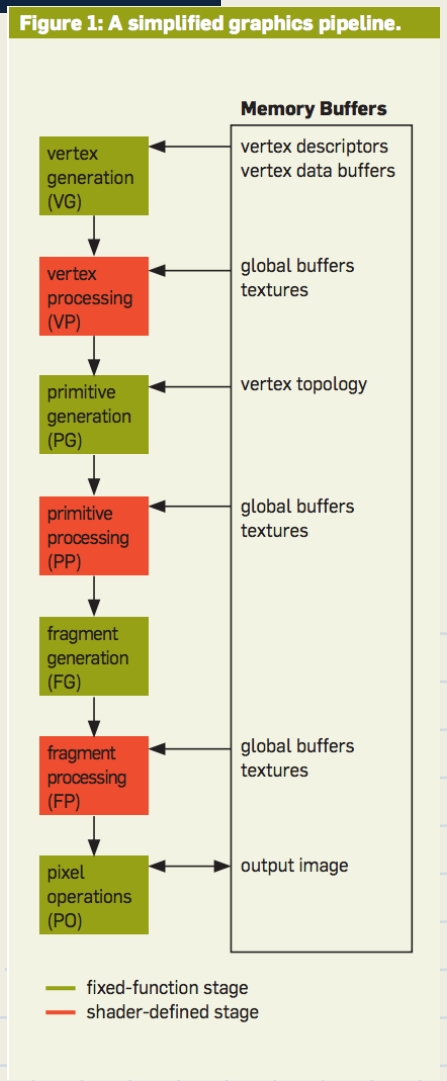
Terminology

- Streaming Processor abstraction
  - Programming model designed to abstract away all graphics terminology of GPU
  - Stream
    - Ordered set of data
  - Kernel
    - Function applied element-wise to a set of streams and that outputs one or more streams

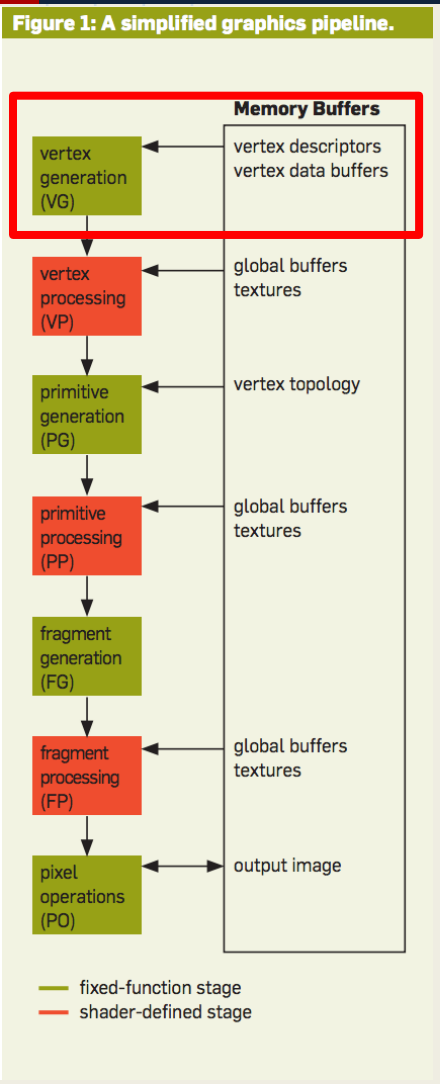
- SIMD - Single Instruction, Multiple Data
  - SPMD - Single Program, Multiple Data

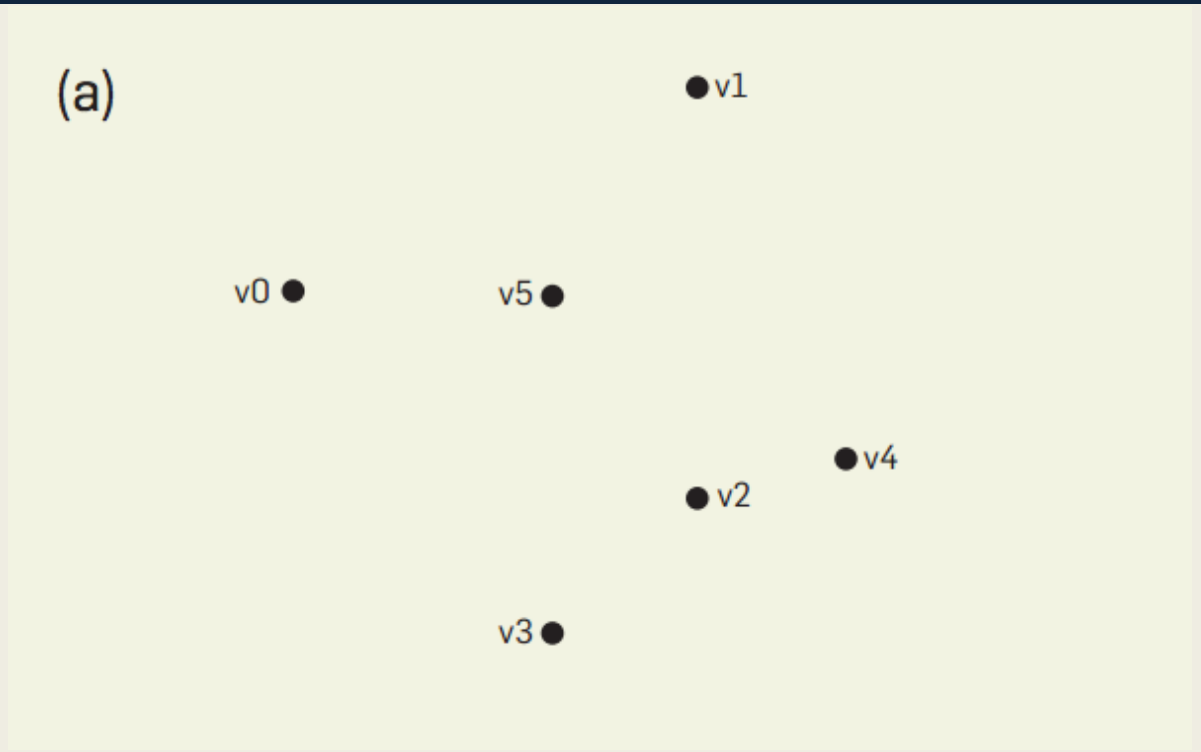
Graphics Pipeline

- A series of generation and processing stages
  - Connected by stream-entities
- Processing stages are programmable
  - Shader functions used to alter appearance of graphical output.



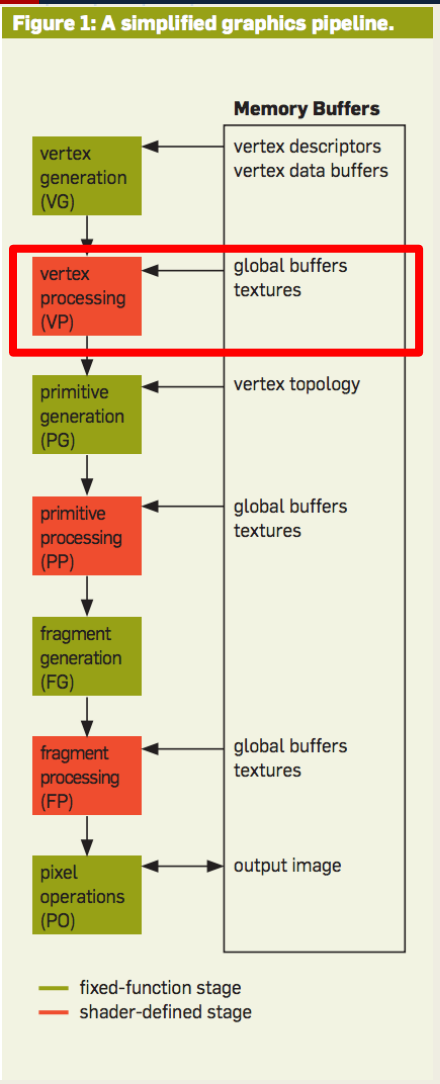
Vertex Generation

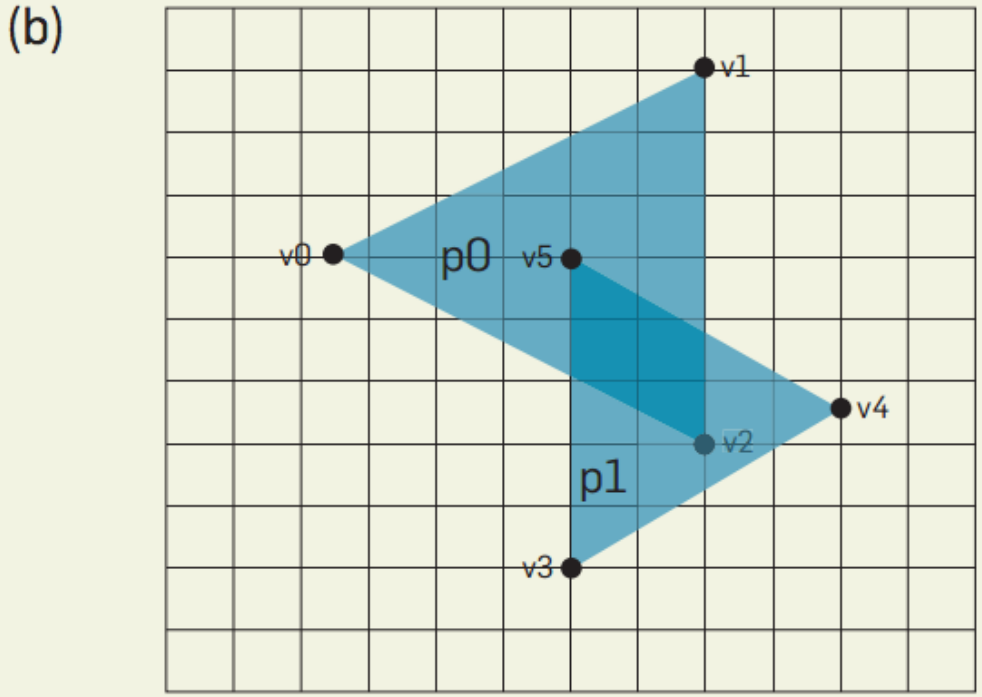




- VG - prefetches vertex and texture data from memory and constructs a stream of vertex data.

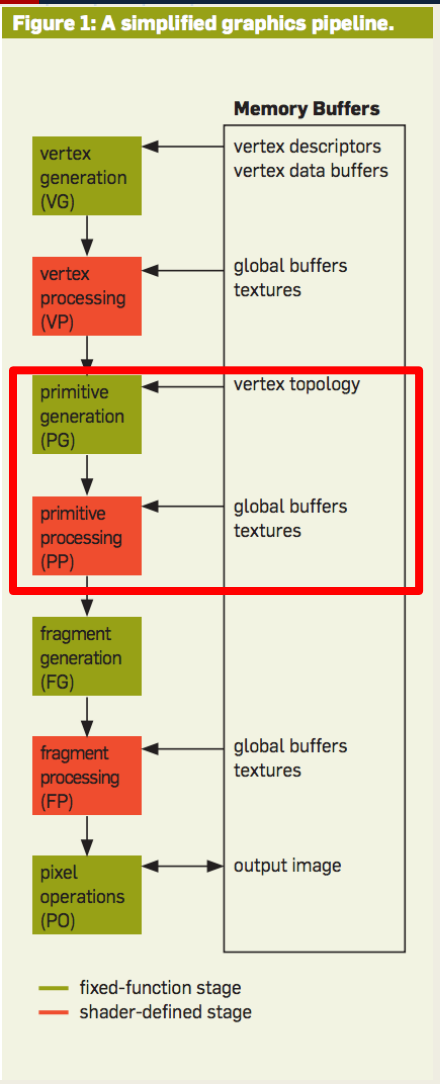
Vertex Processing

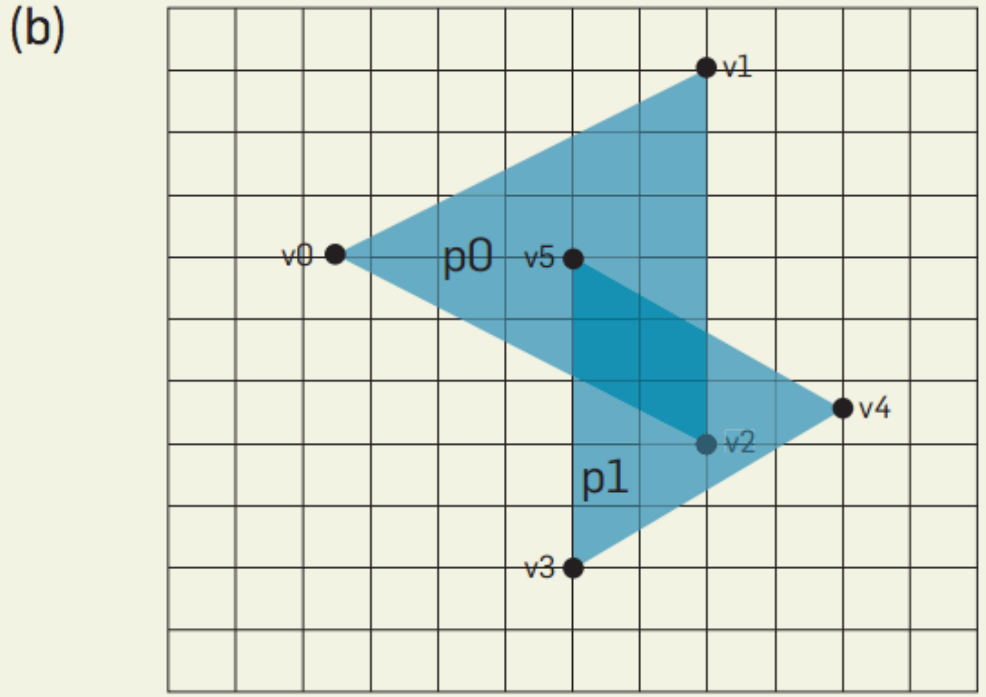




- VP - programmable operation on each vertex (e. g. computing projection from 3D space to screen).

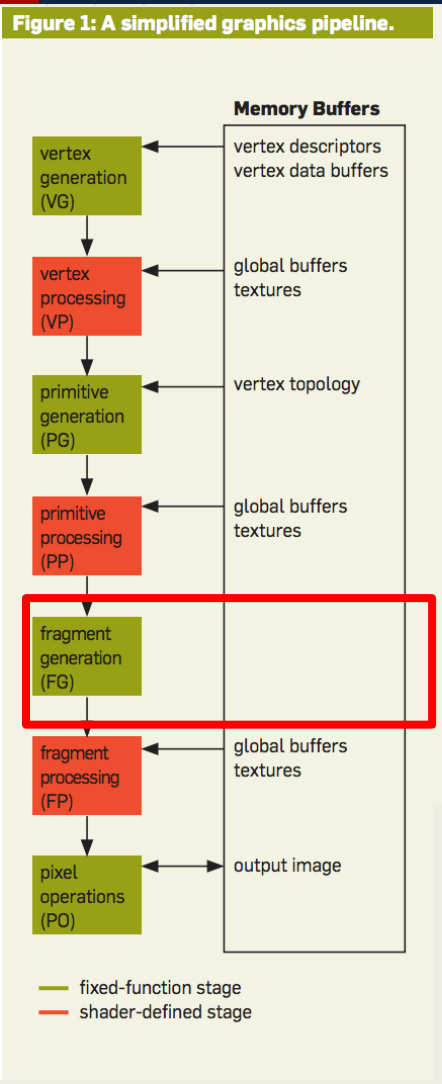
Primitive Generation
Primitive Processing

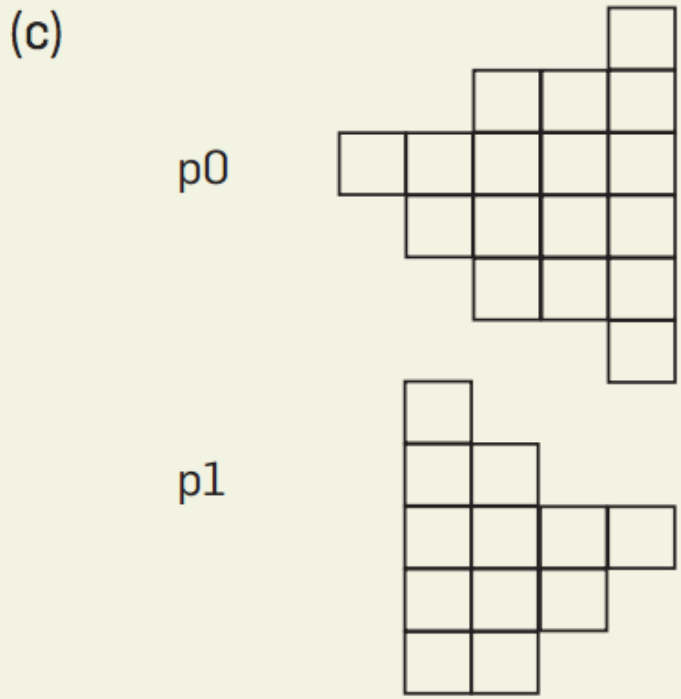




- PG - groups vertices into ordered streams of primitives
- PP - Produces zero or more output primitives.

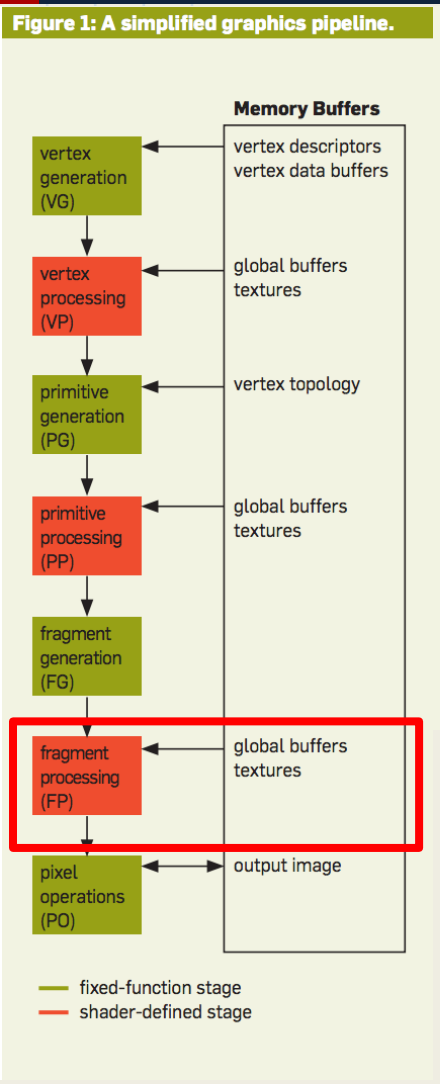
Fragment Generation

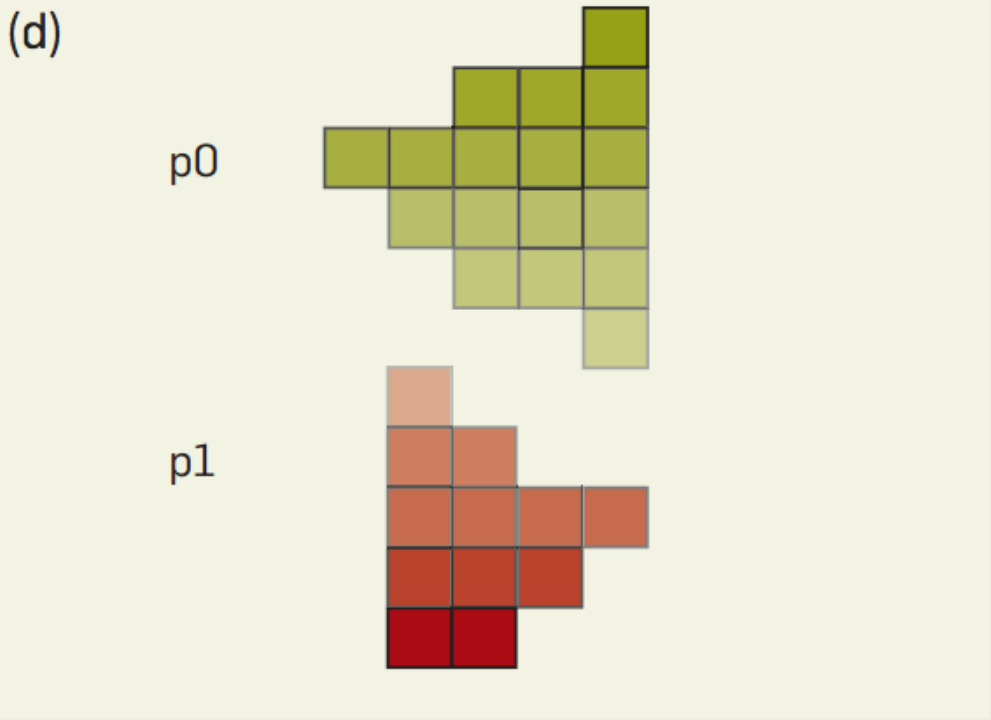




- FG - performs rasterization through sampling. Distance from camera and other parameters are saved

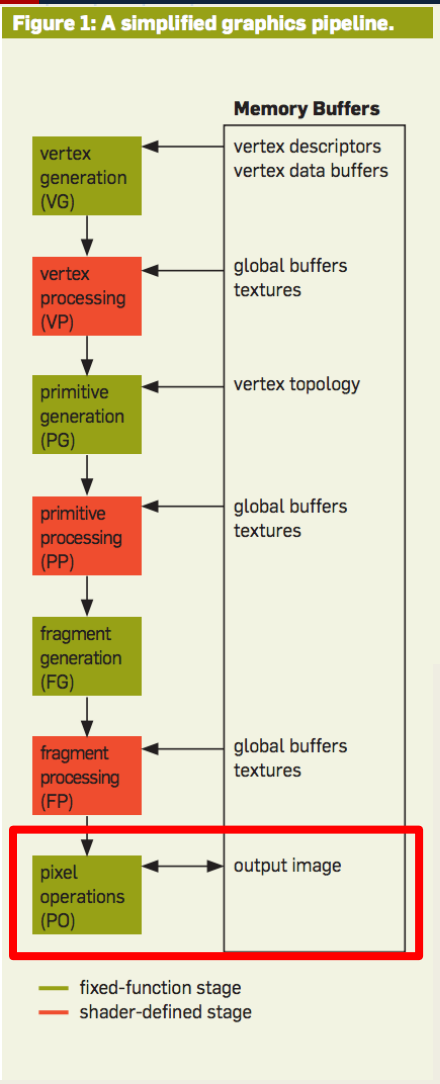
Fragment Processing



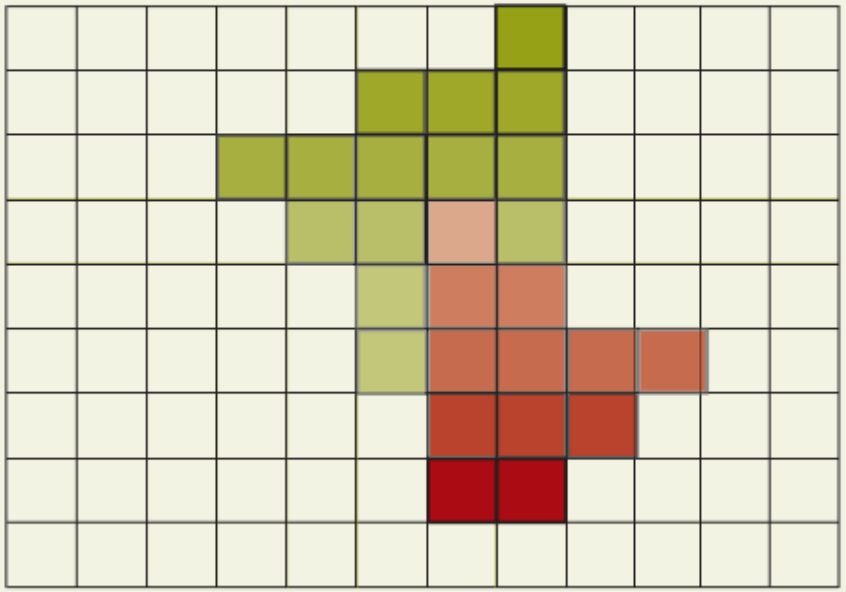


- FP - simulates light interaction to determine color and opacity.

Pixel Operations



(e)



- PO - calculates image pixel values based on distance and obstructions.

Hardware Multithreading

- Threads stall due to memory accesses
- Hardware contains multiple execution units to perform instructions from other threads when a thread stalls.

Figure B: Thread Execution on the example GPU core

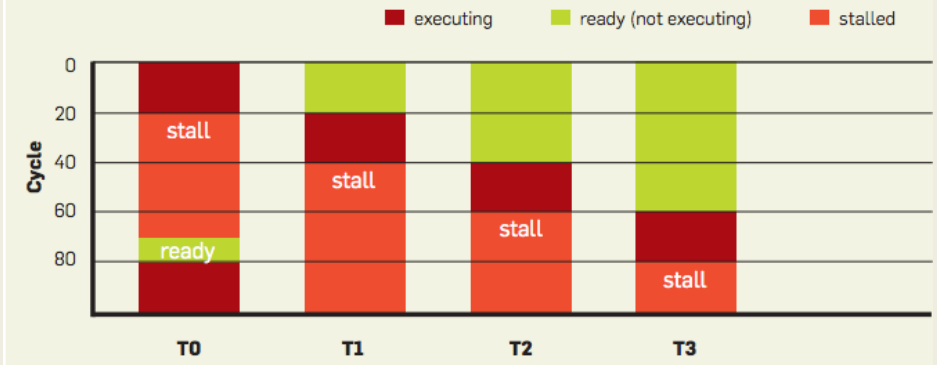
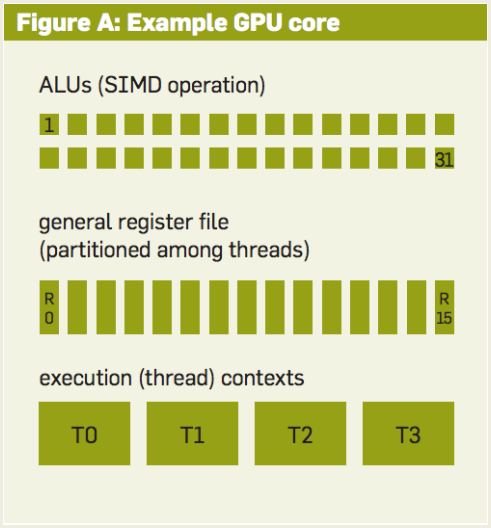


Figure A: Example GPU core



CPU vs GPU

CPUs

- Parallelism through time multiplexing
- Emphasis on low memory latency
- Allows wide range of control flows + control flow optimizations
- Low-latency caches that allow performant out-of-order execution
- Very high clock speeds

GPUs

- Parallelism through space multiplexing
- Emphasis on high memory throughput
- Very control flow restricted
- High-latency caches that tend to be read-only
- Mid-tempo clock speeds

Other GPU Points

- Power efficient for large parallel operations
- Inexpensive on a TFLOP basis
- Difficult to program - but getting better!

CUDA

- **C**ompute **U**nified **D**evice **A**rchitecture
- Platform and programming model by Nvidia
- Introduced in 2006 w/ GeForce 8800 GTX
- First architecture targeted at general purpose use
  - CUDA C provides high-level language familiar to most programmers
  - ALUs built for more general types of computation
  - *Unified Shader Model* improves use of GPU resources

Unified Shader Models

- *Unified Shader Architecture*: all GPU units designed to handle any shader
- *Unified Shader Model*: all shaders have similar instruction set
- Unified *Model* does NOT require Unified *Architecture*!
- Advantages over "classical" model:
  - more dynamic and flexible use of GPU resources
  - open to different workflows
    - both of these make USA/USM well-suited to GPGPU programming

Programming in CUDA

- CUDA C: good old C + a few new functions, structs, and primitives
- Kernel functions: `__global__` and `__device__`
  - `__global__`: code executed on GPU from CPU
  - `__device__`: code executed on GPU from other GPU functions
- Grid abstraction: *spatial multiplexing*
  - grid → blocks → threads
  - grid (2D) + block (3D) = 5 degrees of indexing freedom
- SIMD paradigm: **S**ingle **I**nstruction **M**ultiple **D**ata
  - well-suited to data-parallel tasks
  - conditionals are costly and should be avoided

CUDA Workflow

1. allocate data in main memory *and on GPU*

```
15      // allocate arrays on host
16      a_h = (float *)malloc(size);
17      b_h = (float *)malloc(size);
18
19      // allocate array on device
20      cudaMalloc((void **) &a_d, size);
```

2. move data from MM → GPU

```
26      // copy data from host to device
27      cudaMemcpy(a_d, a_h, sizeof(float) * N, cudaMemcpyHostToDevice);
```

3. issue kernel over given block + thread count

```
34      // Part 2 of 2. Call incrementArrayOnDevice kernel
35      incrementArrayOnDevice <<< nBlocks, blockSize >>> (a_d, N);
```

CUDA Workflow

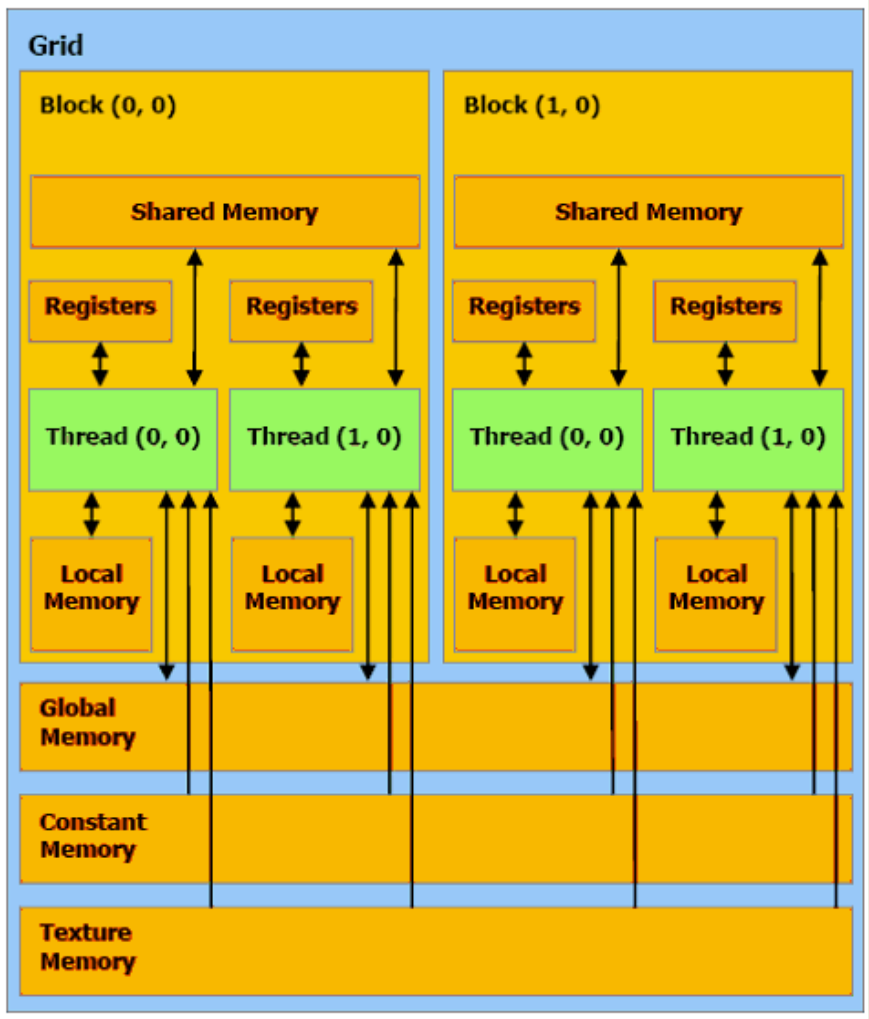
4. kernels execute in parallel

```
2  __global__ void incrementArrayOnDevice(float *a, int N) {
3      // calculate index to work over; increment
4      int idx = blockIdx.x * blockDim.x + threadIdx.x;
5      if (idx < N)
6          a[idx] = a[idx] + 1.f;
7  }
```

5. copy data from GPU → MM

```
37     // Retrieve result from device and store in b_h
38     cudaMemcpy(b_h, a_d, sizeof(float) * N, cudaMemcpyDeviceToHost);
```

CUDA Memory Layout



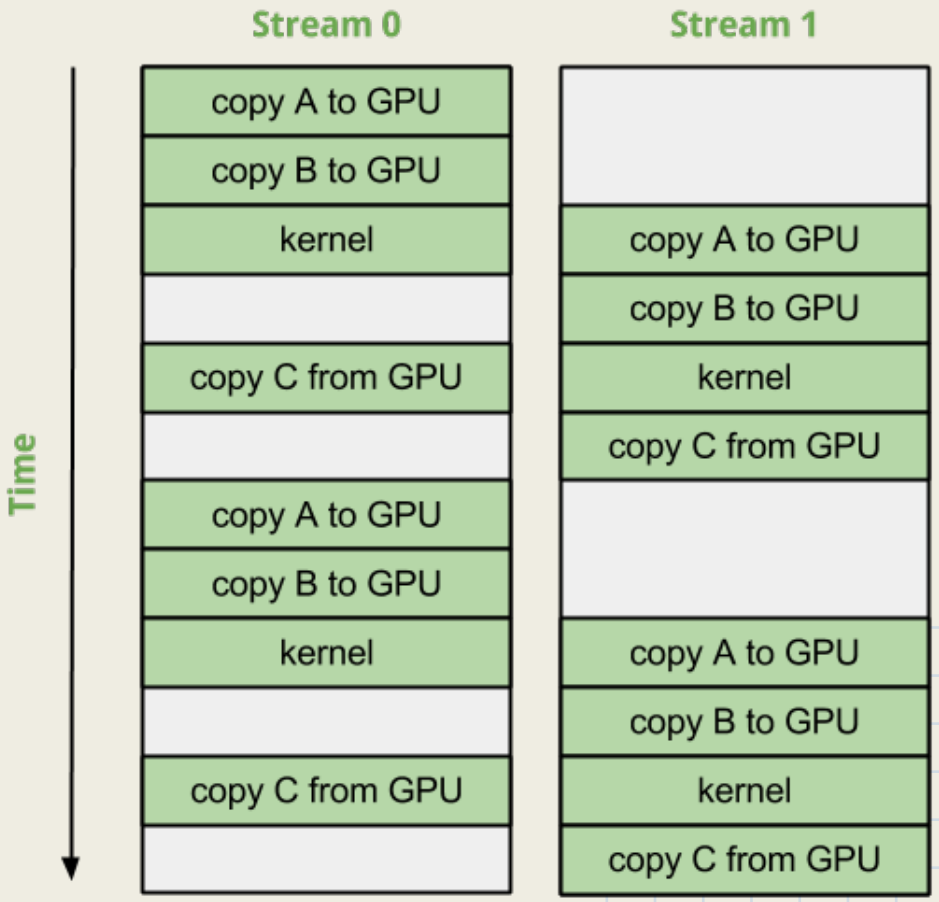
- Per-thread registers
  - Very fast; lifetime of thread
- Per-block shared memory
  - Very fast; lifetime of block
- All-block global memory
  - ~100x slower than shmem
  - high throughput achieved through coalescing
- Per-thread local memory
  - gotcha! as slow as global
- All-block constant memory
  - half-warp broadcast reduces bandwidth
- All-block texture memory
  - useful when exploiting spatial locality

CUDA Global Memory

- Global memory achieves high throughput through *coalescing*
  - Works under certain global memory access patterns
  - *Half-warp coalescing*: accesses by all threads in a half-warp (16 threads) are coalesced
    - threads must access 32, 64, or 128-bit data types
    - data accessed must be properly word-aligned
    - threads must access words of coalesced access in sequence
  - Responsibility is placed upon CUDA programmer
    - no coalescing → huge hit to memory throughput
  - Can use CUDA Profiler to track amount of coalesced/non-coalesced accesses

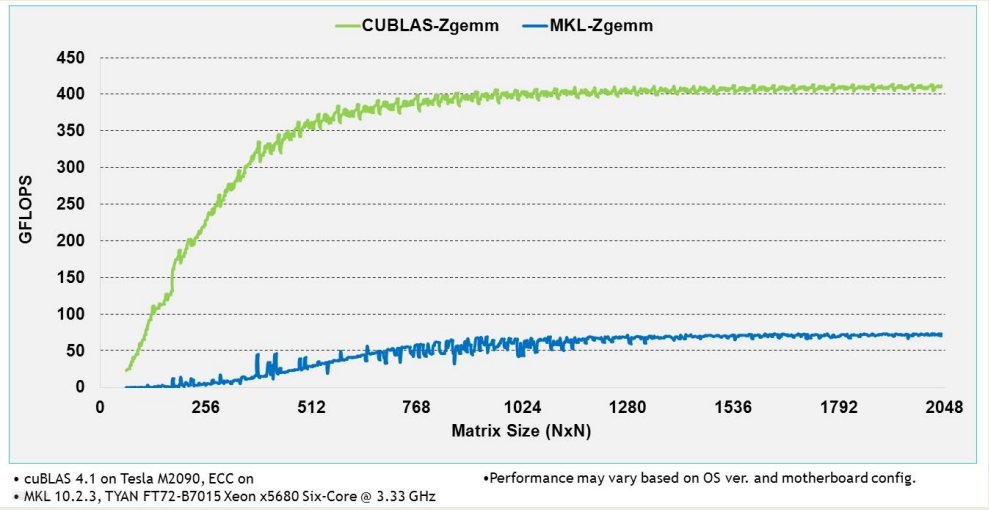
CUDA Streaming

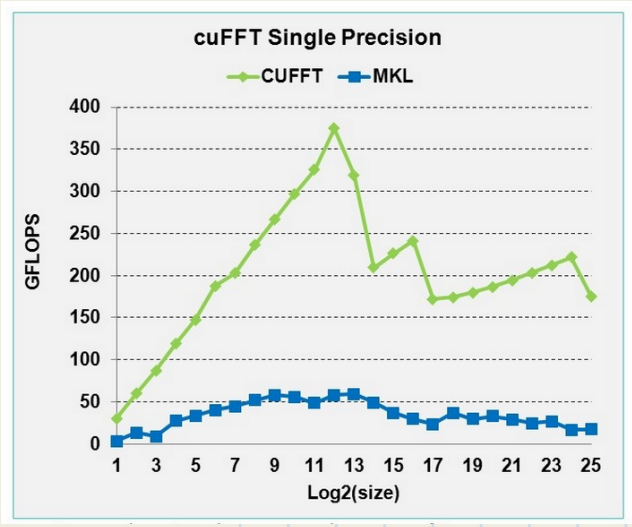
- Task parallelism can be achieved through *streaming*
  - stream: queue of tasks to be performed on GPU
- asynchronous copies hide latency of memory movement
  - requires page-locked ("pinned") memory
    - ruins virtual memory abstraction
    - steps on other processes' toes
- later versions support two streaming memory accesses



CUDA Applications

- CUBLAS: CUDA Basic Linear Algebra Subprograms
  - Adapted from well-known FORTRAN BLAS package
  - Three tiers of routines
    - level 1: vector scaling, inner product
    - level 2: matrix-vector products, matrix triangularization
    - level 3: matrix-matrix multiplication
- CUFFT: CUDA Fast Fourier Transforms
  - 1D, 2D, and 3D transforms for real-valued and complex data

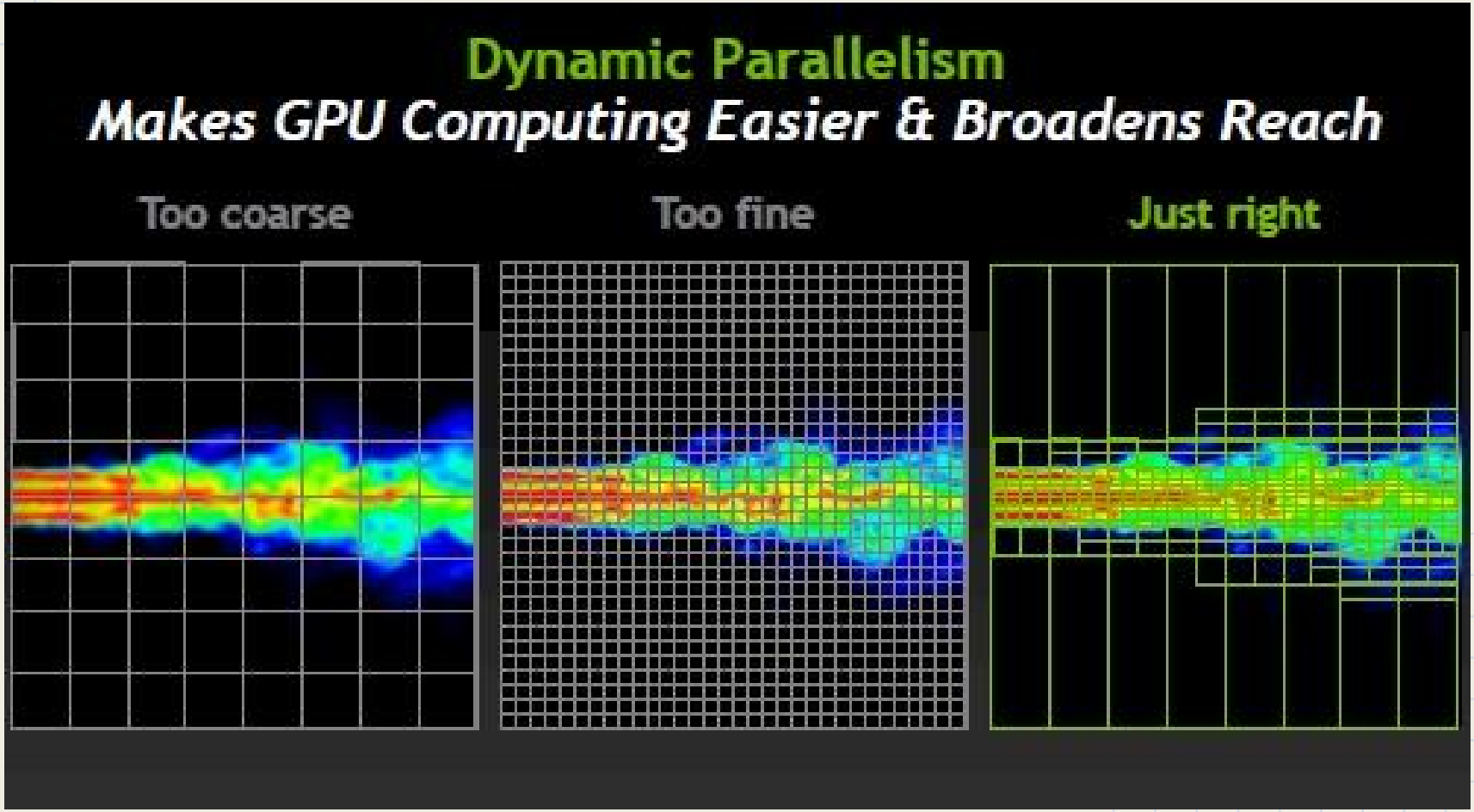




NVIDIA Kepler GK 110 Architecture

- Dynamic Parallelism
  - Without ANY CPU help, GPU can
    - dynamically create new worker threads
    - synchronize among thread results
    - schedule worker threads
  - Advantages:
    - GPU can adapt to varying amounts and types of parallel workloads (choose optimal # threads and program parameters)
    - CPU can perform other tasks in the meanwhile
    - avoids CPU-GPU data transfers

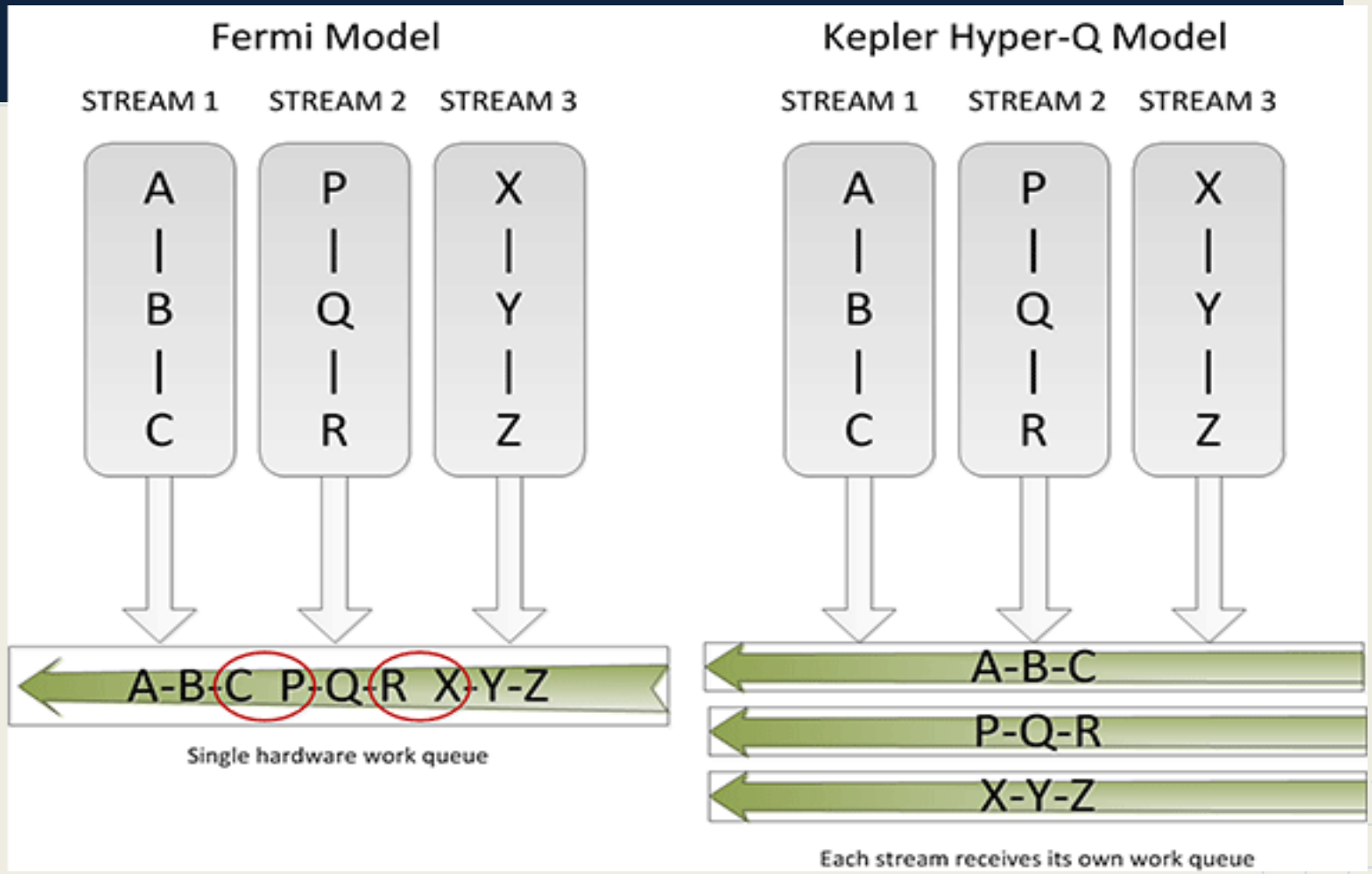
Dynamic Parallelism Example



Kepler: New Features Cont'd

- Hyper-Q
  - Multiple streams (cores/threads/processes) can run work on a single GPU at the same time, using separate HW work queues
  - Prevents streams from blocking each other due to false dependencies
- GPUDirect
  - Allows multiple GPUs on the same machine/network to share data directly without using the CPU or main memory
  - RDMA feature allows third-party devices such as SSDs to directly access GPU memory
  - Greatly improves message passing performance

Hyper-Q Example



Left: only (C,P) and (R,X) can run concurrently
Right: all 3 streams can run concurrently

Kepler Full Chip Diagram

15 SMX Units (Streaming Multiprocessors)

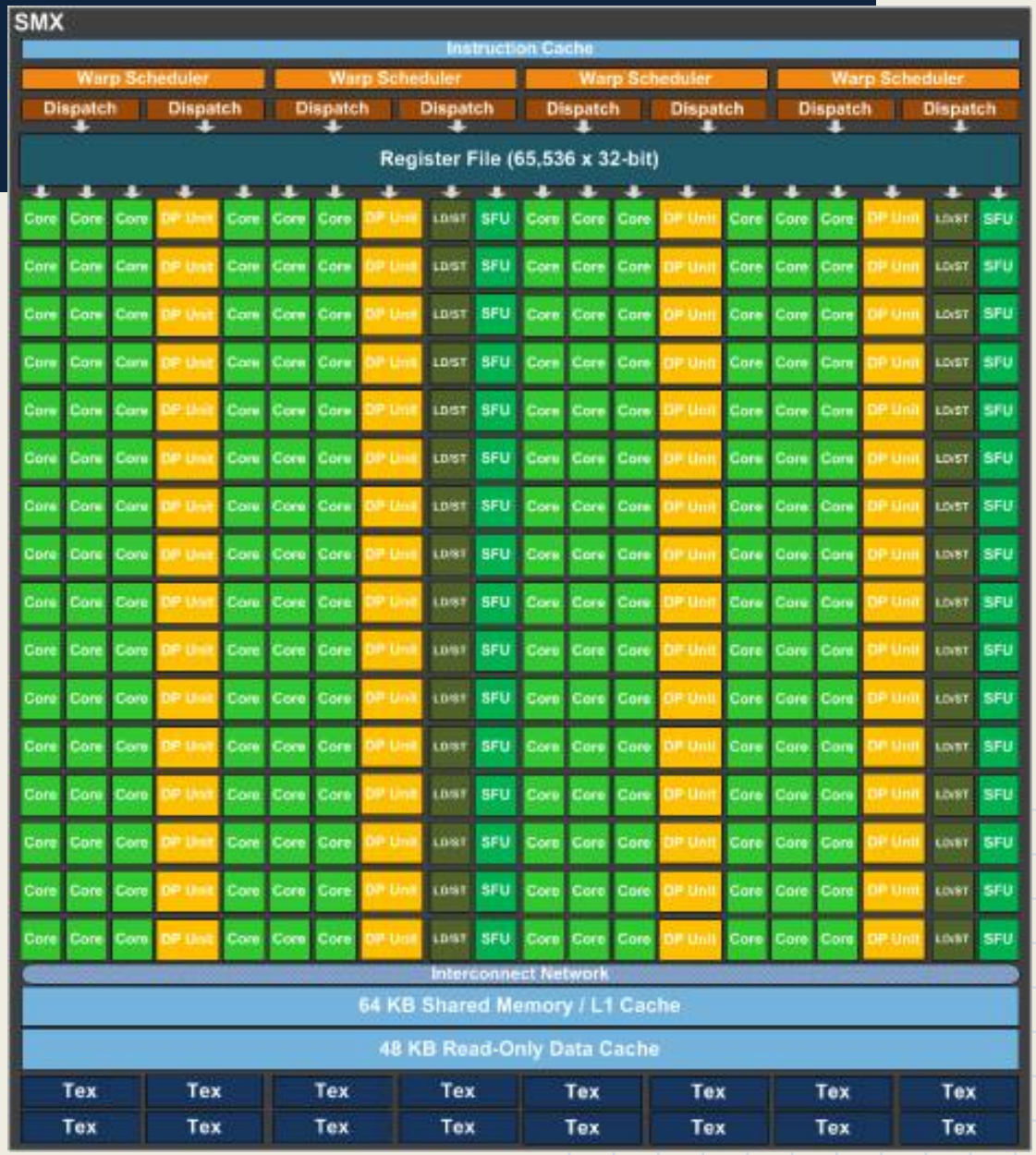
+

6 64-bit memory controllers

Kepler GK110 Full chip block diagram

SMX Unit

- 192 single-precision CUDA cores
- 64 double-precision CUDA cores
- 32 special function units (SFU)
- 32 load/store units (LD/ST)
- 4 warp schedulers, 2 instruction dispatch units each
- 64KB memory
- 48KB read-only data cache



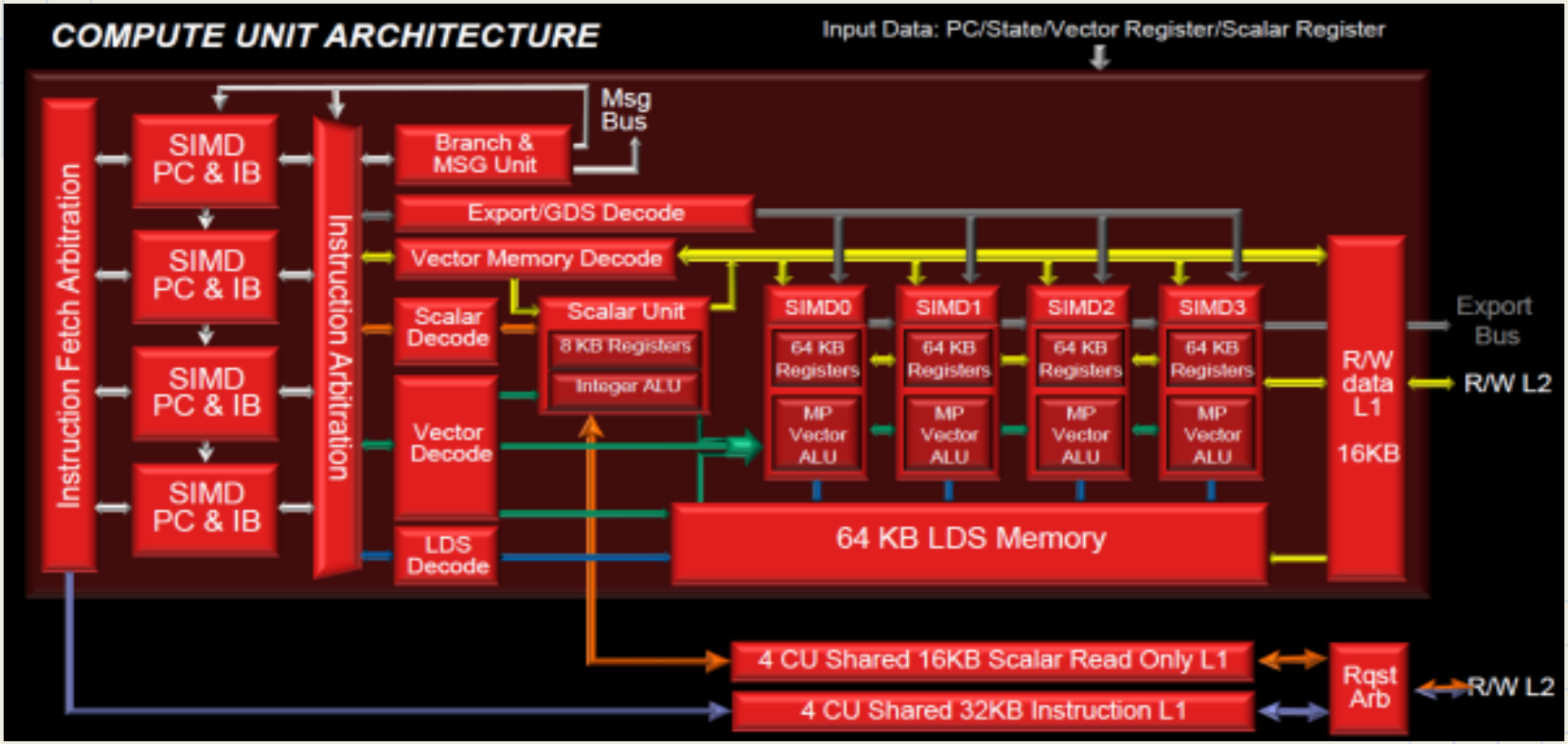
SMX Core Architecture

- Quad Warp Scheduler
  - 32 threads / warp and 2 instruction dispatch units / warp scheduler
  - 4 warps can be executed concurrently; 2 instructions at a time
- 255 registers per thread
- Shuffle Instruction
  - Allows any thread in a warp to read registers of any other thread in the same warp in a single step, instead of going through shared memory with separate LD/ST insts.
- 64KB memory split between shared memory & L1 cache
- 48KB read-only cache for constant data
  - can be managed automatically by the compiler, or manually by the programmer

AMD GCN Architecture

- Tighter CPU-GPU integration
  - Virtual Memory: supports 4KB pages
    - could allow CPU & GPU to share single address space in the future
  - GCN includes I/O MMU, which maps GPU addresses to CPU addresses
  - 64B cache lines
- Cache Coherency: data shared between cores through L2 cache, instead of having to synchronize by flushing to memory

Compute Unit Architecture

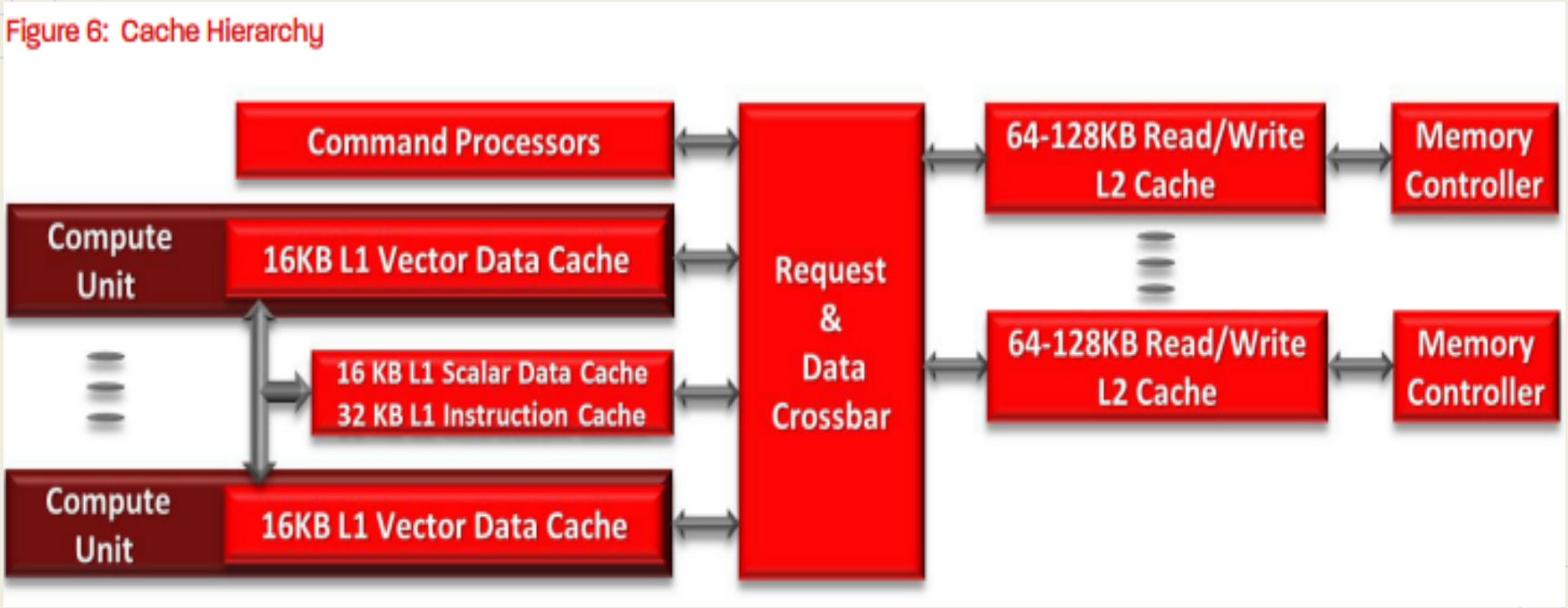


4 SIMD Units (10 wavefronts each), 16KB R/W L1-D cache, 32KB L1-I cache per 4 CU's, 64KB Local Data Share (LDS)

Compute Units (CU's)

- Basic computational building block
- 4 SIMD units; each has PC and IB for 10 wavefronts
- Each cycle, single SIMD picked via RR to issue up to 5 instructions
  - instructions must be of different types, from different wavefronts
- SIMD executes in parallel across multiple wavefronts, instead of in parallel within a single wavefront
- 16KB R/W L1-D cache & 32KB L1-I cache per 4 CU's
  - LRU replacement
- Local Data Share (LDS): 64KB memory used for intra-work-group synchronization

GCN Cache Hierarchy



L1-D cache per CU, L1-I cache per 4 CU's,
L2 cache partitioned and shared by all CU's

Cache Hierarchy

- L1-D cache per CU; L1-I cache per 4 CU's
  - write-through
  - data written to L2 cache at end of wavefront instruction, or at a barrier
  - work-group coherency
- L2 cache shared among all CU's and partitioned into one slice per memory channel
  - write-back
  - absorbs L1-D cache misses
  - synchronizes among different wavefronts --> global coherency
- All caches use LRU replacement policy

GPGPU Challenges

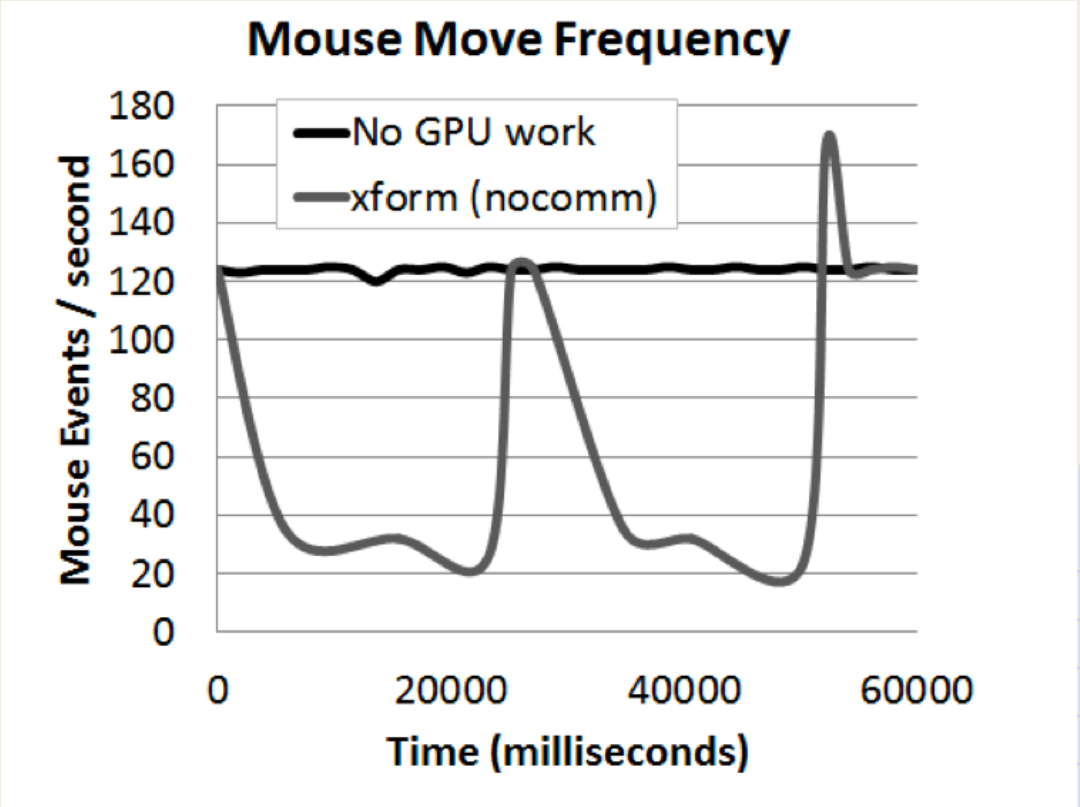
- GPGPUs are more than just I/O devices
  - First class computational devices
  - Fairness and isolation guarantees
- Preempting GPUs is not easy
  - Large number of parallel operations
- Limited interface
  - Drivers are black boxes
  - Existing OS/kernel interfaces use ioctl
- Memory can be disjoint

Additional Motivation

- Data movement tied to algorithms
- High-level languages (CUDA) are hard to use.
- New applications that require OS support
  - encrypted file systems
  - gesture detection

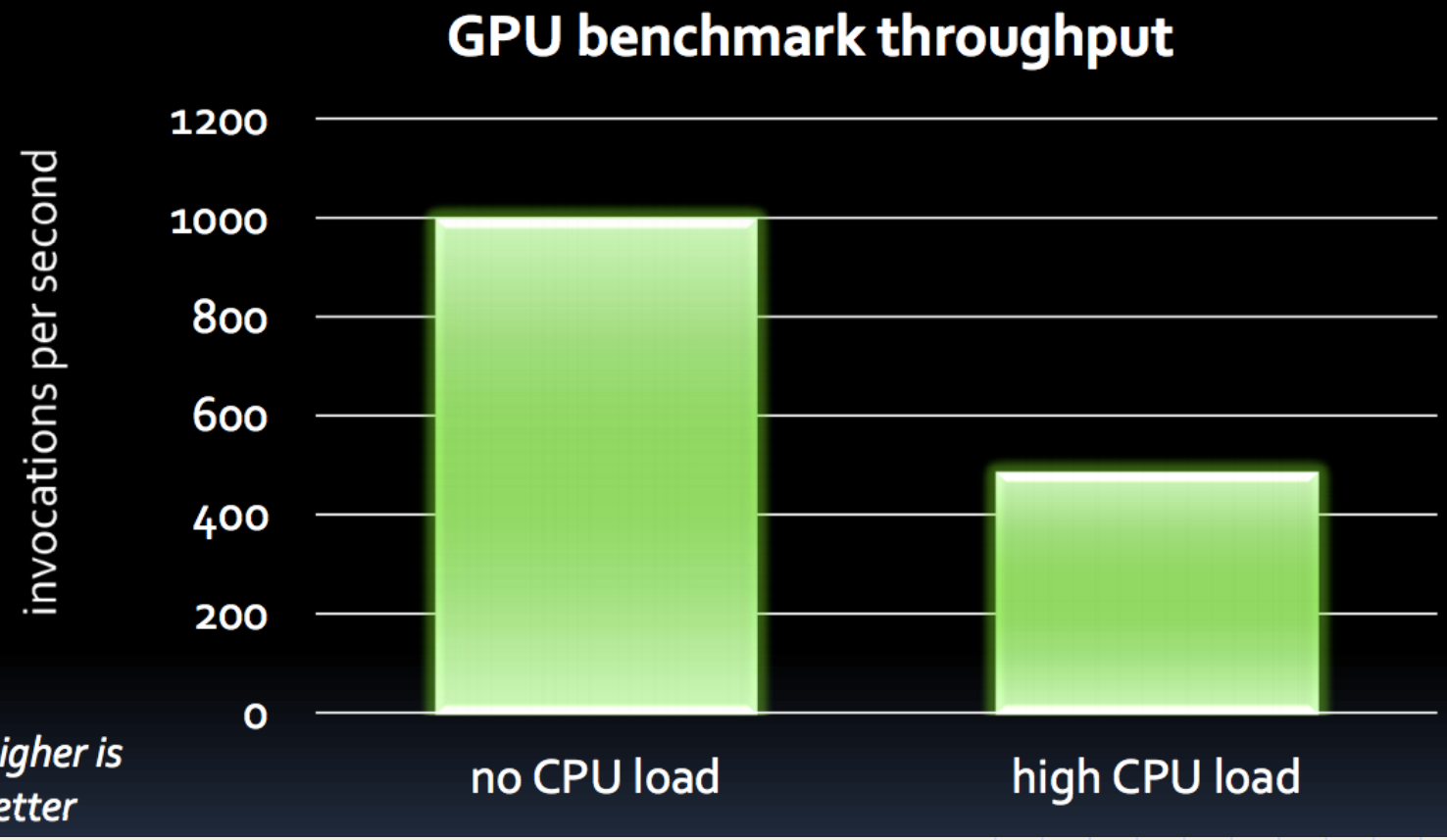
Scheduling Bottlenecks

- GPU accelerated tasks can impair seemingly unrelated tasks
  - GPU work causes system pauses
- CPU work interferes with GPU throughput

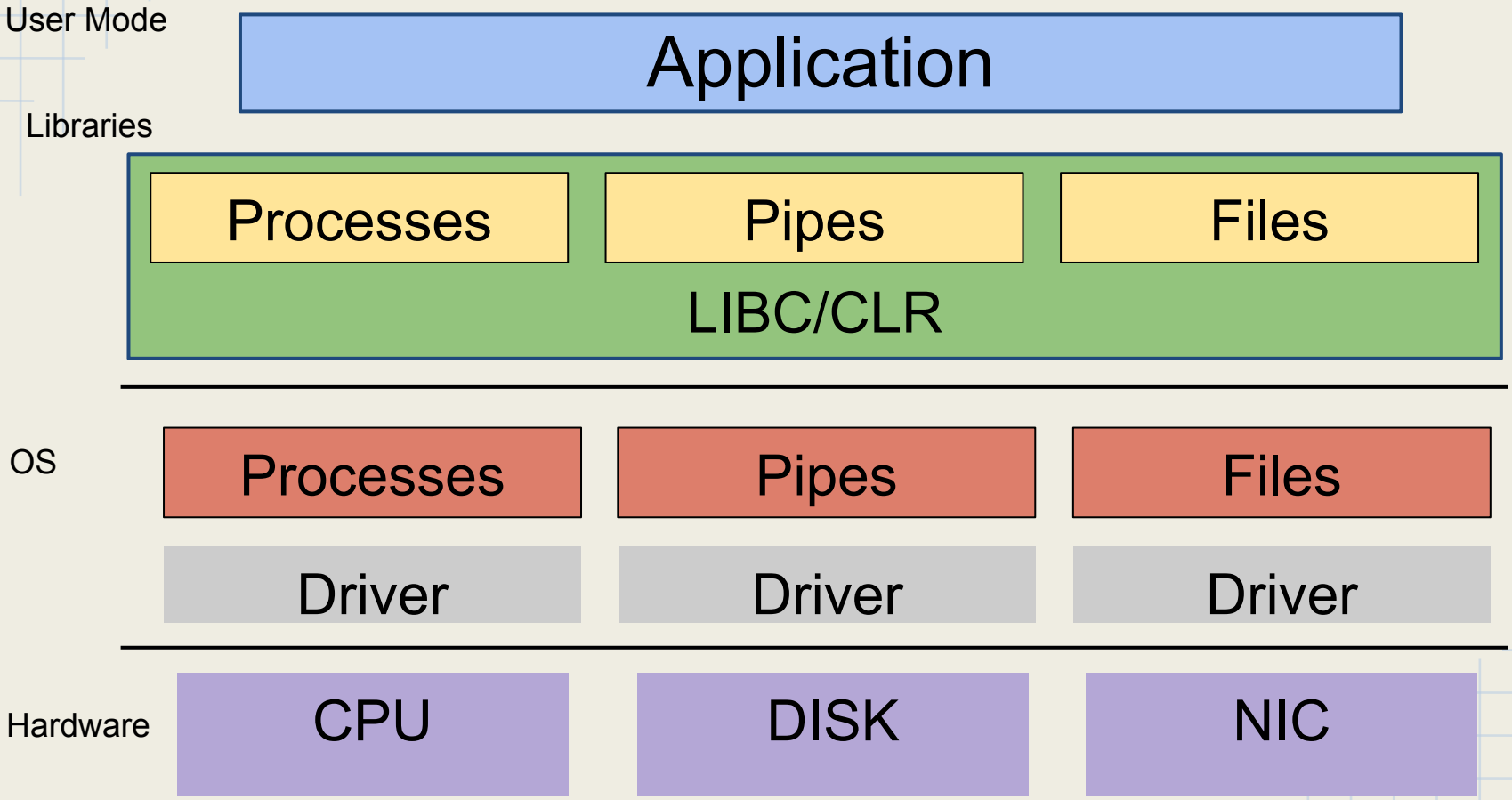


Scheduling Bottlenecks (cont.)

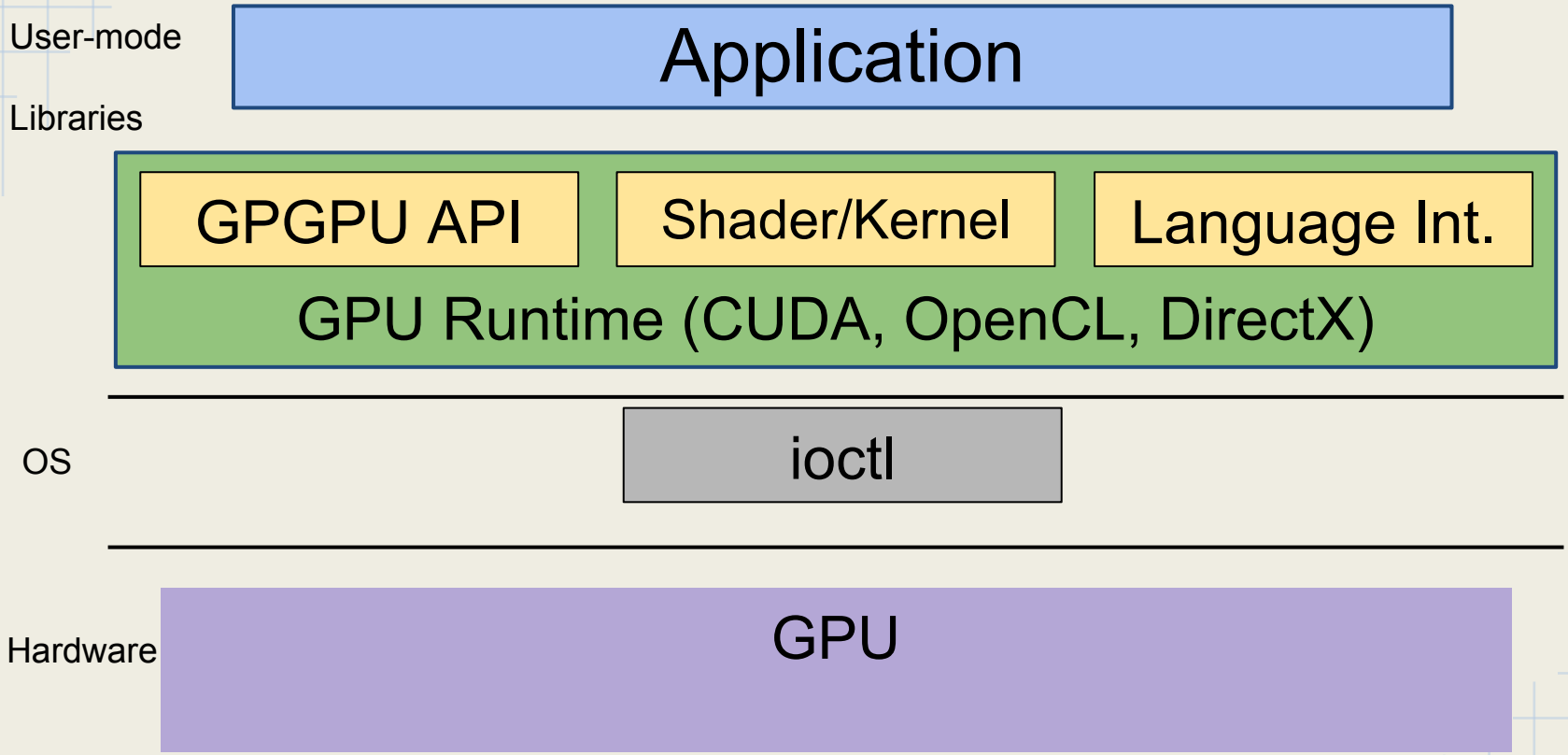
No OS support → No isolation



OS Support


User Mode
Application
Libraries
Processes
Pipes
Files
LIBC/CLR
OS
Processes
Pipes
Files
Driver
Driver
Driver
Hardware
CPU
DISK
NIC

GPGPU Support



PTask

1. GPUs under a single resource manager for fairness and isolation guarantees
2. Simplify development for accelerators/GPGPUs by introducing a programming model that manages devices, performs I/O, and deals with disjoint memory spaces.
3. Create an environment that allows modular and fast code

Dataflow Programming

- Modularity and efficiency

Matrix Multiplication

```
matrix gemm(A, B) {
    matrix AxB = new matrix();
    copyToDevice(A);
    copyToDevice(B);
    invokeGPU(gemm_kernel, A, B, AxB);
    copyFromDevice(AxB);
    return AxB;
}
```

But what about A x B x C?

```
matrix AxBxC(A,B,C) {
    return gemm(gemm(A, B), C);
}
```

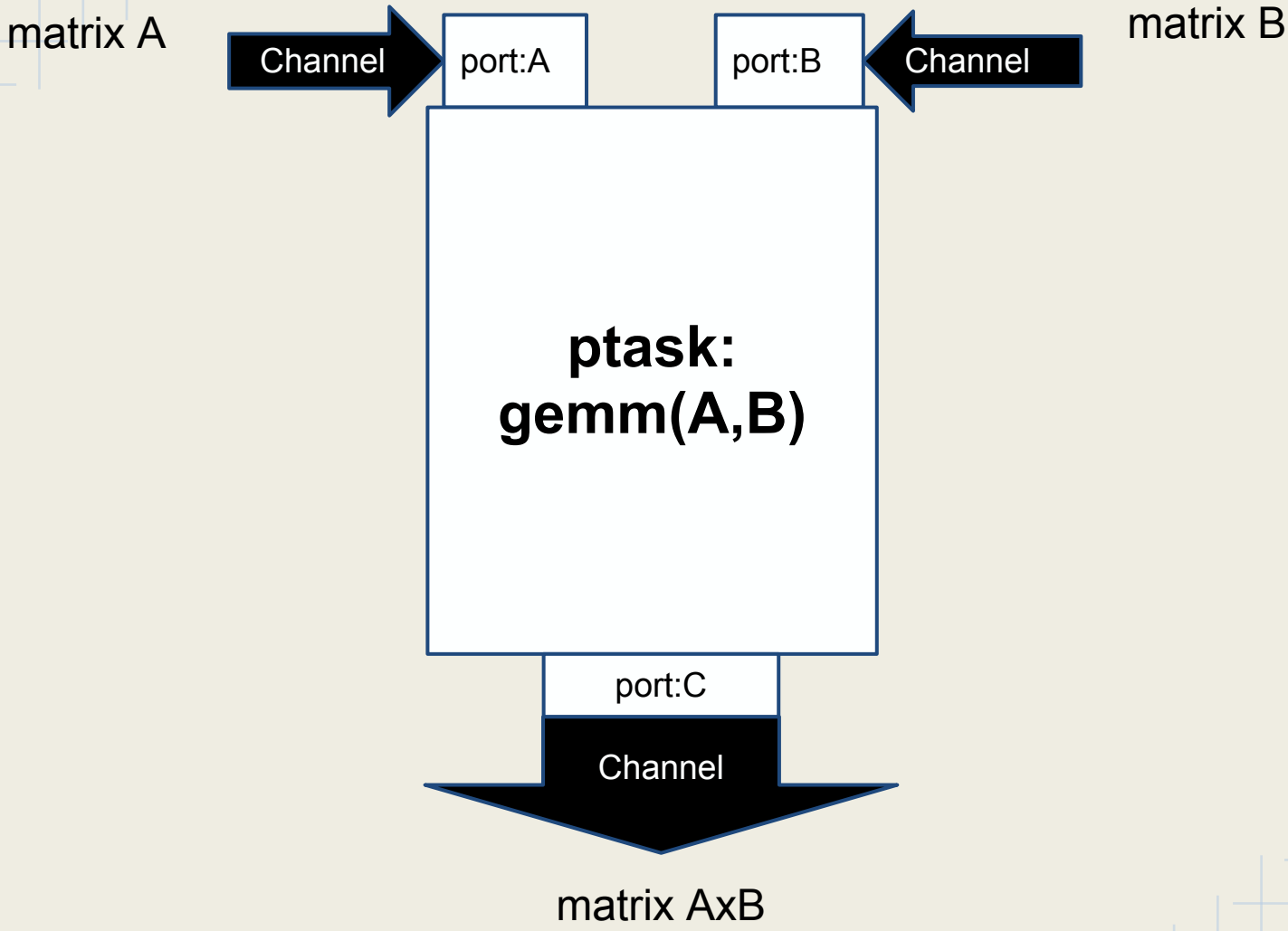
Matrix Multiplication (Cont.)

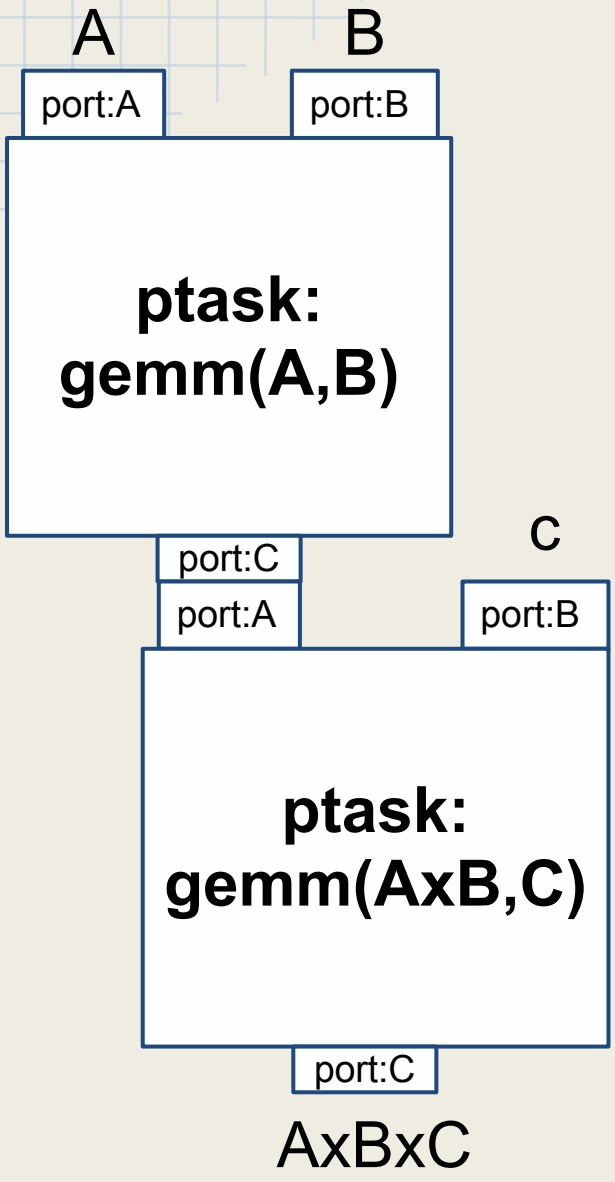
```
matrix AxBxC(A,B,C) {
    matrix AxB = new matrix();
    matrix AxBxC = new matrix();
    copyToDevice(A);
    copyToDevice(B);
    copyToDevice(C);
    invokeGPU(gemm_kernel, A, B, AxB);
    invokeGPU(gemm_kernel, AxB, C, AxBxC);
    copyFromDevice(AxBxC);
    return AxBxC;
}
```

Not modular!

Dataflow Programming

- Allows modularity and efficiency
- Graph structured computation model
  - Units of computation are vertices
  - Vertices have data sources and data sinks (ports)
  - Channels connect ports
- Dataflow managed by the OS

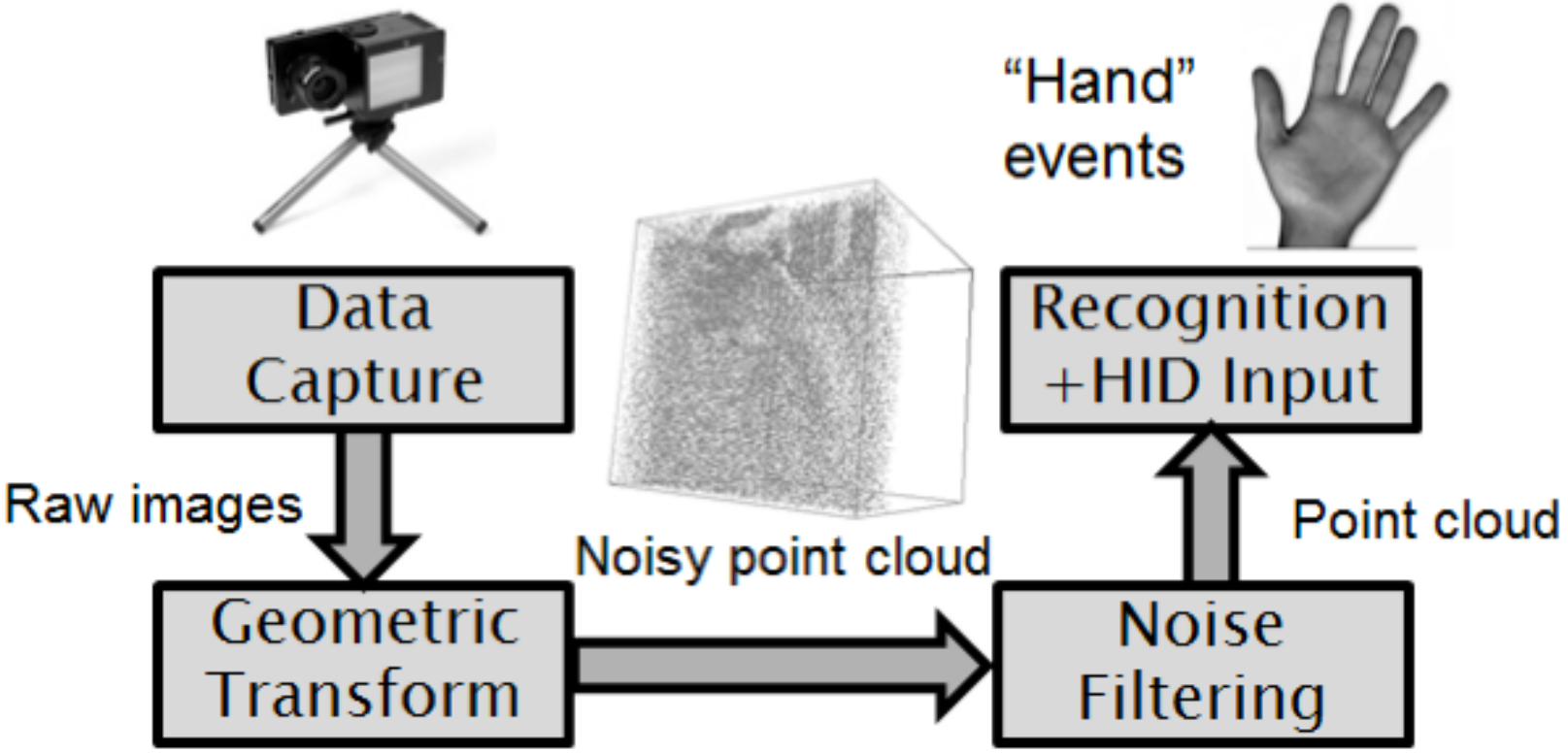

matrix A
Channel
port:A
port:B
Channel
matrix B
ptask:
gemm(A,B)
port:C
Channel
matrix AxB



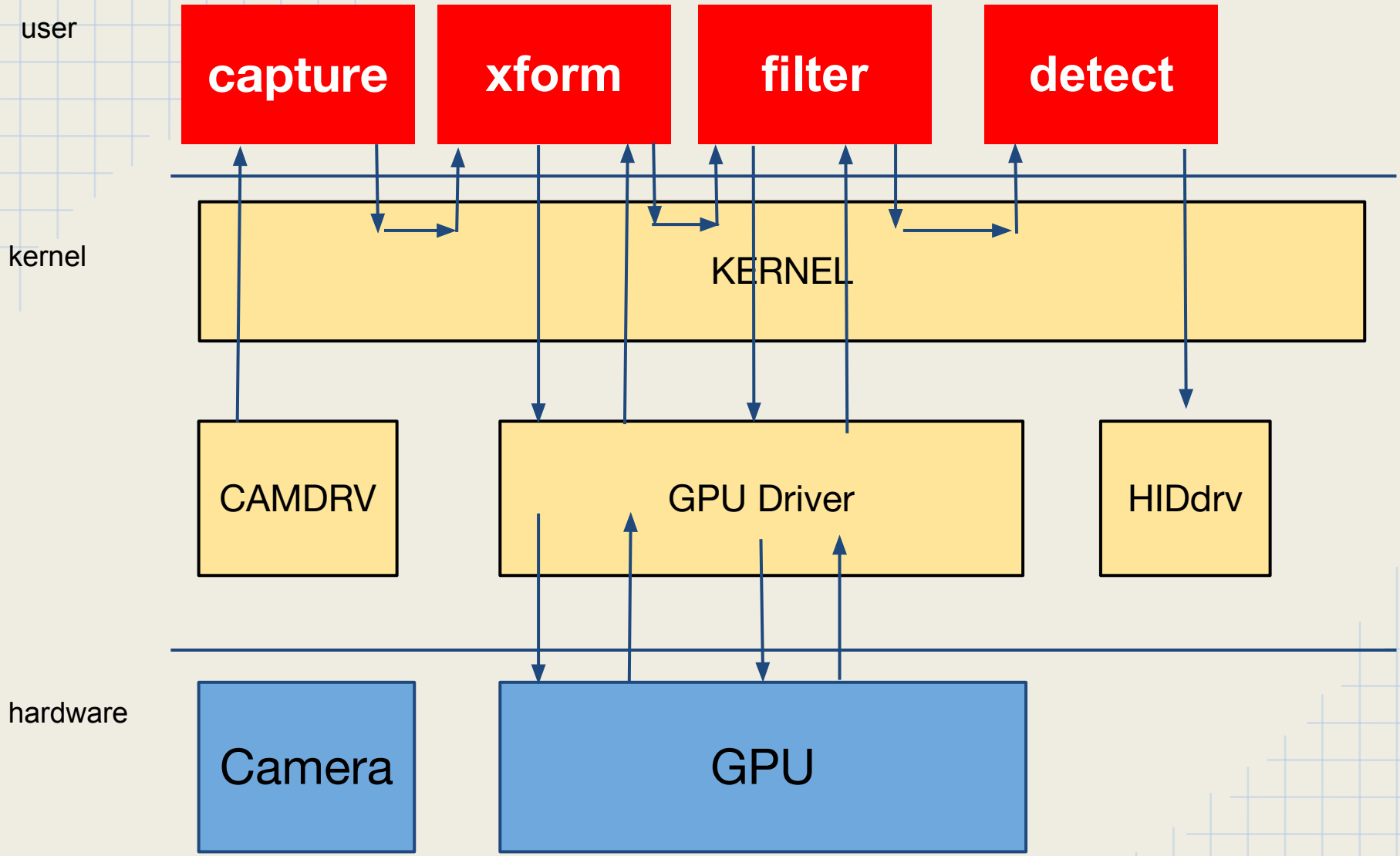
- Dataflow is managed by the OS.
- PTasks (vertices) are computation units
- Ports connected by channels (edges)
- Both modular and efficient

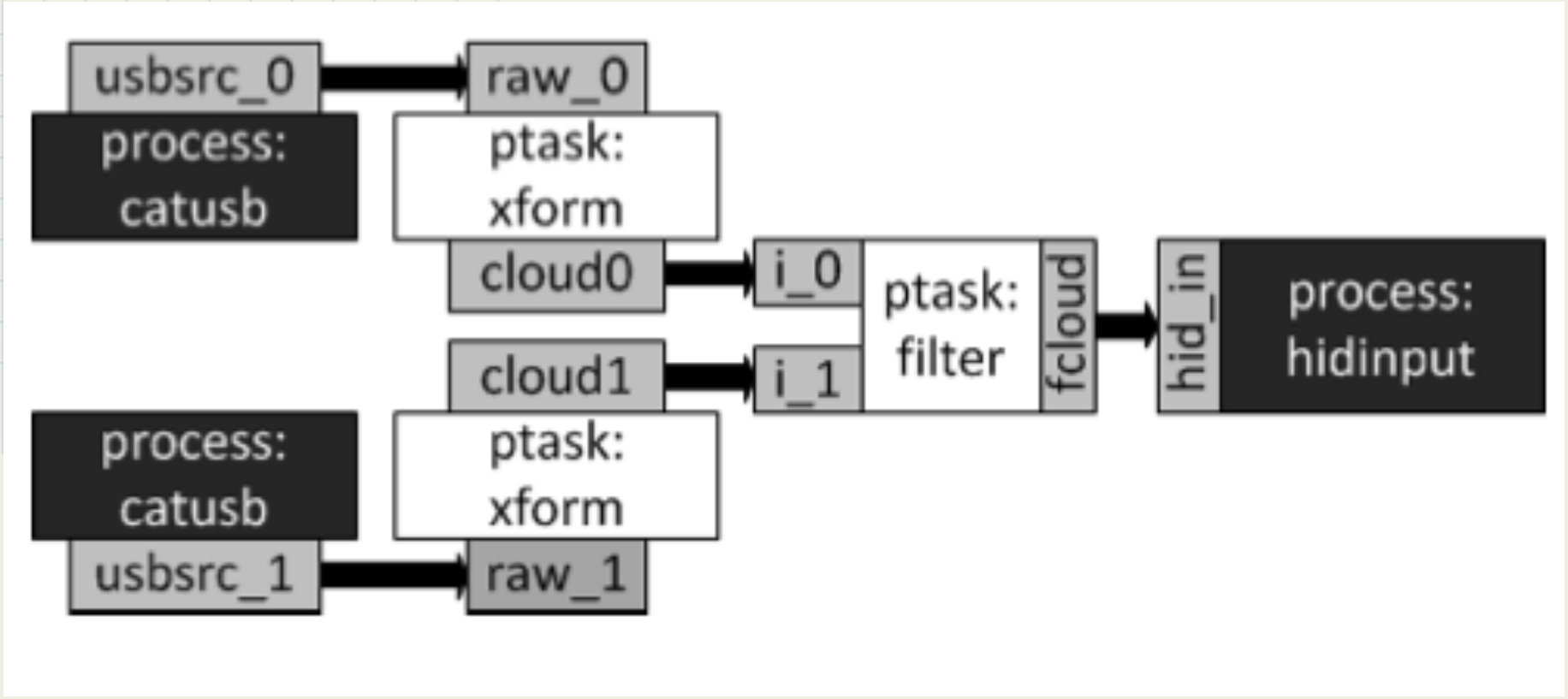
Case Study: Gesture Recognition

- Computationally intensive
  - Ideal for GPGPU acceleration
- Latency requirements
- Largely data parallel
- Multiple user-kernel memory copies
- Follows dataflow paradigm



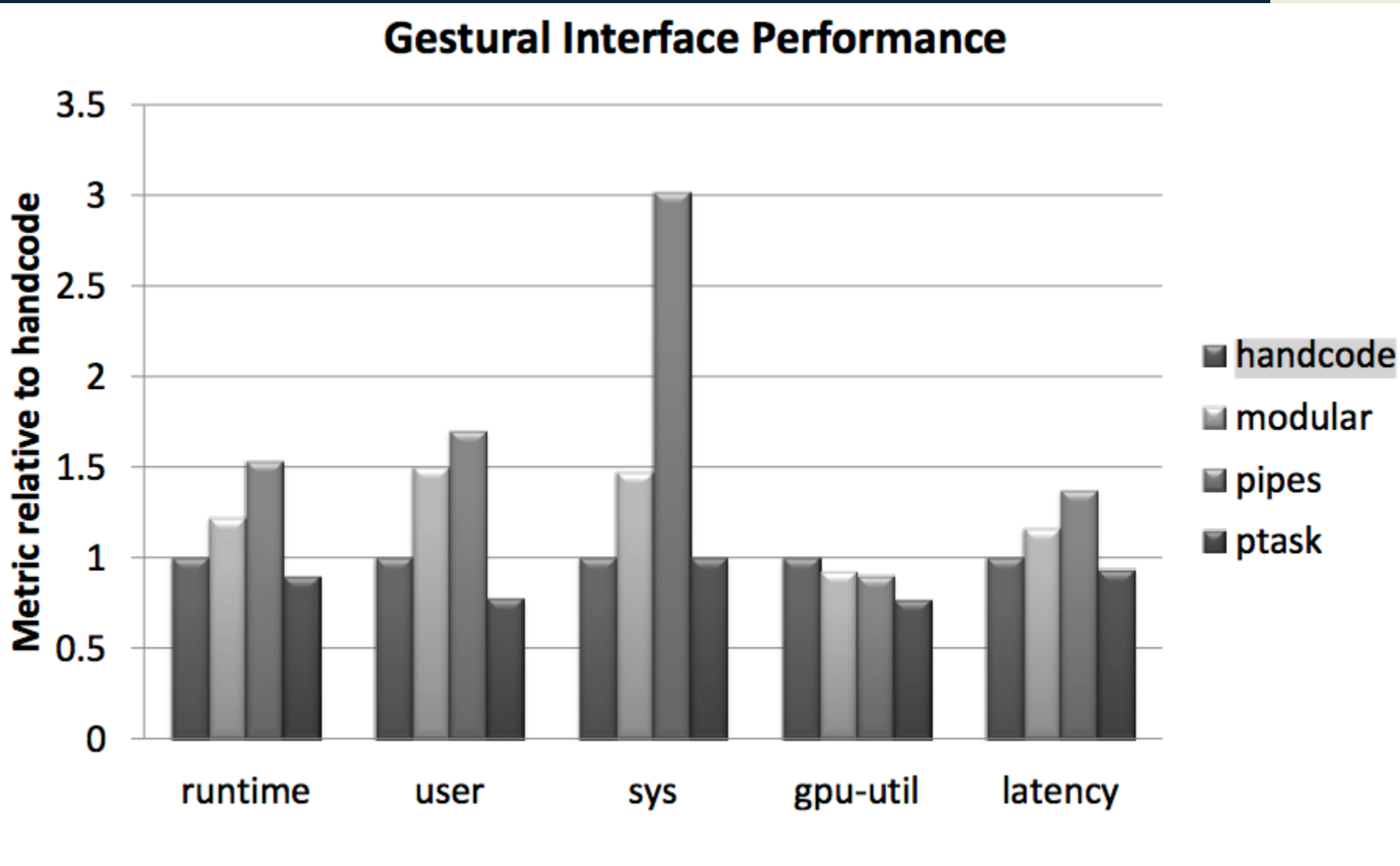
> capture | xform | filter | detect &


user
capture
xform
filter
detect
kernel
KERNEL
CAMDRV
GPU Driver
HIDdrv
hardware
Camera
GPU

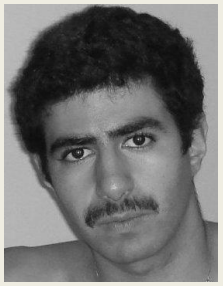

usbsrc_0
raw_0
process:
catusb
ptask:
xform
cloud0
i_0
ptask:
filter
fcloud
hid_in
process:
hidinput
cloud1
i_1
process:
catusb
ptask:
xform
usbsrc_1
raw_1

	impl		fps	tput (MB/s)	lat (ms)	user	sys	gpu	gmem	thrds	ws-delta
Core2-Quad	host-based	real-time	20.2	35.8	36.6	78.1	3.9	–	–	1	–
GTX580	handcode	real-time	30	53.8	10.5	4.0	5.3	21.0	138	1	–
		unconstrained	138	248.2	–	2.4	6.4	41.8	138	1	–
	modular	real-time	30	53.8	12.2	6.0	8.1	19.4	72	1	0.8 (1%)
		unconstrained	113	202.3	–	5.7	8.6	55.7	72	1	0.9 (1%)
	pipes	real-time	30	53.8	14.4	6.8	16.6	18.9	72	3	45.3 (58%)
		unconstrained	90	161.9	–	12.4	24.6	55.4	76	3	46.3 (59%)
	ptask	real-time	30	53.8	9.8	3.1	5.5	16.1	71	7	0.7 (1%)
		unconstrained	154	275.3	–	4.9	8.8	65.7	79	7	1.4 (2%)

- Handcode
  - remove unnecessary data copying
- Modular
  - similar to Handcode but condensed into one process
- Pipes
  - > capture | xform | filter | detect &
- PTasks
  - Uses the PTask API


Gestural Interface Performance
Metric relative to handcode
3.5
3
2.5
2
1.5
1
0.5
0
runtime
user
sys
gpu-util
latency
handcode
modular
pipes
ptask

References



- GPU Computing J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. May 2008
- A Closer Look at GPUs K. Fatahalian, M. Houston. October 2008.
- Managing Accelerators: the Operating System Perspective K. Shantonu.
- CUDA C Programming Guide Nvidia Corporation. October 2012.
- CUDA by Example J. Sanders and E. Kandrot. April 2011.
- CUDA, Supercomputing for the Masses series R. Farber. April 2008 - September 2010.
- NVIDIA Kepler GK110 Architecture. Nvidia Corporation. 2012.
- AMD GCN Architecture. AMD. June 2012.
- PTask: Operating Systems Abstractions To Manage GPUs as Compute Devices. C Rossbach, J. Currey, M. Silberstein, B. Ray, E. Witchel. Microsoft Research. October 2011.

PTask API

sys_open_graph	Create/open graph
sys_open_port	Create/open port
sys_open_ptask	Create/open a ptask
sys_open_channel	Create and bind a channel
sys_open_template	Create/open a template
sys_push	Write to a channel/port
sys_pull	Read from a channel/port
sys_run_graph	Run a graph
sys_terminate_graph	Terminate graph
sys_set_ptask_prio	Set ptask priority
sys_set_geometry	Set iteration space

PTask Scheduling

- First-available
- Fifo
- Priority
- Data-aware