Computer Vision on GPUs

Jan-Michael Frahm, University of North Carolina *at Chapel Hill, USA*
P. J. Narayanan, IIIT, Hyderabad, India
Joe Stam, Nvidia Corporation, USA
Justin Hensley, AMD, USA
Oleg Maslov, Intel, Russia
Nicolas Pinto, MIT, USA

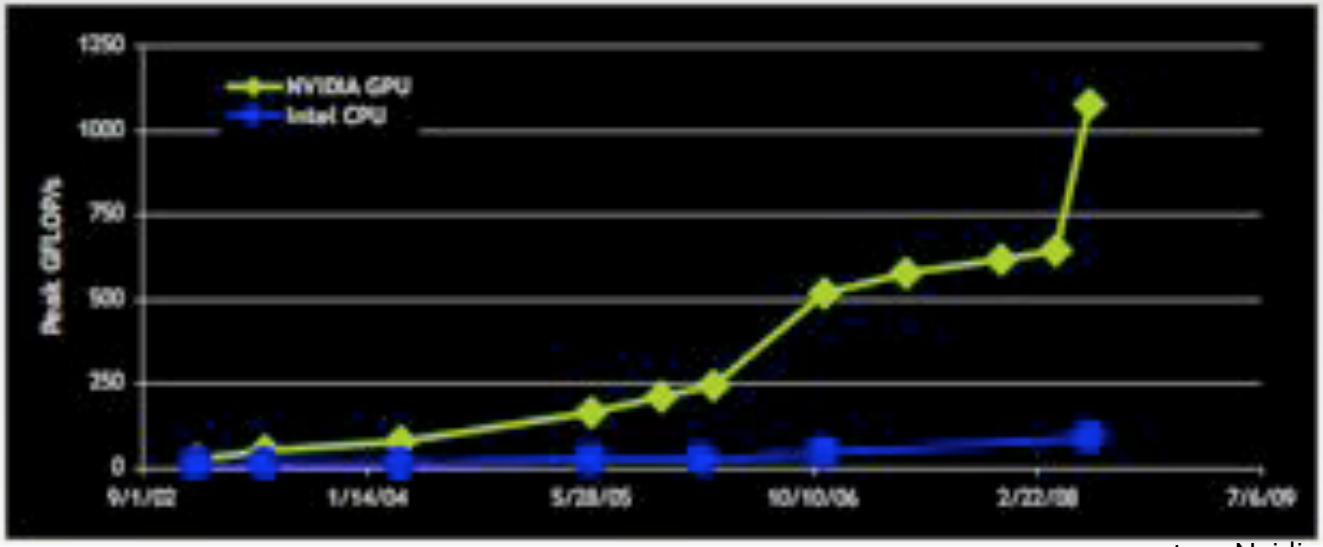
course slides at:
www.cs.unc.edu/~jmf/CVPR2009_CVGPU.pdf

Computer Vision on GPU

- Jan-Michael Frahm, UNC, 3D computer vision group, real-time scene reconstruction
- P. J. Narayanan, IIIT, Hyderabad, vision on GPUs
- Joe Stam, Nvidia, Computer vision and image processing on GPUs
- Justin Hensely, AMD/ATI, GPGPU, face recognition
- Oleg Maslov, Intel, Larrabee group, vision on parallel architectures
- Nicolas Pinto, MIT, DiCarlo lab, accelerate computer vision efforts with an emphasis on bio-inspired models using GPUs, PS3, …

Computation Power

- Single core CPU's are not progressing with Moores law anymore (transistors yes, computation no)
- GPUs are massively parallel and easy to extend to more cores
- Parallel computing is the future to enhance fast



courtesy Nvidia

GPUs and Parallel Computing

- GPUs are highly parallel computing platforms
  - pipeline computation
  - no cache management
- Programming languages
  - CG
  - CUDA
  - OpenCL
- Alternative highly parallel architectures/concepts
  - Larrabee
  - throughput devices

Schedule

09:00 Introduction

09:30 Stream/GPU Computing: Universal concepts:

10:10 Coffee Break

10:30 Universal concepts (Contd)

10:50 CUDA: Overview of Architecture & Progamming

12:00 Lunch break

13:30 OpenCL: Overview of Architecture & Prog

14:20 Larrabee and Manycore architectures

15:00 Coffee Break

15:20 Application Case studies:

16:40 Vision on throughput-computing devices

17:00 Panel discussion

GPU Computing: Basic Concepts

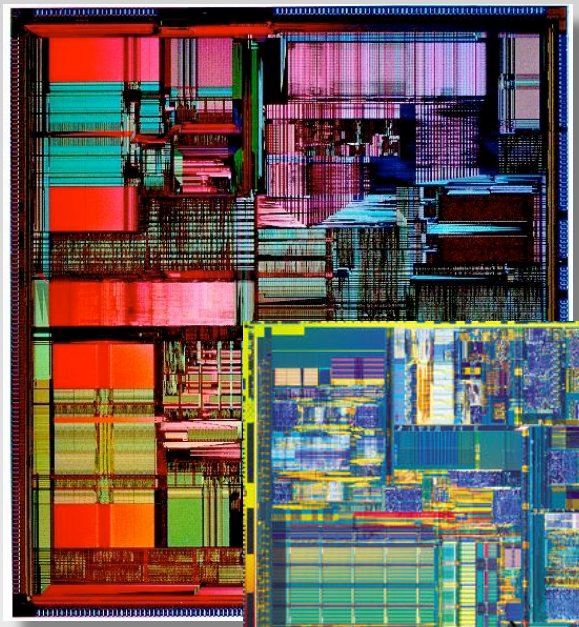
P.J. Narayanan
Centre for Visual Information Technology
IIIT, Hyderabad
India

Motivation

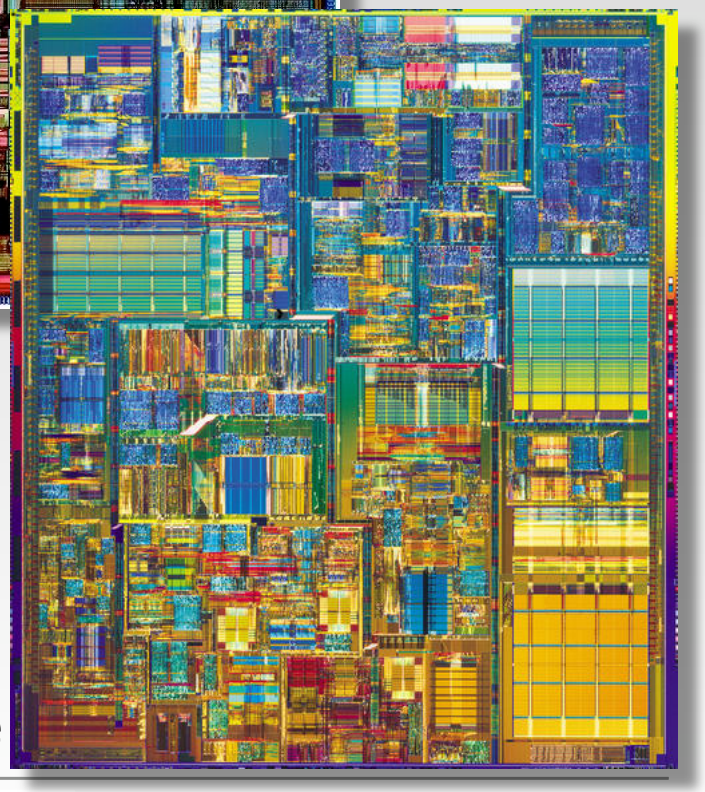
- GPUs are now common. They also have high computing power per dollar, compared to the CPU
- Today's computer system has a CPU and a GPU, with the GPU being used primarily for graphics.
- GPUs are good at some tasks and not so good at others. They are especially good at processing large data such as images
- Let us use the **right processor** for the **right task**.
- Goal: Increase the overall throughput of the computer system on the given task. Use CPU and GPU synergistically.

Processors: Recent Evolution

- Microprocessors and SMP systems dominated for the past 20 or so years
- Moore's law, large volume mean increased everything
  - Pentium of 1993: 3 million transistors, 66 MHz clock
  - Pentium 4 of 2005: 175M transistors, 3.8 GHz
- Clock speed increase brought real performance increase
- Complex architecture



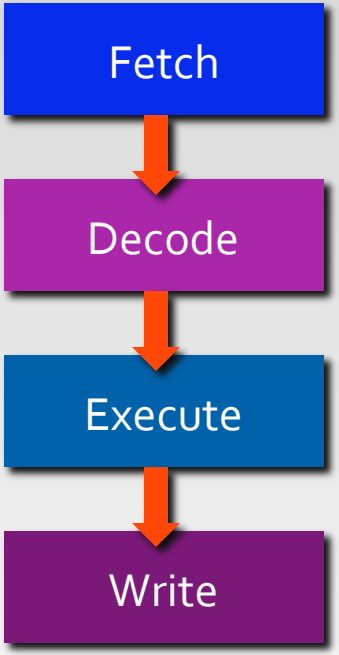
Pentium Die



Pentium 4 Die

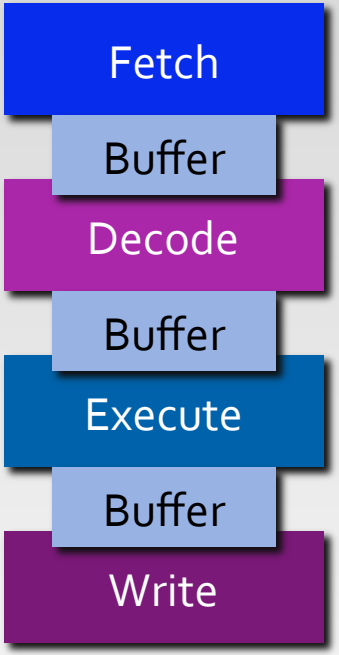
Simple CPU Architecture

- Fetch, Decode, Execute, Store
  - One or more memory operations



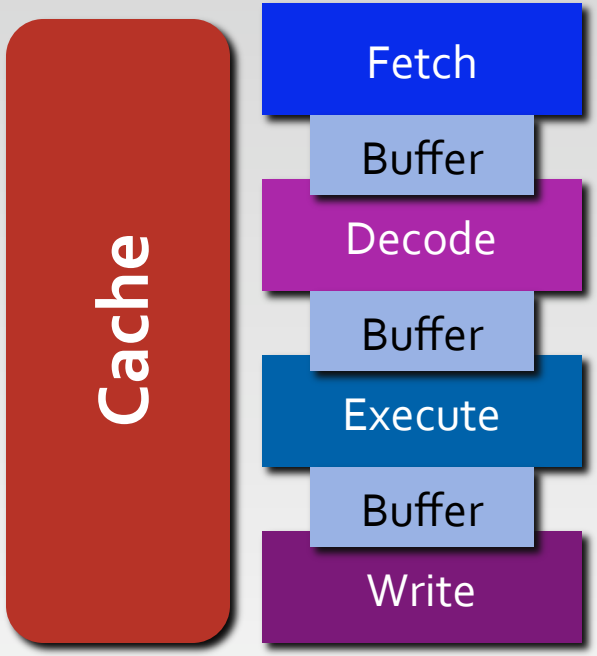
Simple CPU Architecture

- Fetch, Decode, Execute, Store
  - One or more memory operations
- Pipelining: All units operate simultaneously
  - Need buffers to store results
  - Slow memory operations



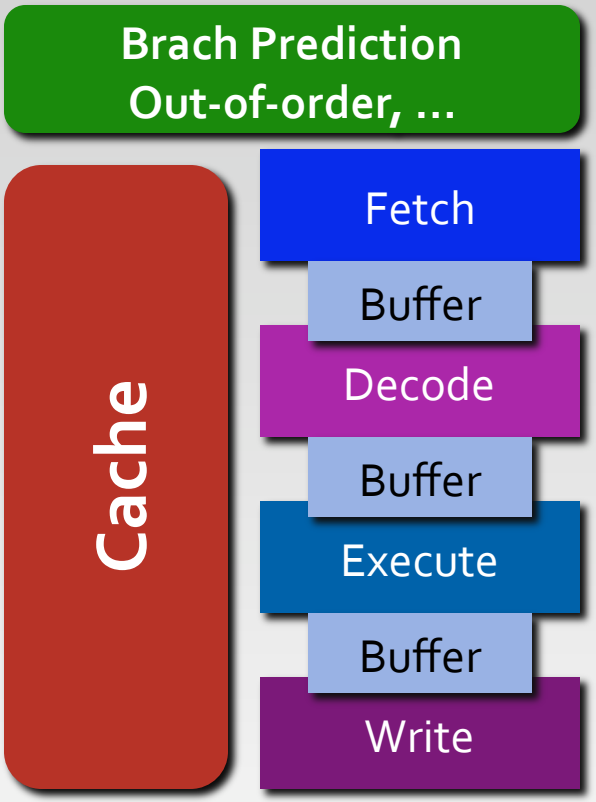
Simple CPU Architecture

- Fetch, Decode, Execute, Store
  - One or more memory operations
- Pipelining: All units operate simultaneously
  - Need buffers to store results
  - Slow memory operation
- Introduce cache to improve memory performance
  - Multilevel caches, data and instruction caches



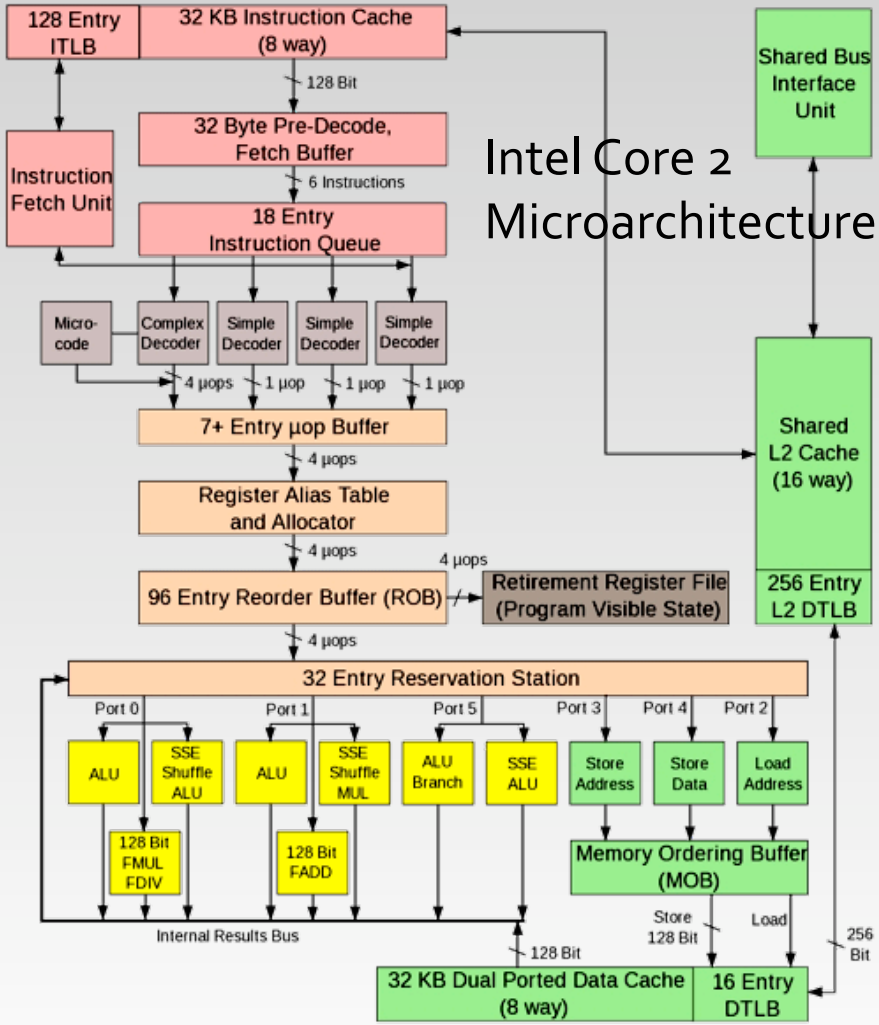
Simple CPU Architecture

- Fetch, Decode, Execute, Store
  - One or more memory operations
- Pipelining: All units operate simultaneously
  - Need buffers to store results
  - Slow memory operation
- Introduce cache to improve memory performance
  - Multilevel caches, data and instruction caches
- Branch Prediction, Out-of-order execution, superscalar, etc.



Complex CPU Architecture

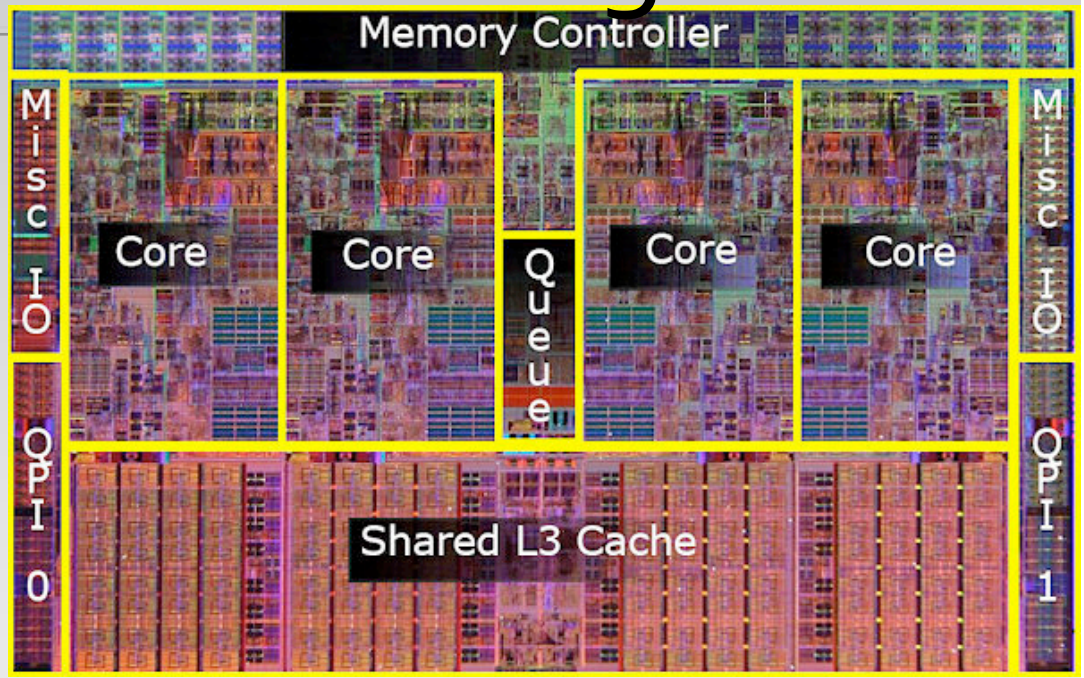
- Clock speeds increased, but memory access latency stays at 60-100 ns
- Processors with 3 level caches, branch prediction, hyperthreading, superscalar execution, etc., later.
- Frequency and power walls were hit early. No free lunch!
- Enter *multicore*, a way to use the increasing transistors



Intel Core 2 Architecture

Performance Scaling

- Multiple cores don't improve performance automatically. Most applications use only one core.
- Model: Multiple processors sharing a common bus and memory space. Common and separate caches



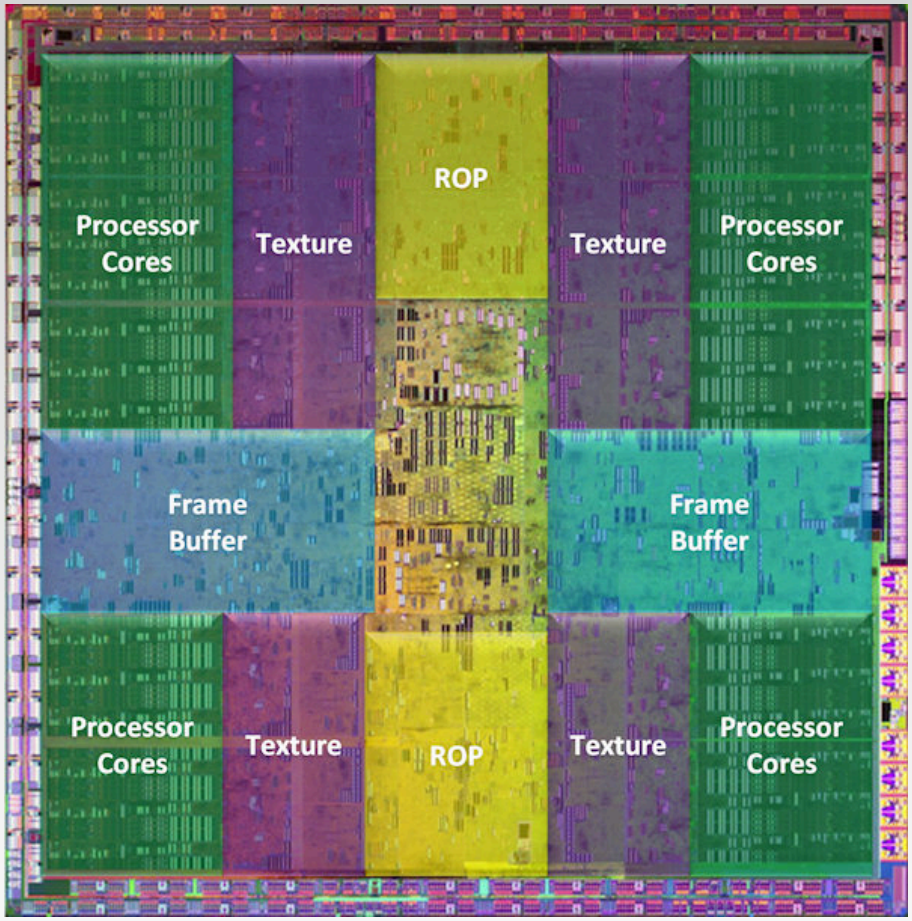
- Compilers can't automatically exploit multiple cores. Special tools required. OpenMPI, Intel tuners, etc.
- Libraries are the way to go: Intel MKL, IPP, etc., are optimized for automatic and transparent performance scaling

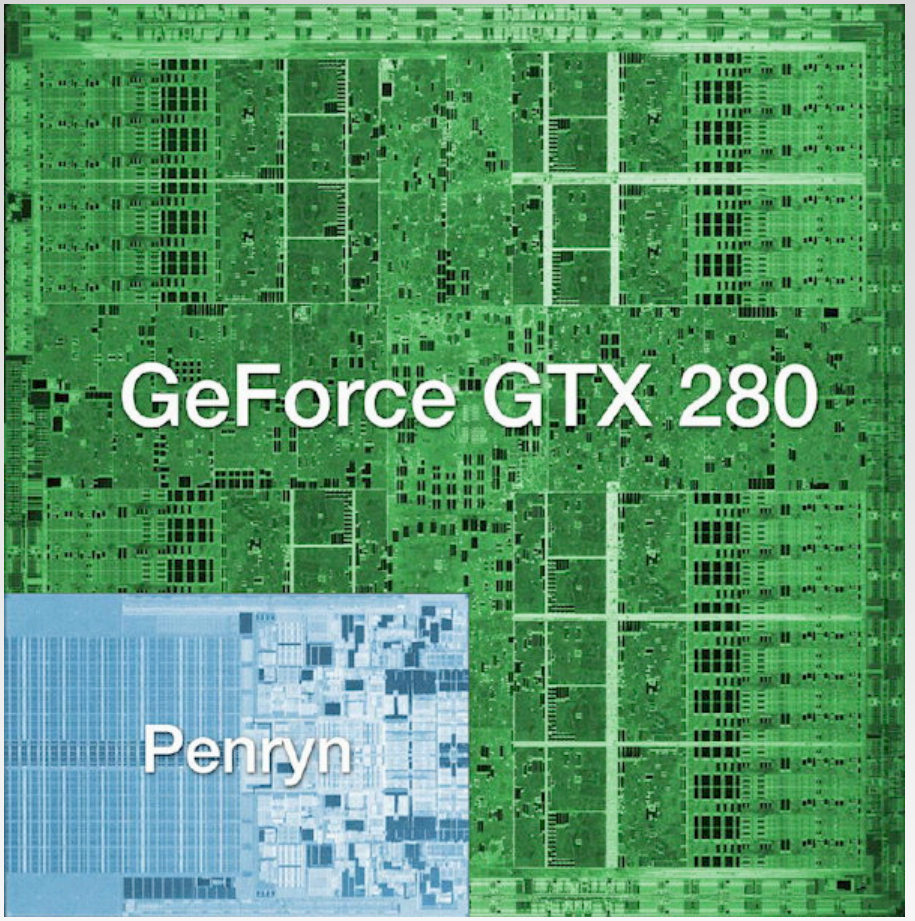
GPU: Evolution

- Graphics : a few hundred triangles/vertices map to a few hundred thousand pixels
- Process pixels in parallel. Do the same thing on a large number of different items.
- Data parallel model: parallelism provided by the data
  - Thousands to millions of data elements
  - Same program/instruction on all of them
- Hardware: 8-16 cores to process vertices and 64-128 to process pixels by 2005
  - Less versatile than CPU cores
  - SIMD mode of computations. Less hardware for instruction issue
  - No caching, branch prediction, out-of-order execution, etc.
  - Can pack more cores in same silicon die area

GPU & CPU

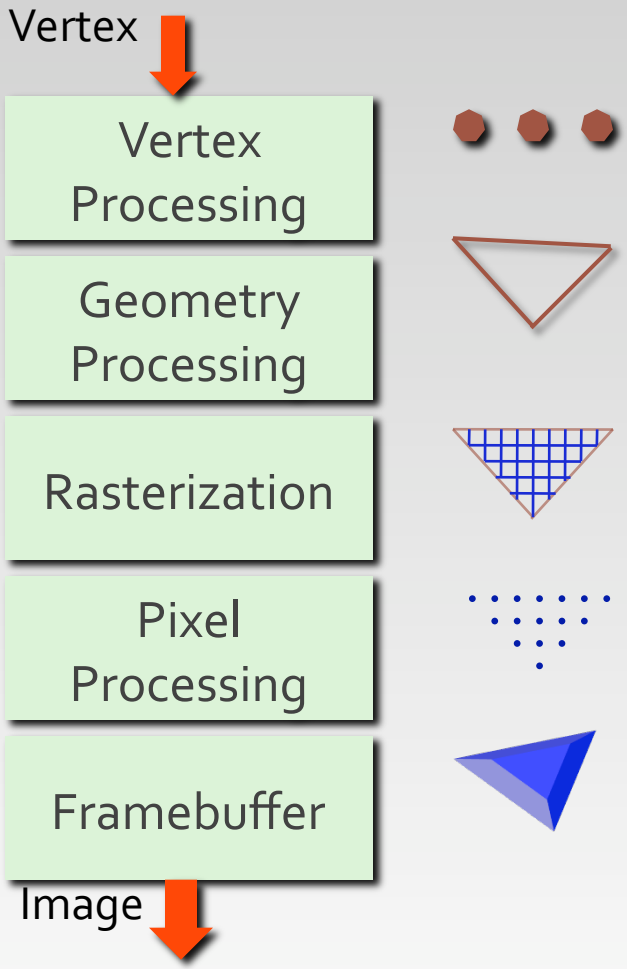
Nvidia GTX280





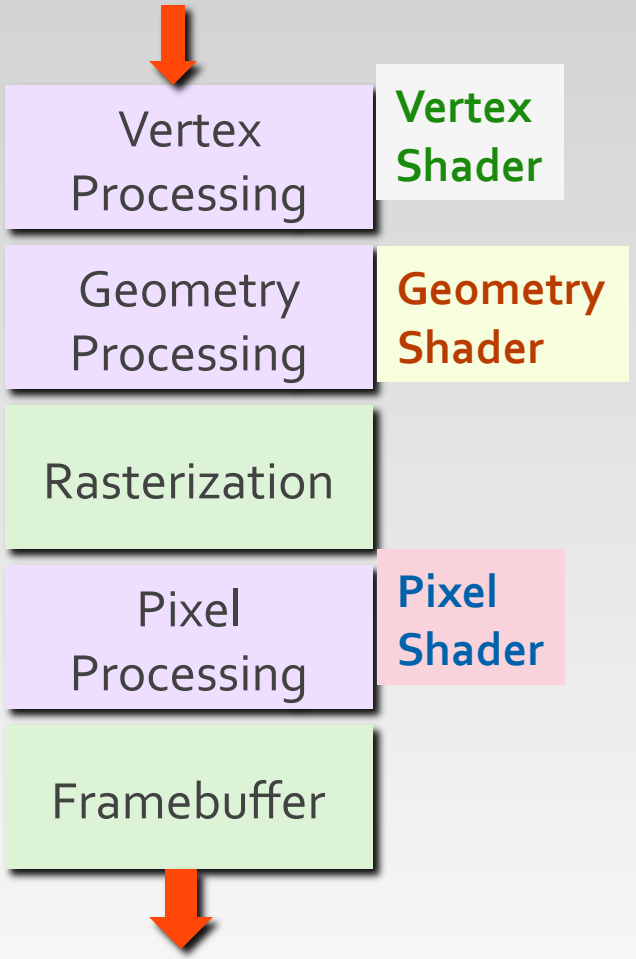
What do GPUs do?

- GPU implements the graphics pipeline consisting of:
  - Vertex transformations
    - Compute camera coords, lighting
  - Geometry processing
    - Primitive-wide properties
  - Rasterizing polygons to pixels
    - Find pixels falling on each polygon
  - Processing the pixels
    - Texture lookup, shading, *Z*-values
  - Writing to the framebuffer
    - Colour, *Z*-value
- Computationally intensive



Programmable GPUs

- Parts of the GPU pipeline were made programmable for innovative shading effects
- ***Vertex***, ***pixel***, & later ***geometry*** stages of processing could run user's *shaders*.
- Pixel shaders perform *Data-parallel* computations on a parallel hardware
  - 64-128 single precision floating point processors
  - Fast texture access
- GPGPU: High performance computing on the GPU using shaders. Efficient for vectors, matrix, FFT, etc.

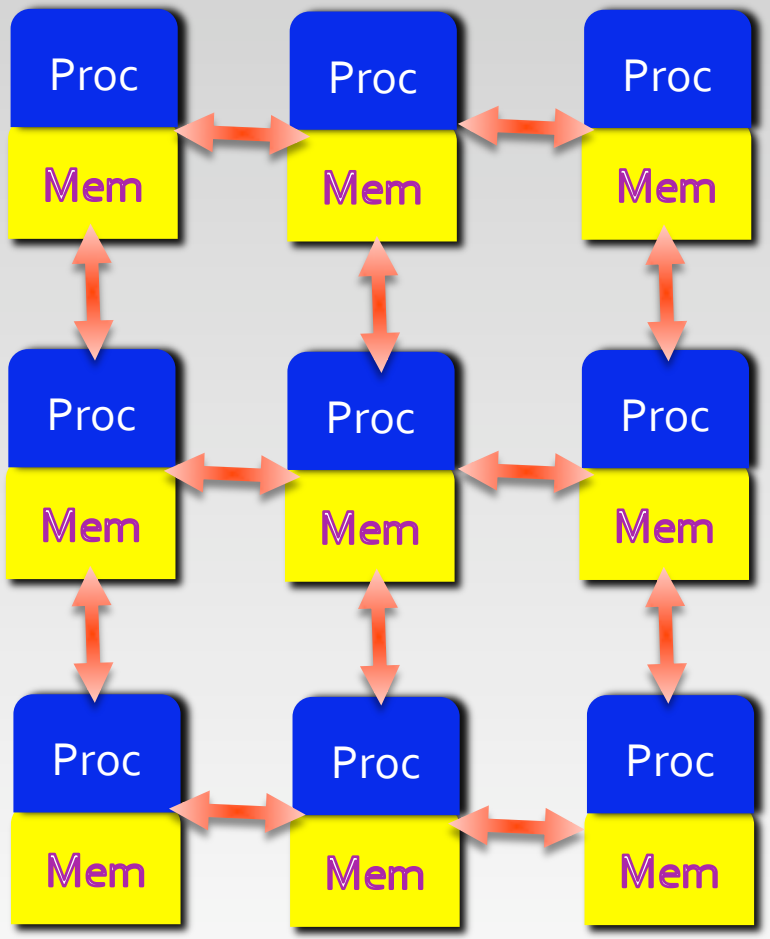


New Generation GPUs

- The DX10/SM4.0 model required a uniform shader model
- Translated into common, unified, hardware cores to perform vertex, geometry, and pixel operations.
- Brought the GPUs closer to a general parallel processor
- A number of cores that can be reconfigured dynamically
  - More cores: 128 → 240 → 320
  - Each transforms data in a common memory for use by others

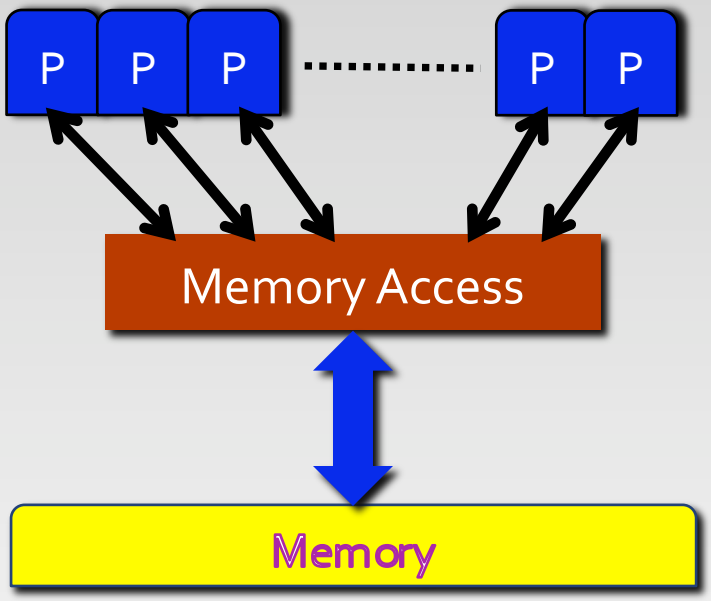
Old Array Processors

- Processor and Memory tightly attached
- A network to interconnect
  - Mesh, star, hypercube
- Local data: Memory read/write
  Remote data: network access
- Data reorganization is expensive to perform
- Data-Parallel model works
- Thinking Machines CM-1, CM-2. MasPar MP-1, etc



Current GPU Architecture

- Processing elements have no local memory
- Bus-based connection to the common, large, memory
- Uniform access to all memory for a PE
  - Slower than computation by a factor of 500
- Resembles the PRAM model!
- No caches. But, instantaneous locality of reference improves performance
  - Simultaneous memory accesses combined to a single transaction
- Memory access pattern determines performance seriously



Compute Power

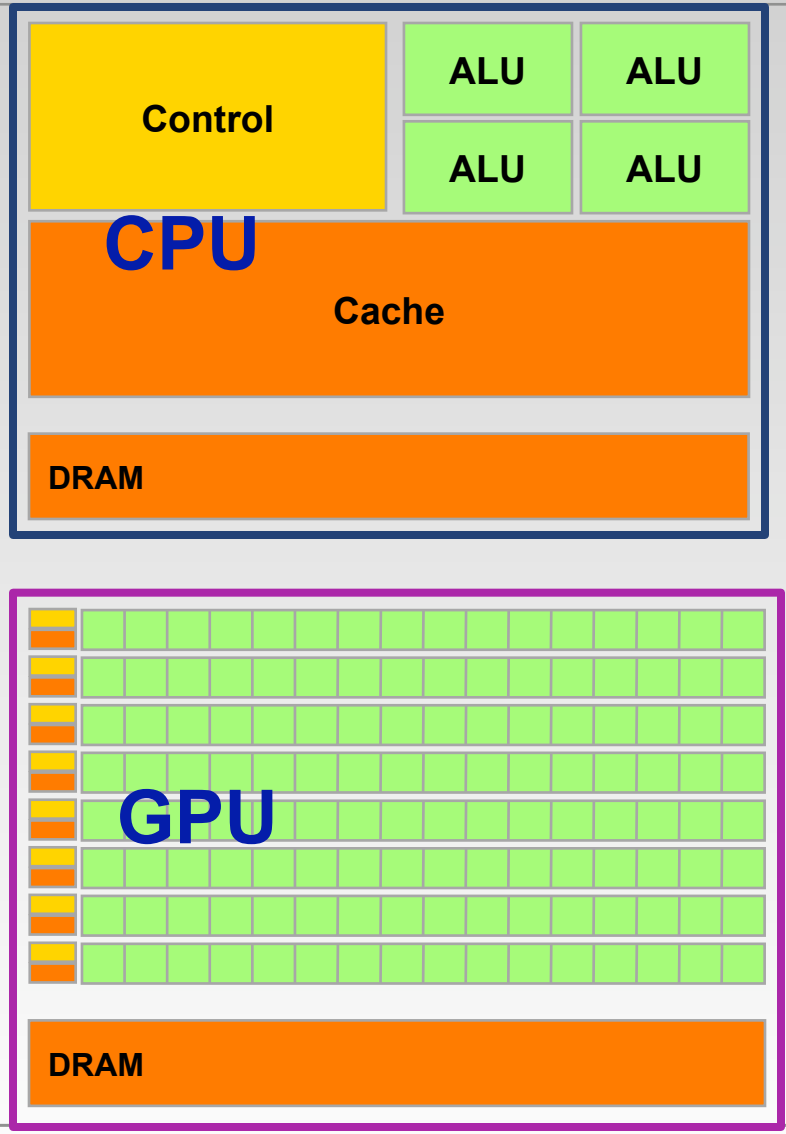
- Top-of-the-line commodity GPUs provide 1 terraflop (TFLOP) of performance for approximately $400
- High Performance Computing or "Desktop Supercomputing" is possible
- However, programming model is not the easiest. They are primarily meant for rendering in Computer Graphics
- Alternate APIs and Tools are needed to bring the power to everyday programmers

Tools and APIs

- OpenGL/Direct3D for older, GPGPU exposure
  - Shaders operating on polygons, textures, and framebuffer
- CUDA: an alternate interface from Nvidia
  - Kernel operating on grids using threads
  - Extensions of the C language
- CAL: A low-level interface from ATI/AMD
  Brook: A stream computing language from Stanford, available on ATI/AMD processors
- DirectX Compute Shader: Microsoft's version
- OpenCL: A promising open compute standard
  - Apple, Nvidia, AMD, Intel, TI, etc.
  - Support for task parallel, data parallel, pipeline-parallel, etc.
  - Exploit the strengths of all available computing resources

CPU vs GPU

- CPU Architecture features:
  - Few, complex cores
  - Perform irregular operations well
    - Run an OS, control multiple IO, pointer manipulation, etc.
- GPU Architecture features:
  - Hundreds of simple cores, operating on a common memory (like the PRAM model)
  - High compute power but high memory latency (1:500)
  - No caching, prefetching, etc
  - High *arithmetic intensity* needed for good performance
    - Graphics rendering, image/signal processing, matrix manipulation, FFT, etc.

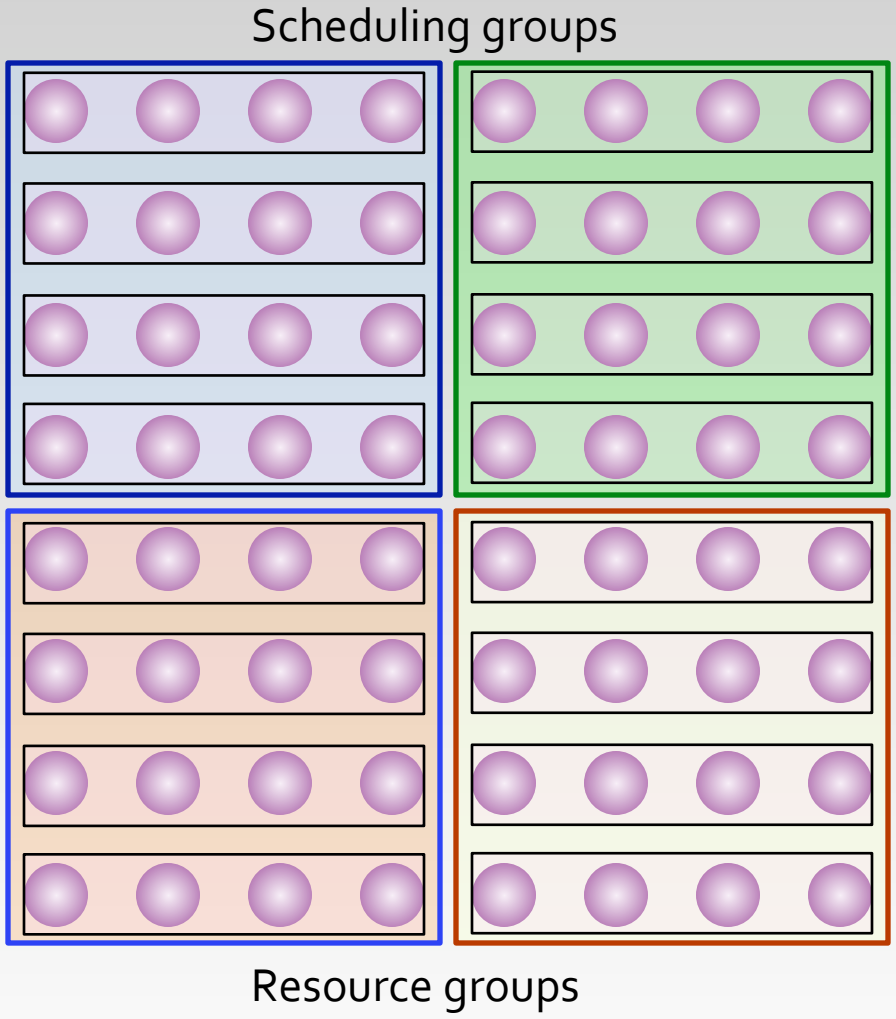


Massively Multithreaded Model

- Hiding memory latency: Overlap computation & memory access
  - Keep multiple threads in flight simultaneously on each core
  - Low-overhead switching. Another thread computes when one is stalled for memory data
  - Alternate resources like registers, context to enable this
- A large number of threads in flight
  - Nvidia GPUs: up to 128 threads on each core on the GTX280
  - 30K time-shared threads on 240 cores
- Common instruction issue units for a number of cores
  - SIMD model at some level to optimize control hardware
  - Inefficient for if-the-else divergence
- Threads organized in multiple tiers

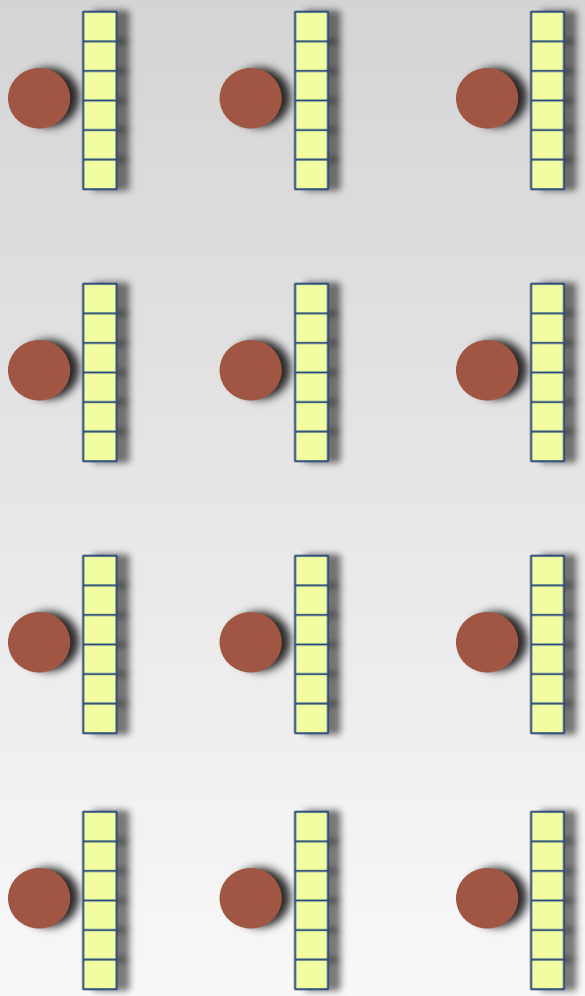
Multi-tier Thread Structure

- Data parallel model: A kernel on each data element
  - A kernel runs on a core
  - CUDA: an invocation of the kernel is called a *thread*
  - OpenCL: the same is called a *work item*
- Group data elements based on simultaneous scheduling
  - Execute truly in parallel, SIMD mode
  - Memory access, instruction divergence, etc., affect performance
  - CUDA: a *warp* of threads
- Group elements for resource usage
  - Share memory and other resources
  - May synchronize within group
  - CUDA: *Blocks* of threads
  - OpenCL: *Work groups*



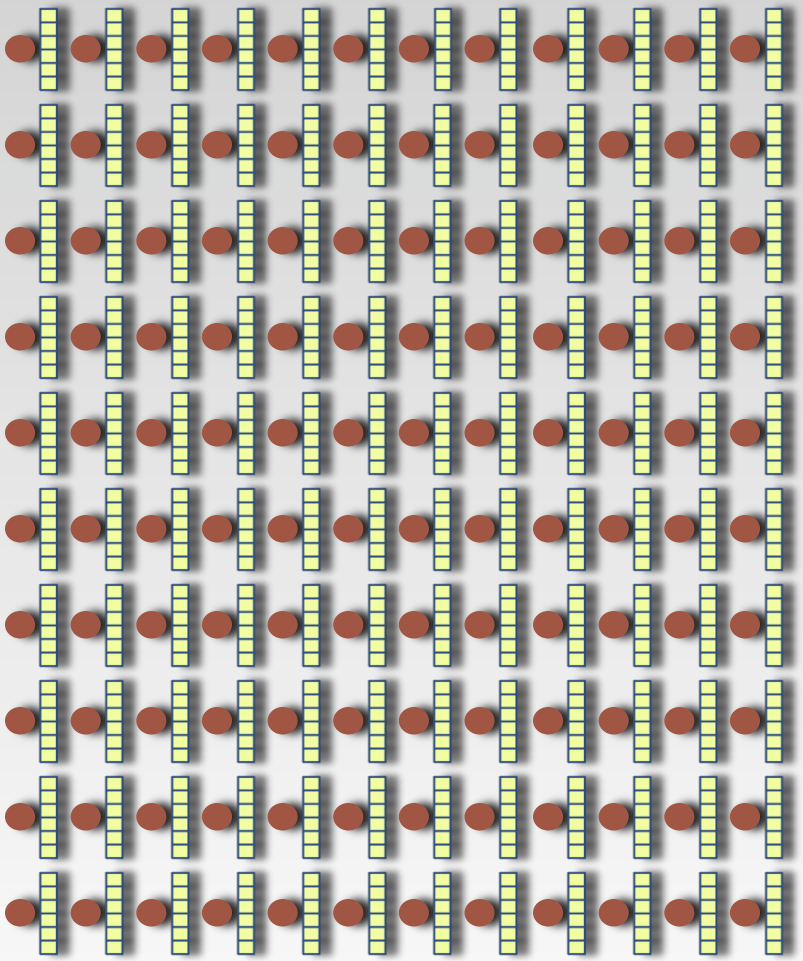
Data-Parallelism

- Data elements provide parallelism
  - Think of many data elements, each being processed simultaneously



Data-Parallelism

- Data elements provide parallelism
  - Think of many data elements, each being processed simultaneously
  - Thousands of threads to process thousands of data elements
- Not necessarily SIMD, most are SIMD or SPMD
  - Each kernel knows its location, identical otherwise
  - Work on different parts using the location

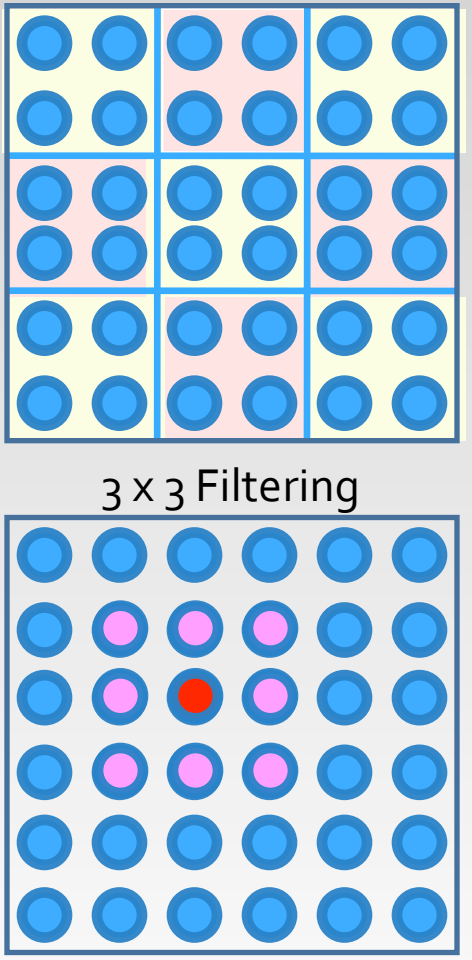


Thinking Data-Parallel

- Launch *N* data ***locations***, each of which gets a ***kernel*** of code
- Data follows a *domain* of computation.
- Each invocation of the kernel is aware of its location *loc* within the domain
  - Can access different data elements using the *loc*
  - May perform different computations also
- Variations of SIMD processing
  - Abstain from a compute step: if (**f(*loc*)**) then … else …
    - Divergence can result in serialization
  - Autonomous addressing for gather: a := b[**f(*loc*)**]
  - Autonomous addressing for scatter: a[**g(*loc*)**] := b
    - GPGPU model supports gather but not scatter
  - Operation autonomy: Beyond SIMD.
    - GPU hardware uses it for graphics, but not exposed to users

Image Processing

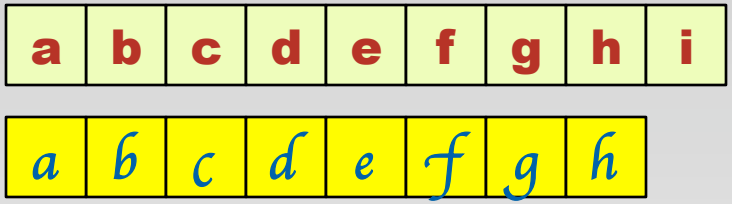
- A kernel for each location of the 2D domain of pixels
  - Embarrassingly parallel for simple operations
- Each work element does its own operations
  - Point operations, filtering, transformations, etc.
- Process own pixels, get neighboring pixels, etc
- Work groups can share data
  - Get own pixels and "apron" pixels that are accessed multiple times

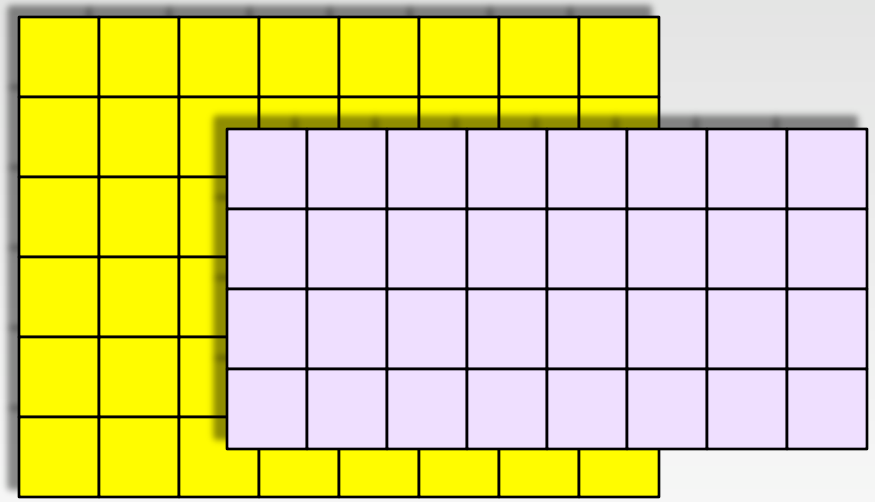


3 x 3 Filtering

Regular Domains

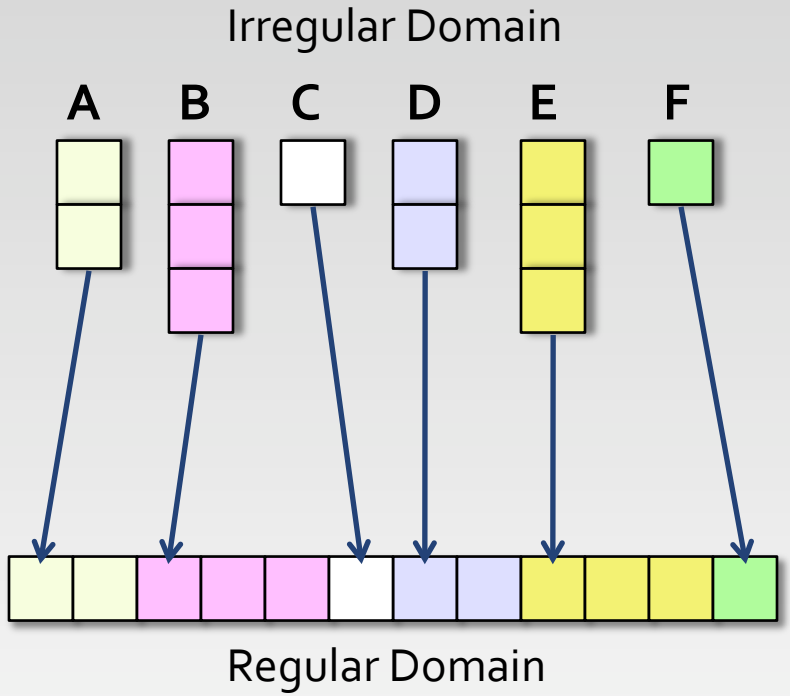
- Regular *1D*, *2D*, and *nD* domains map very well to data-parallelism
- Each work-item operates by itself or with a few neighbors
- Need not be of equal dimensions or length
- A mapping from *loc* to each domain should exist





Irregular Domains

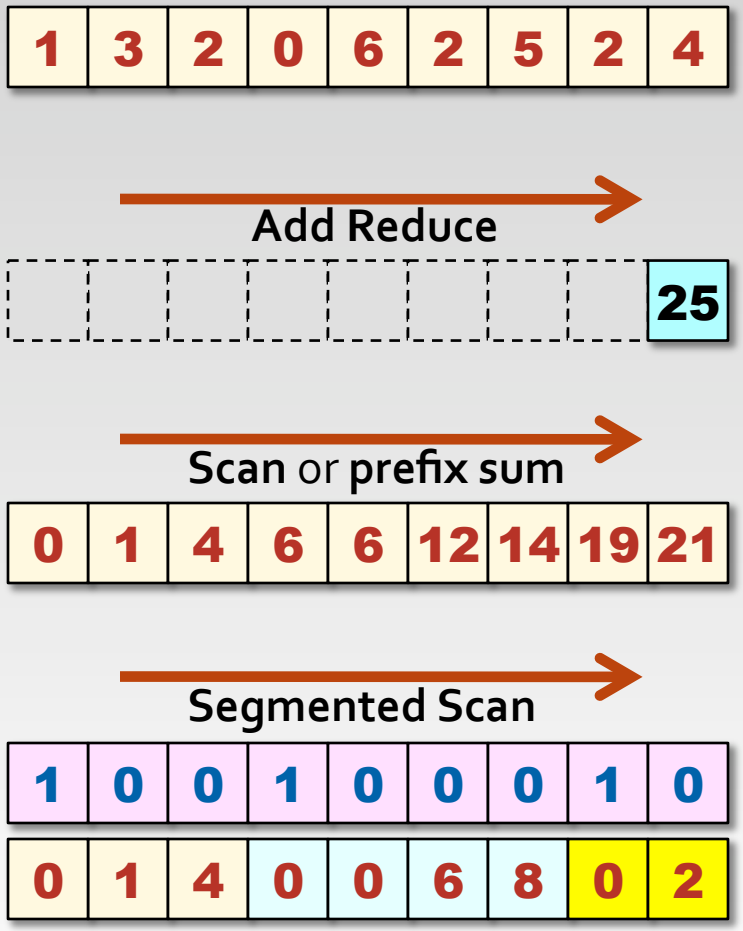
- A regular domain generates varying amounts of data
  - Convert to a regular domain
  - Process using the regular domain
  - Mapping to original domain using new location possible
- Needs computations to do this
- Occurs frequently in data structure building,work distribution, etc.



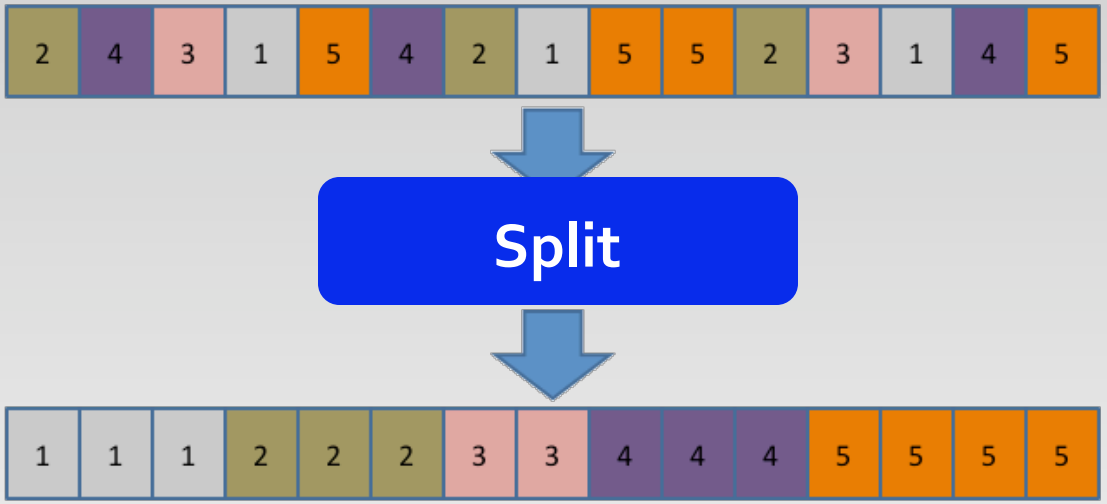
Data-Parallel Primitives

- Deep knowledge of architecture needed to get high performance
  - Use primitives to build other algorithms
  - Efficient implementations on the architecture by experts
- **reduce, scan, segmented scan**: Aggregate or progressive results from distributed data
  - Ordering distributed info
- **split, sort**:
  - Mapping distributed data

[Guy Blelloch (1989)]



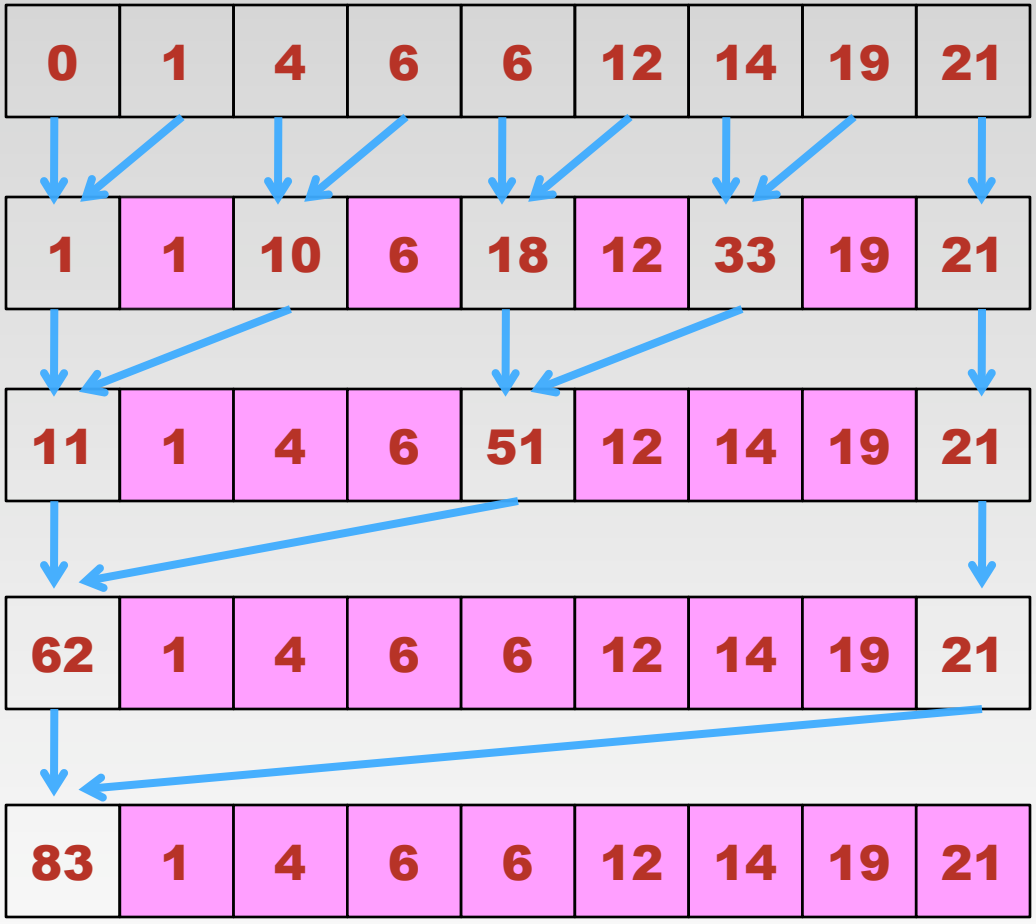
Split Primitive



- Rearrange data according to its category. Categories could be anything.
- Generalization of sort. Categories needn't ordered themselves
- Important in distributing or mapping data

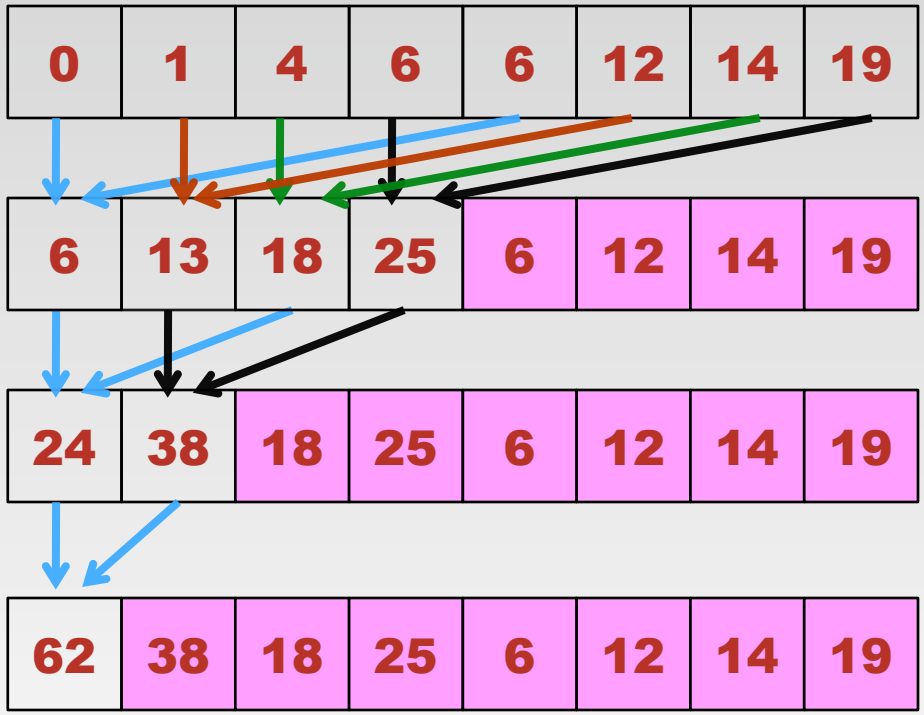
Parallel Reduction: Take 1

- How to implement parallel reduction?
- Multiple threads operating on parts of the data in parallel
  - Use a tree structure?
- Grouping of threads for scheduling may cause divergence and serialization



Parallel Reduction: Take 2

- Nearby data elements are scheduled together in practice.
- Avoiding divergence within them improves performance
- Access elements *N/2* away
- Divergence only at the end
- Small change in thinking for big performance gain



Logical AND/OR

- Each work item has a flag *done*.
- Termination: All are done
- A logical AND of all *done* flags needed
- **Assumption**: One write succeeds when multiple, parallel writes take place to memory
- All work items write the same value!

Start with GDone set to **true**

True

All work items do:
if (! *done*) then
GDone = **false**

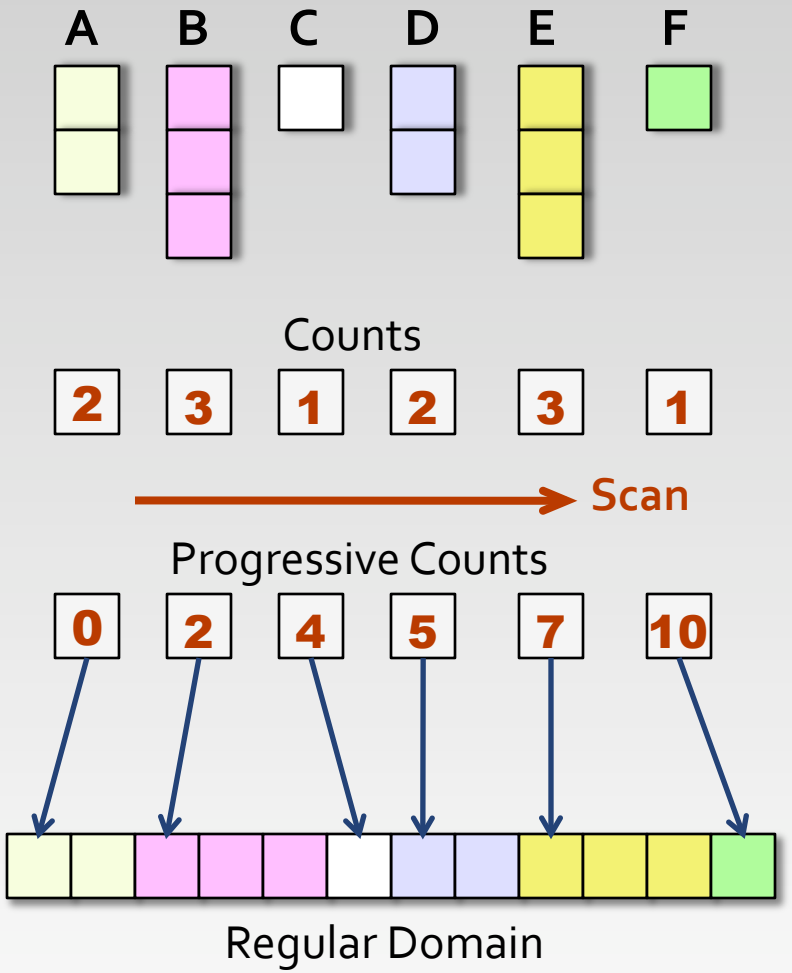
Final result: **true** iff all work items have *done* == **true**

True

False

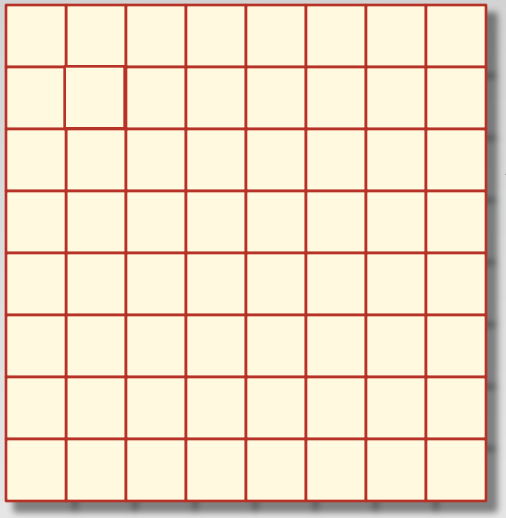
Handling Irregular Domains

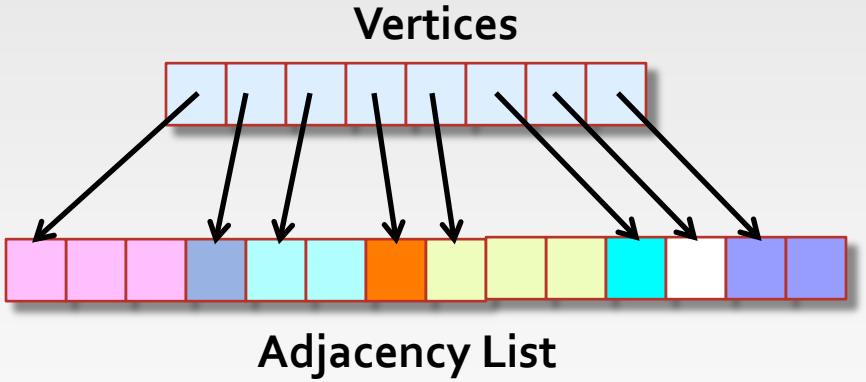
- Convert from irregular to a regular domain
- Each old domain element counts its elements in new domain
- Scan the counts to get the progressive counts or the starting points
- Copy data elements to own location



Graph Algorithms

- Not the prototypical data-parallel application; an irregular application.
- Source of data-parallelism: Data structure (adjacency matrix or adjacency list)
- A 2D-domain of V^2 elements or a 1D-domain of **E** elements
- A thread processes each edge in parallel. Combine the results





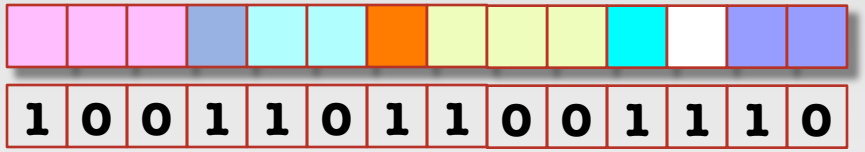
Find min edge for each vertex

- Example: Find the minimum outgoing edge of each vertex

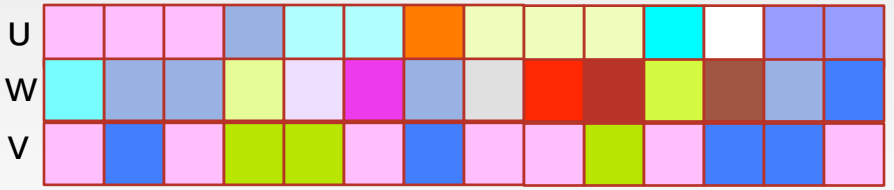
  Soln 1: Each node-kernel loops over its neighbors, keeping track of the minimum weight and the edge

```
for each node in parallel
  for all neighbours v
    if w[v] < min
       min = w[v]
       mv = v
```

Soln 2: Segmented min-scan of the weights array + a kernel to identify min vertex

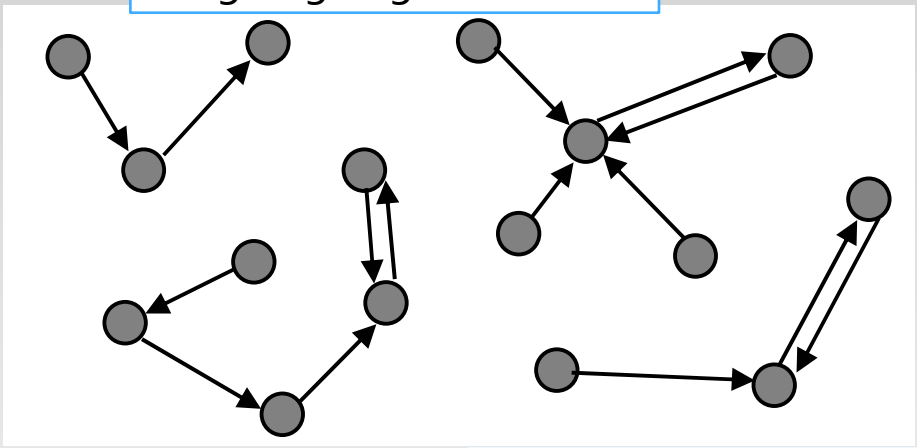


Soln 3: Sort the tuple (*u*, *w*, *v*) using the key (*w*, *v*) for all edges (*u*, *v*) of the graph of weight *w*. Take the first entry for each *u*.

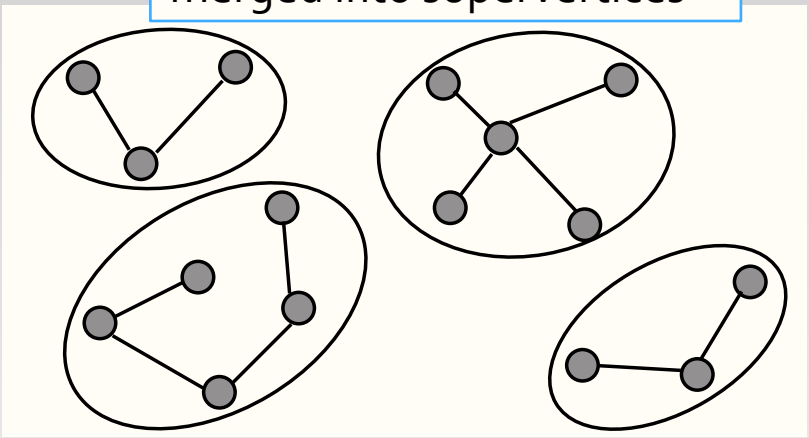


Boruvka's Approach to MST

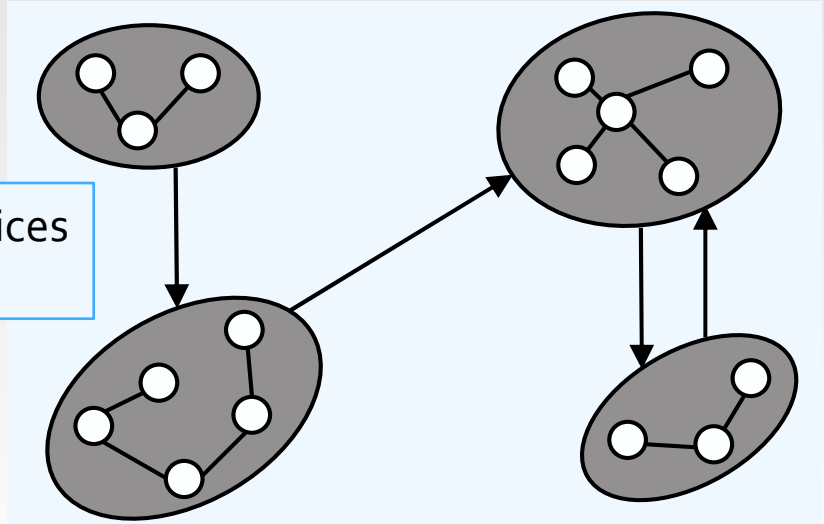
Each node finds the min outgoing edge



Connected components are merged into supervertices



Supervertices are vertices for the next iteration



Repeat these recursively till a single-node graph is formed

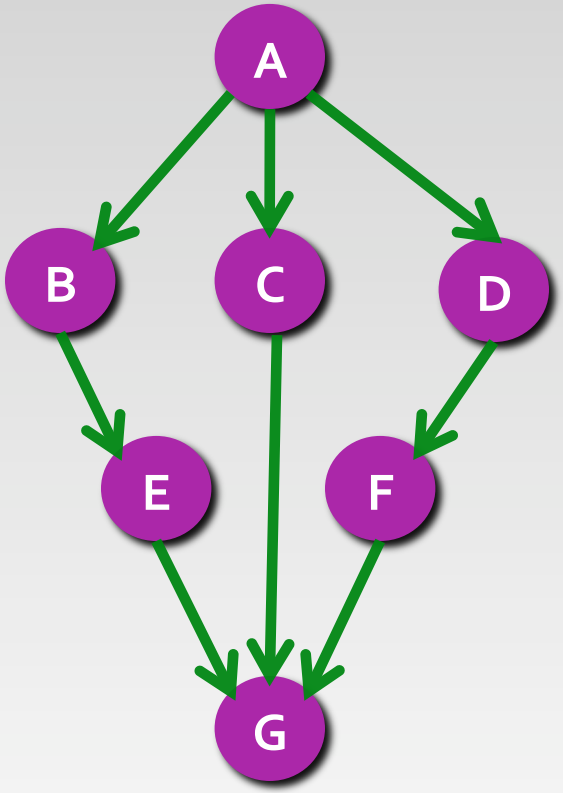
MST on the GPU

- Finding supervertices
  - Pointer doubling to come to the lowest numbered node of a component. This is the representative of its supervertex
  - Split on the supervertex representative with node number to identify supervetex number
  - Segmented scan and compact to identify new node IDs for each supervertex
- Form a new graph for recursive application
  - Remove non-minimal edges between a pair of supervertices by a 64-bit split of (u, v, w) records
  - Compact remaining edges to get a reduced graph of supervertices
- Apply algorithm recursively until a single node is reached
- MST of a 5 million node graph in under a second on a GTX280. 30-50 times speedup over CPU implementation

Vineet, Harish, Patidar, and Narayanan. **Fast Minimum Spanning Tree for Large Graphs on the GPU**. To appear in *High Performance Graphics. 2009*.

Task Parallel Computing

- The problem is divided into a number of tasks; Data may also be partitioned or shared
- Some can be done in parallel, others depend on previous results
- Combine the results finally
- CPU cores and GPU can be doing task-parallel computing
- OpenCL supports this model of computation as well as the pipelined model
- More on OpenCL later today



Summary

- GPU can be an essential computing platform with a massively multithreaded programming model
- Data-parallel model fits the GPUs best.
- High performance requires deep knowledge of the architecture. High-level primitives can alleviate this greatly.
- Think of CPU **and** GPU together achieving your computing goals. Not one instead of the other
- OpenCL is an exciting new development that can make this possible and portable!

For More Information

- GPGPU: gpgpu.org
- SIGGRAPH Courses:
  - SIGGRAPH 2008: Available at UC, Davis. http://so8.idav.ucdavis.edu/
  - SIGGRAPH Asia 2008: Available at UC, Davis http://sao8.idav.ucdavis.edu/
  - Upcoming course at SIGGRAPH 2009
- CudaZone for Nvidia
- And more …

Thank you!

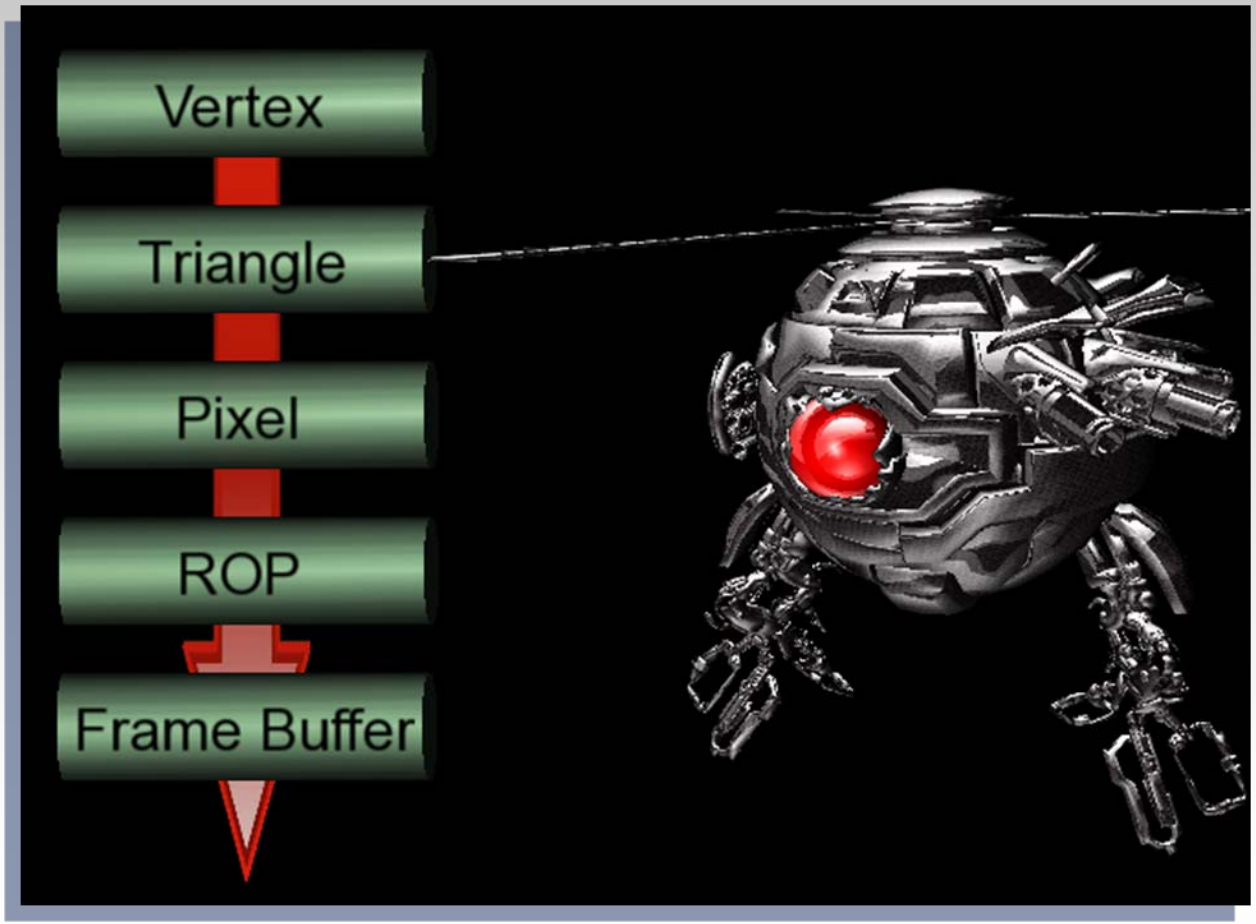
Image credits to owners such as Intel, Nvidia, AMD/ATI, etc.

NVIDIA GPU Architecture and CUDA-C Programming

Joe Stam
Sr. Applications Engineer, NVIDIA

First, What is a GPU?

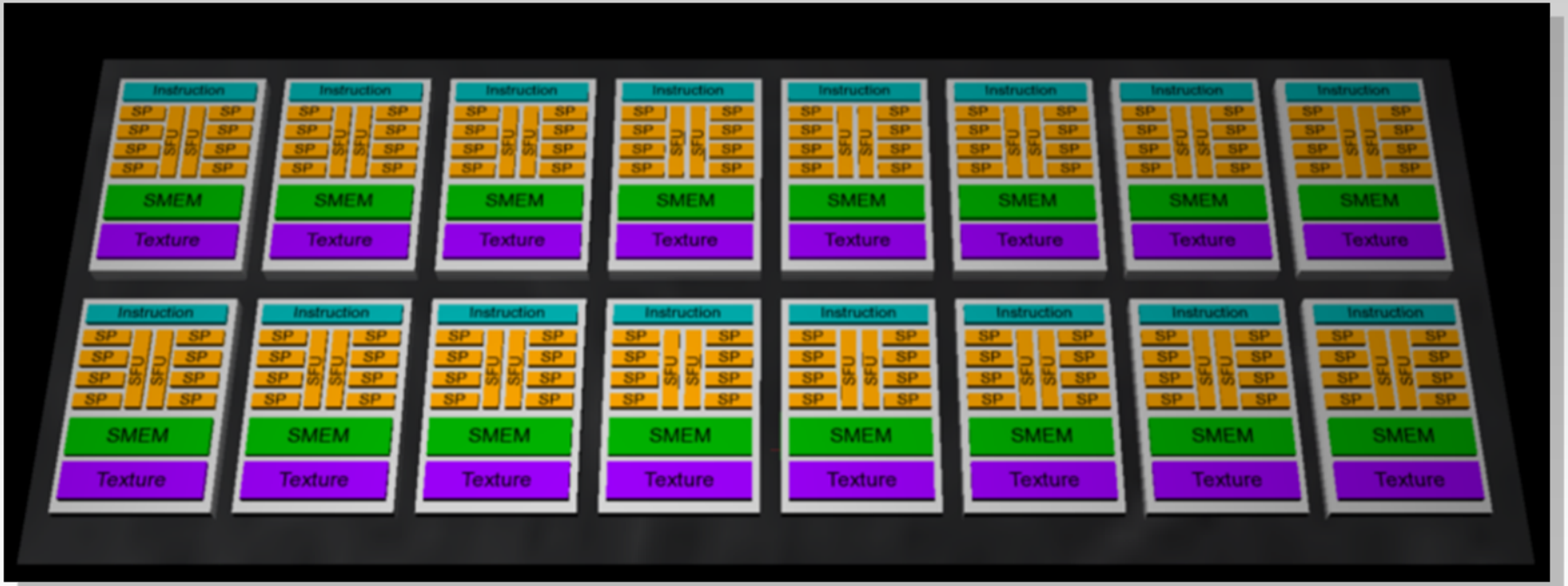
The past…



- Special purpose hardwired processors to accelerate common graphics API functions (i.e. OpenGL, D3D)
- limited applicability to non-graphics tasks
- *However*, raw processing power was so compelling programmers performed coding acrobatics to utilize GPUs for compute intensive applications.

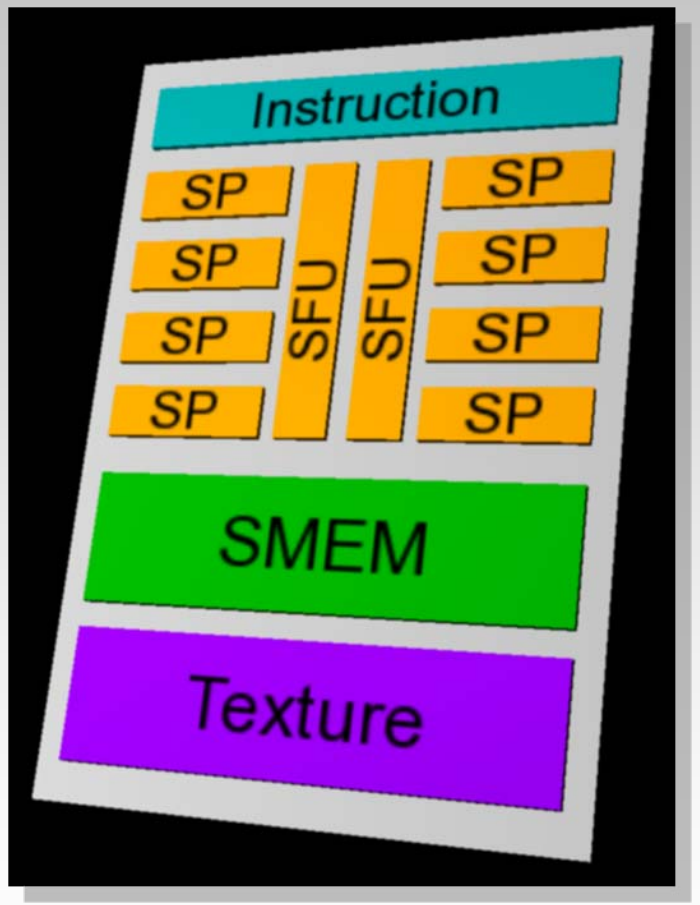
NVIDIA GPUs Today

A massively parallel programmable processor



The Streaming Multiprocessor

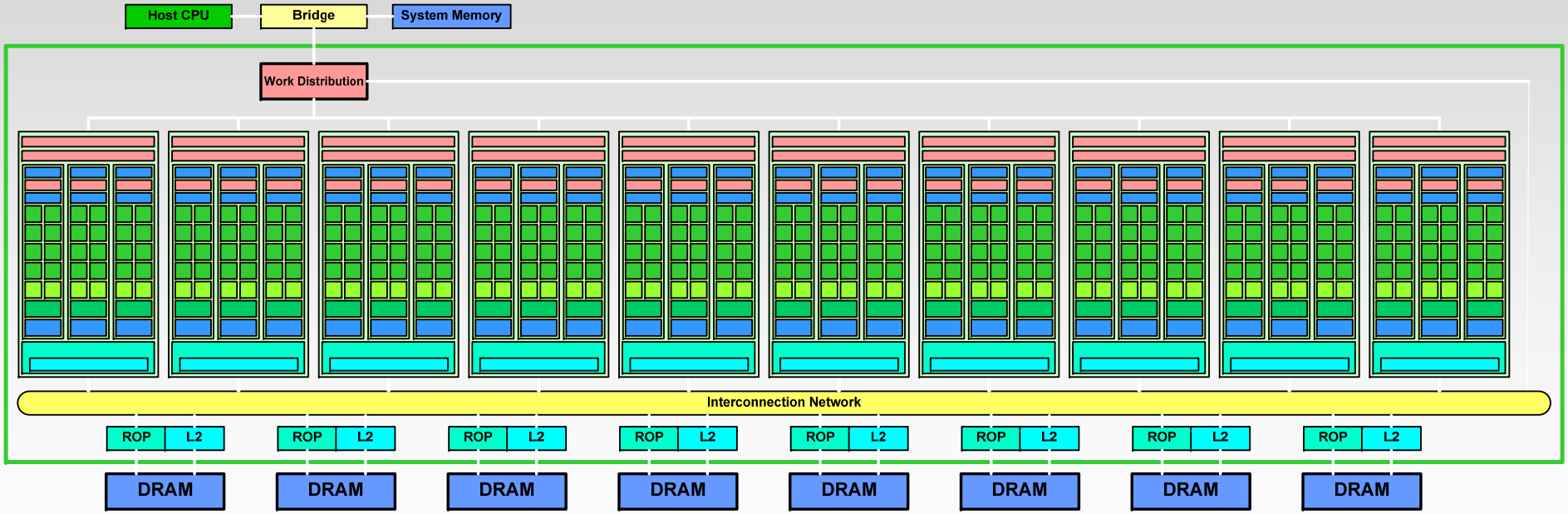
- 8 32-bit floating point Scalar Processor Cores *(DP now available)*
- 2 Special Function units
- 16K Shared Memory
- 8192 or 16384 registers
- Texture Unit



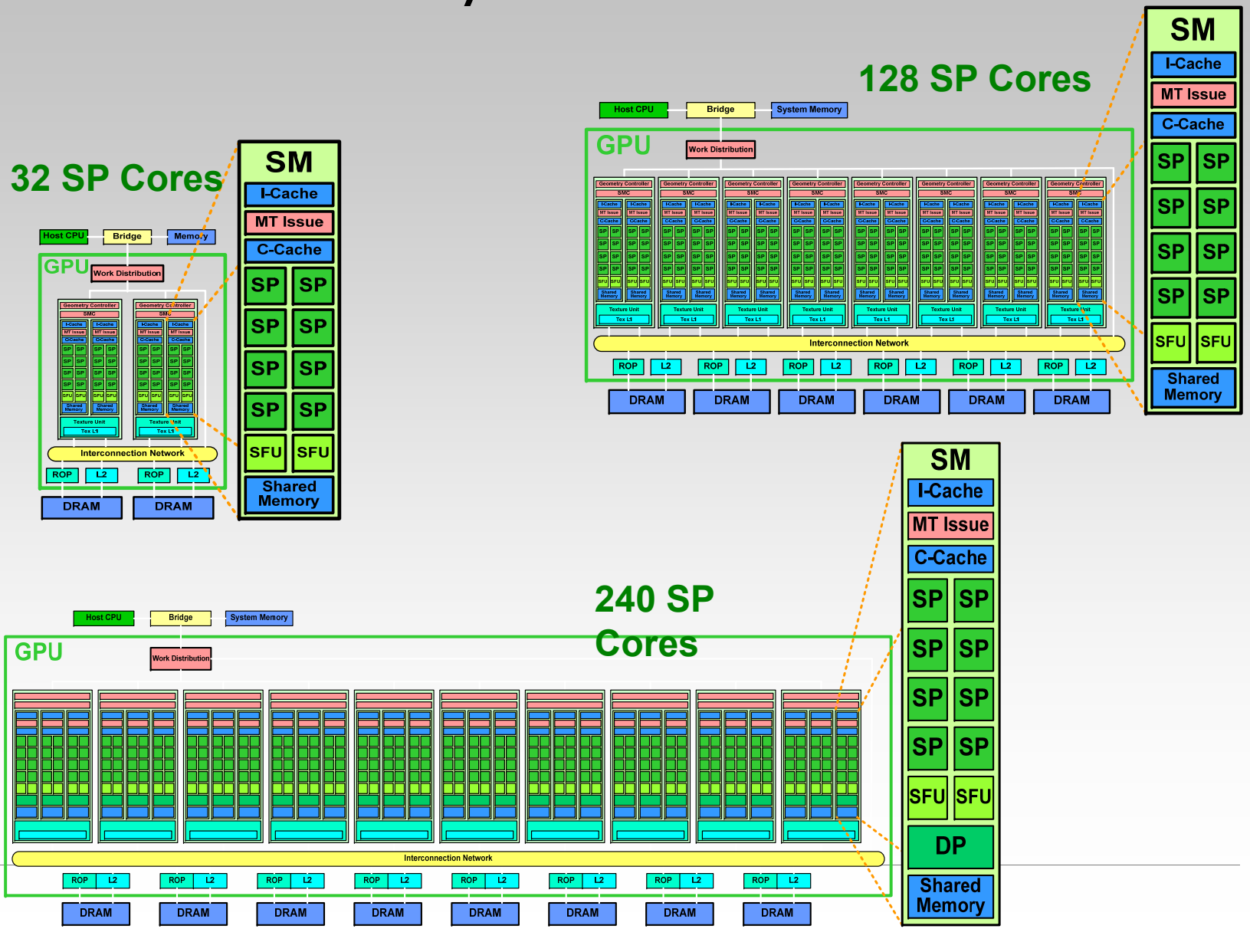
SM's are the building blocks of NVIDIA GPUs

The GT200 Architecture

- Massively parallel general computing architecture
- 30 Streaming multiprocessors @ 1.45 GHz with 4.0 GB of RAM



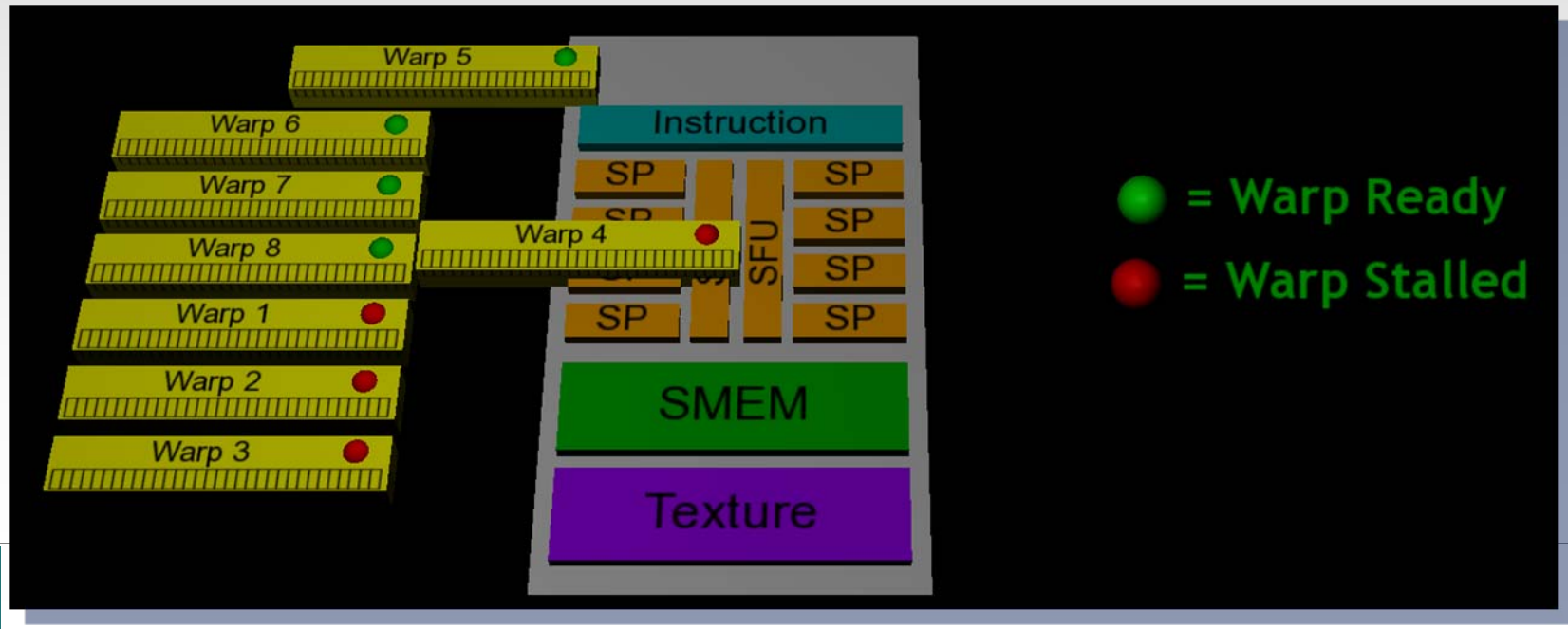
And it comes many sizes



Critical Architecture Features

Latency Hiding

- External Memory is very slow, even on GPUs!
- Instantly switch between threads to hide latency – up to 128 threads / core can be resident



Critical Architecture Features

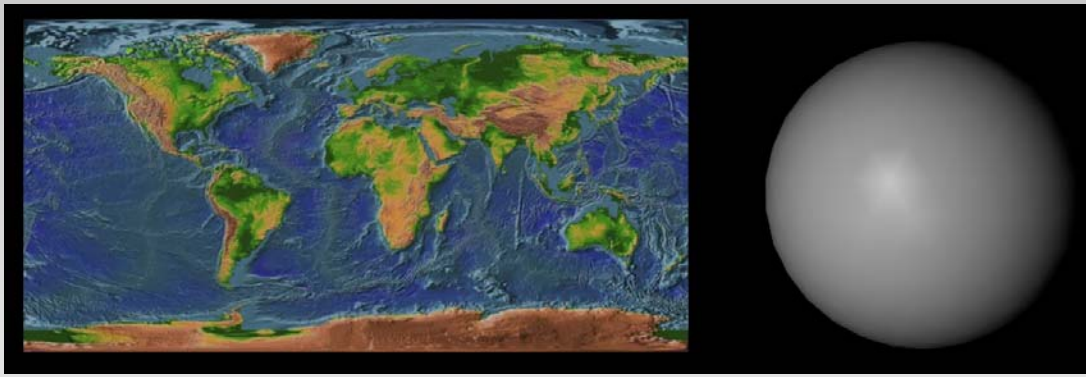
Shared Memory

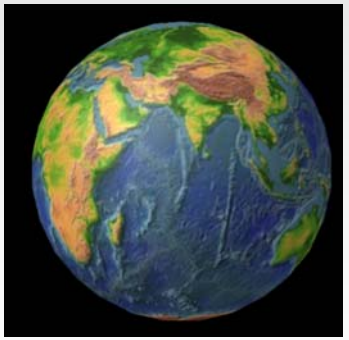
- Acts like a very fast local cache, but programmer controlled
- Allows intermediate computation results to be shared between threads

Critical Architecture Features

Texture Hardware

- Fast, Cached access to image data
- Hardware for:
  - Bilinear Interpolation
  - Type conversion
  - Boundary handling
- Excellent for geometrical distortions





CUDA Hierarchical Thread Structure

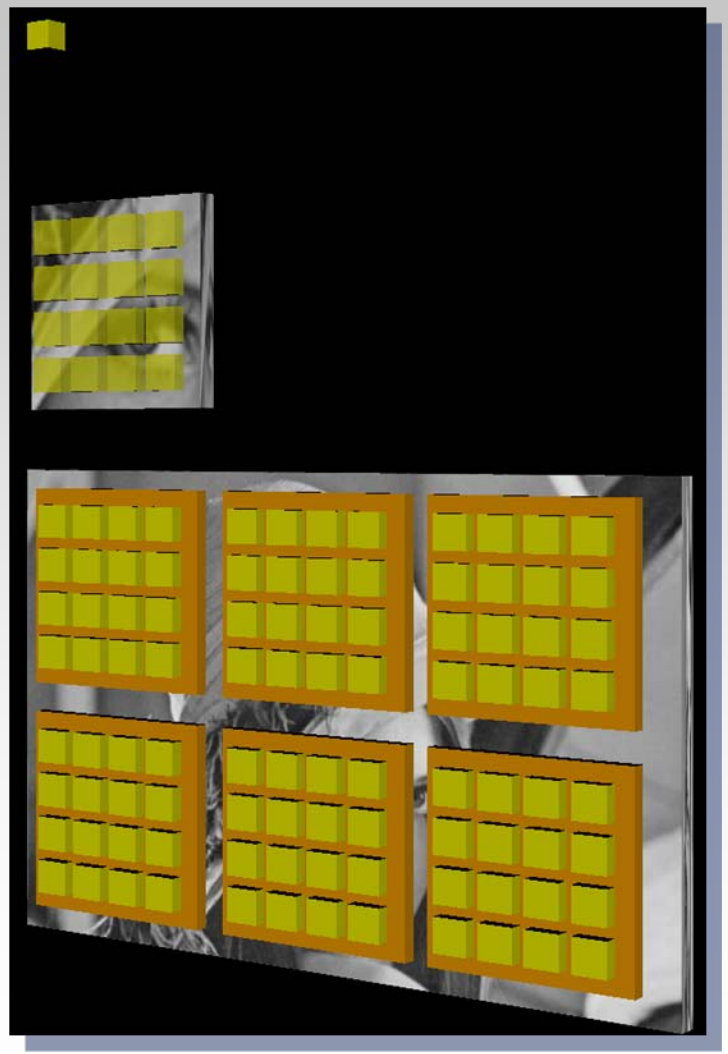
Individual THREADS operate on data

(OpenCL: Work Item)

Threads are grouped into 1, 2, or 3D BLOCKS, which can synchronize and cooperate

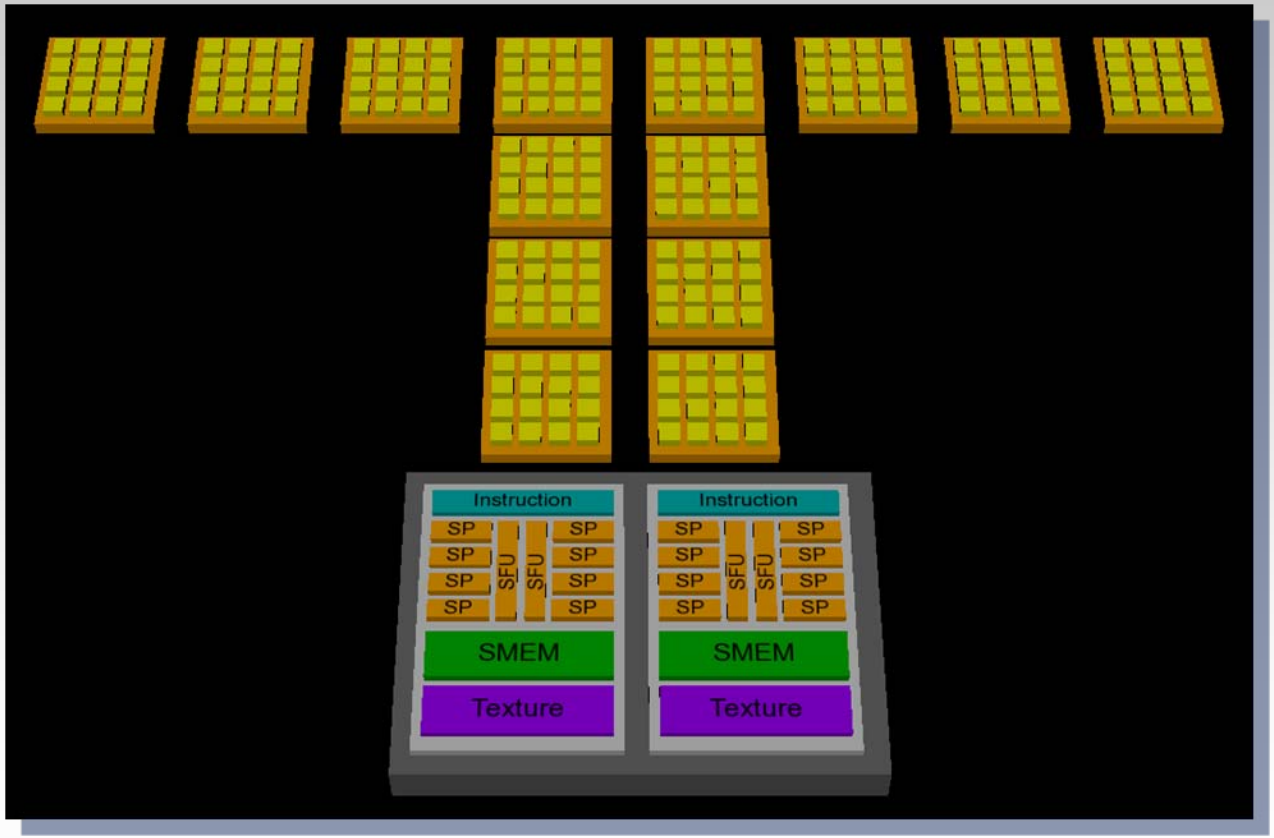
(OpenCL: Work Group)

A 1 or 2D GRID contains multiple blocks and covers the entire data set



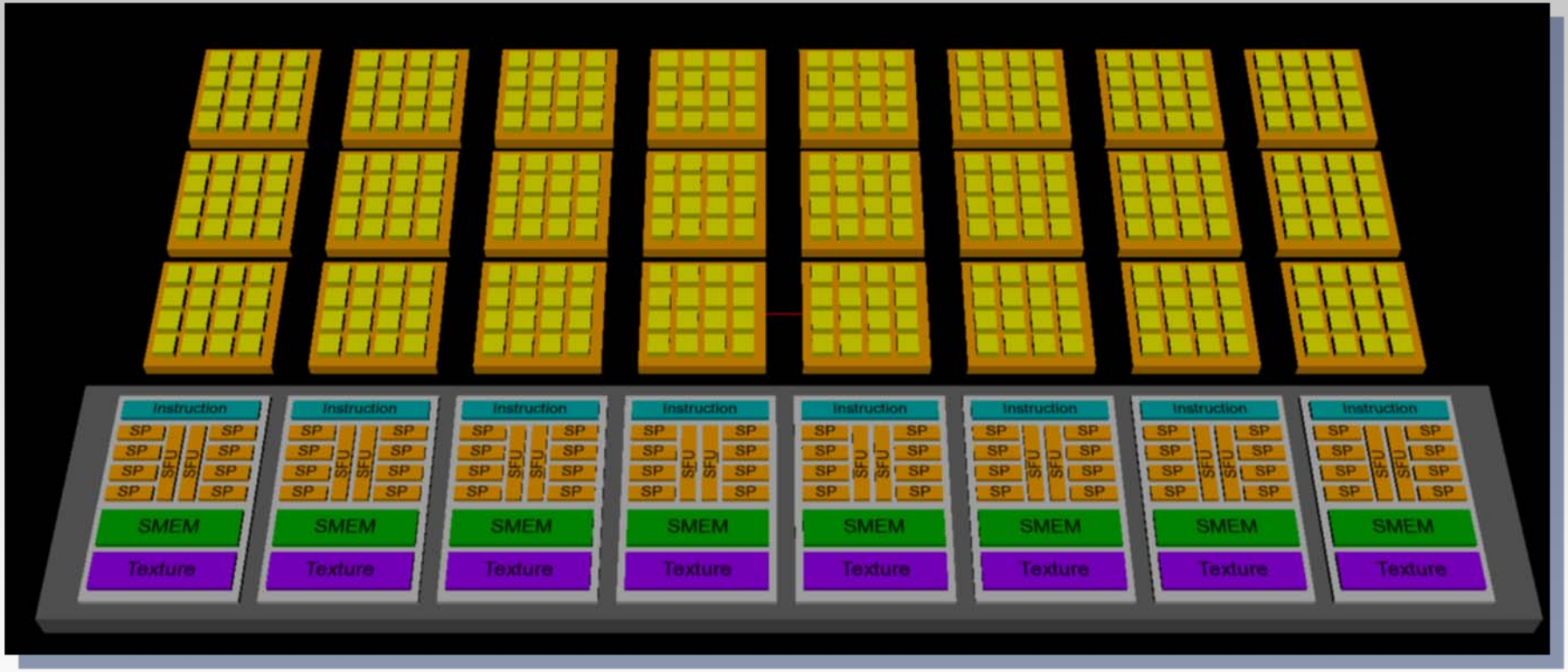
Device Scalability

Write once* – *scale from few to many cores



Small GPU, Blocks are executed sequentially

Device Scalability



Large GPU, Blocks are distributed over more SMs

SIMT Architecture

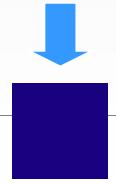
Single Instruction Multiple Thread

- Every machine instruction operates on a Warp of 32 threads
- Fetch & Decode is leveraged for multiple computations
- More flexible than SIMD architectures, divergence is permitted
- Explicate vectorization is not required

CPU

Each instruction operates on 1 data item

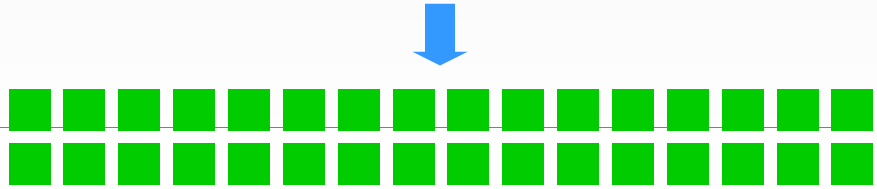
```
10052614  mov     eax,dword ptr [istar
10052617  mov     dword ptr [y],eax
1005261A  jmp     ConvoCPPWorker+2B5h
1005261C  mov     eax,dword ptr [y]
1005261F  add     eax,1
10052622  mov     dword ptr [y],eax
10052625  mov     eax,dword ptr [y]
```



GPU

Each instruction operates on a 32 thread Warp

```
ld.shared.u8     %r166, [%r149+7];
cvt.rn.f32.u32   %f37, %r166;     //
mul.f32          %f38, %f27, %f5;
mad.f32          %f39, %f2, %f6, %f38;
mad.f32          %f40, %f37, %f4, %f39;
mad.f32          %f12, %f15, %f3, %f40;
```



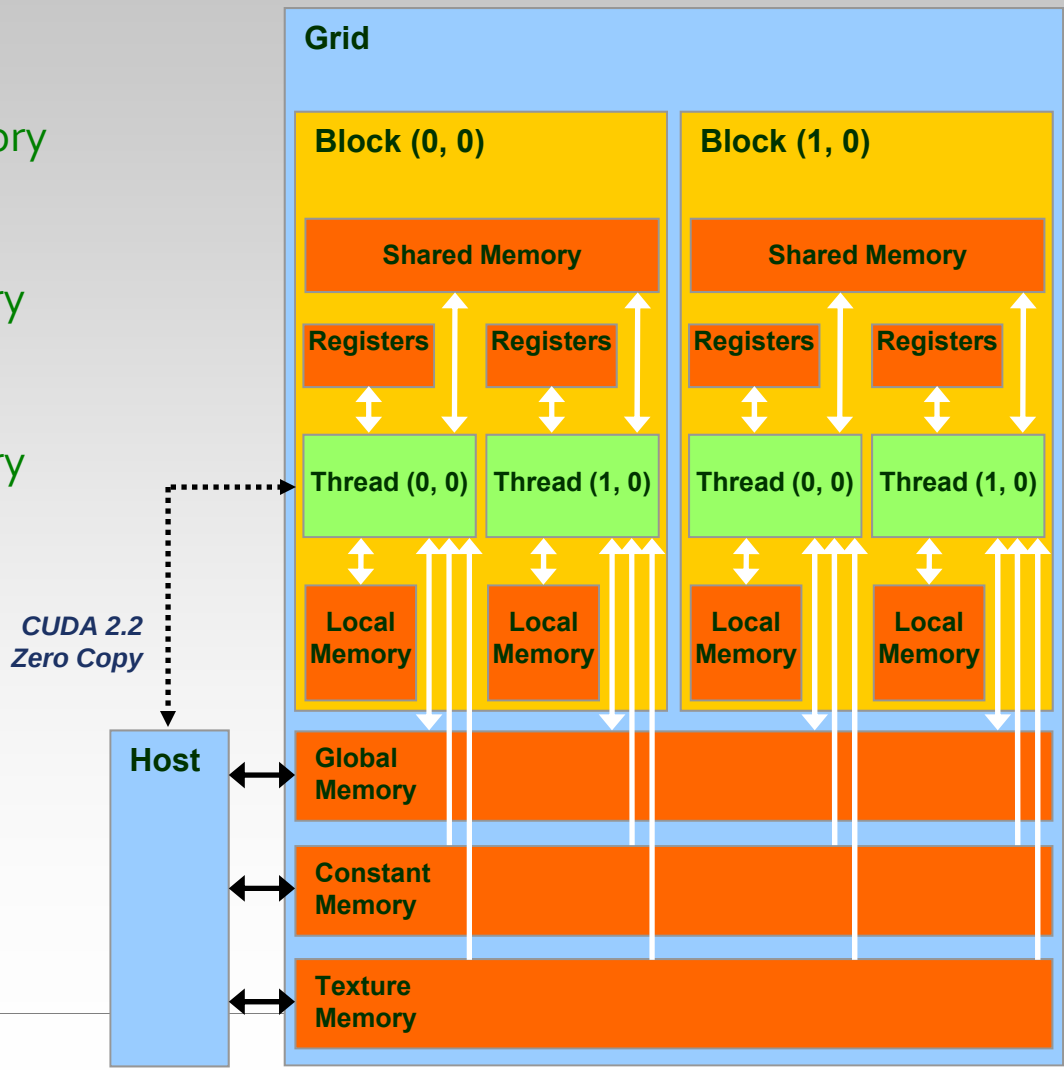
Programming Model: Memory Spaces

Each thread can:

- Read/write per-thread registers
- Read/write per-thread local memory
- Read/write per-block shared memory
- Read/write per-grid global memory
- Read only per-grid constant memory
- Read only per-grid texture memory

The host can read/write global, constant, and texture memory (stored in DRAM)

CUDA 2.2 adds "Zero-Copy", Threads can read & write directly to CPU memory
(SM 1.2 or later. e.g.: Tesla C/S1060 & Quadro FX 5800, 4800, 3800)



Registers

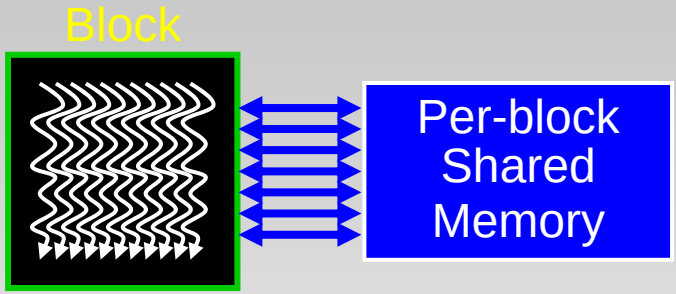
- Pre-thread high-speed registers located in SM
- Used for local-variables in a thread program (up to a limit)
- Only accessible within a single thread
- 8192 or 16384 Registers per SM
- Fast hardware context switching occurs by changing active register file

Local Memory

- Storage for local variables & arrays which do not fit into registers
- Resides in GMEM

Registers & Local memory are both called PRIVATE memory in OpenCL

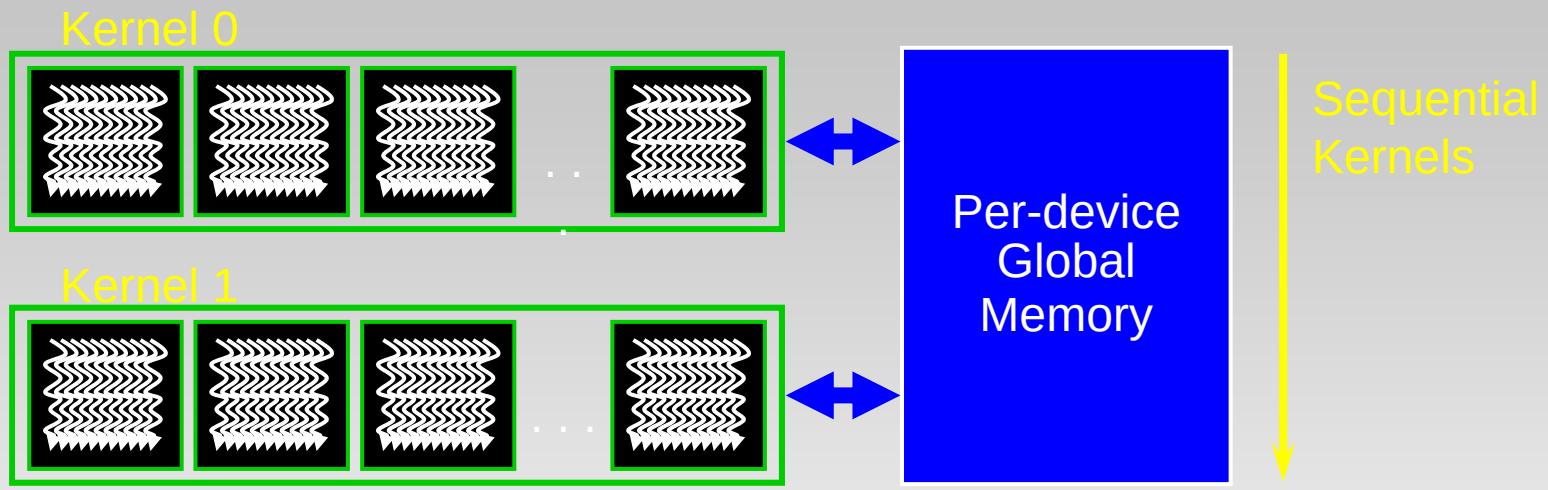
Shared Memory (SMEM)



- High-Speed cache located within each SM
- All threads within a thread block can read & write
- Used for data shared between threads (e.g. a tile of an image)
- Very fast!
- 16 KB / SM
- Banked into 16 - 1K Banks

OpenCL: Shared memory is called LOCAL memory (do not confuse with C Local memory)

Global Memory (GMEM)

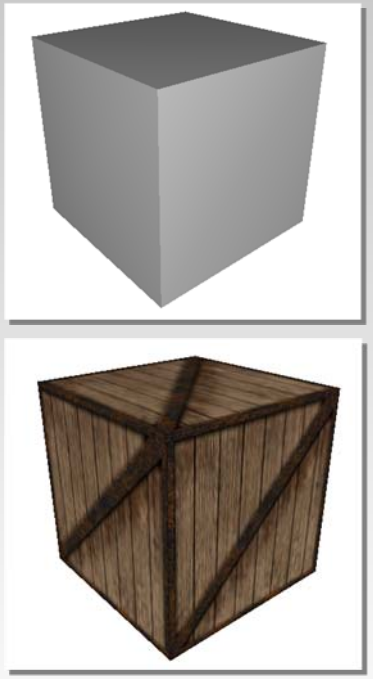


- DRAM located external to GPU (size dependent on configuration)
- Host CPU can read & write
- Relatively *slow*
- All threads can read and write to GMEM at any time
- Non-cached
- Performs best with coherent access patterns

OpenCL: Also called GLOBAL memory

Textures

- Leverage hardware commonly used in Graphics to apply materials to 3D geometry
- Data stored in GMEM, but cached in texture cache
- Allows fast non-uniform access to memory
- 1 function performs:
  - 2-D address computation
  - Boundary clamp
  - Type Conversion
  - Bi-linear interpolation



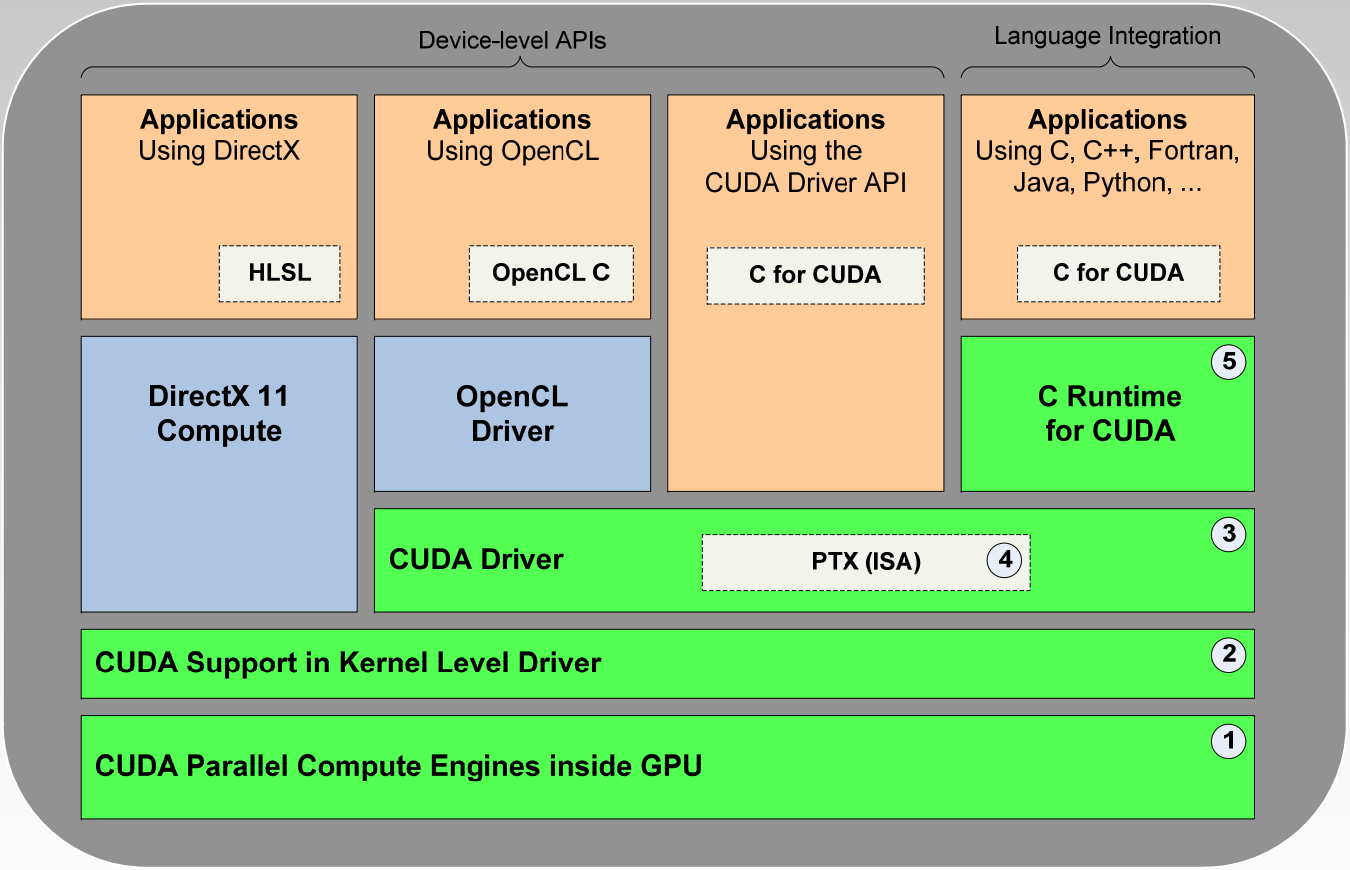
OpenCL: Textures are called IMAGES

Constant Memory

- Cached High-Speed Read-Only Constant data
- 64 KB max per program
- Statically allocated
- Set by Host-CPU prior to thread launch
- Useful for Convolution Kernels, Correlation templates, etc…

OpenCL: Also called CONSTANT memory

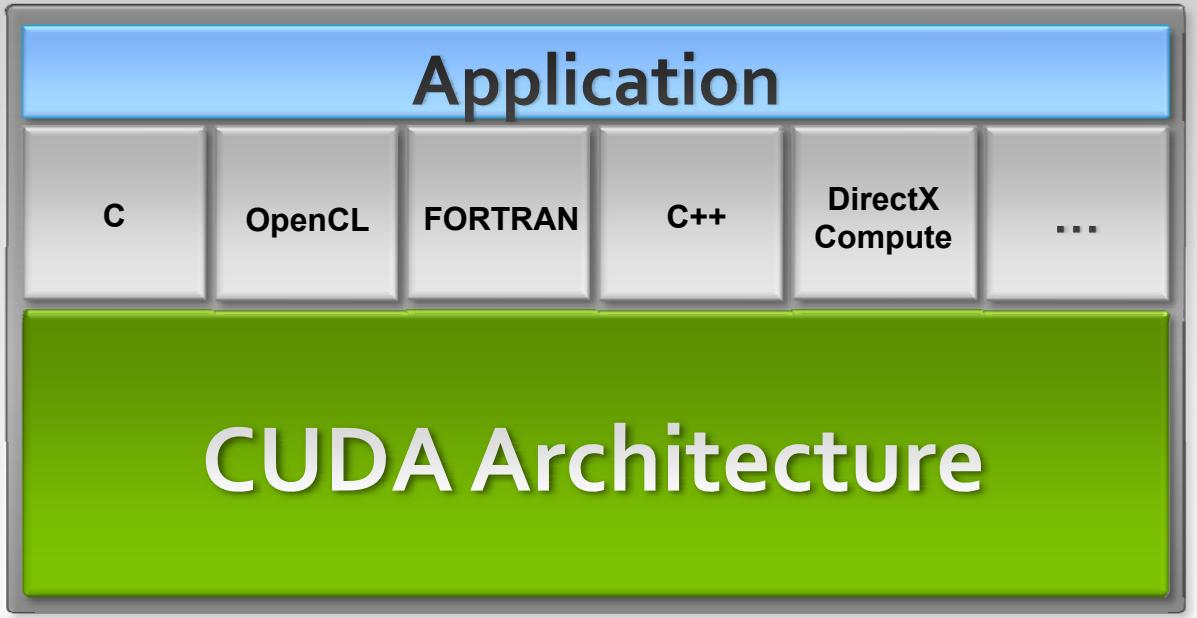
NVIDIA CUDA – GPU Programming Framework



Includes: Libraries, Tools, C-Runtime, Compiler, Debugger, Documentation & Samples

CUDA ‘C’ Programming

- CUDA Kernels may be written in several languages
- We’ll focus on ‘C’ for this session



CUDA – C: Host Component

CPU functions to command & control the GPU

Initialization, Queries, Data movement, Kernel Launch, Synchronization

- **Runtime API** – Facilitates language integration with standard C/C++ code
- **Driver API** – Lower level direct manual control of the GPU

CUDA – C: Kernel Functions

- Functions written to run on the GPU
- Compiled by NVIDIA's nvcc compiler
- Binary code is loaded to GPU automatically by Runtime API, or explicitly by Driver API

THREADS

- Unit of parallelization
- A hardware concept
- Scheduled & context switched in hardware

KERNELS

- Functions run by every thread
- A software concept
- Built in variables allow provide each kernel the Thread & Block Index of the calling thread

Example of a CUDA C Program

CPU program

```
void inc_cpu(int *a, int N)
{
   int idx;

   for (idx = 0; idx<N; idx++)
     a[idx] += 1;
}


int main()
{
    ...
    inc_cpu(a, N);
}
```

CUDA C program

```
__global__ void inc_gpu(int *a, int N)
{
    int idx = blockIdx.x * blockDim.x
                         + threadIdx.x;
    if (idx < N)
     a[idx] += 1;
}


int main()
{
  …
  int blockSize = 256;
  int nBlocks = N / blockSize;
  inc_gpu<<< nBlocks, blockSize >>>(a, N);
}
```

Host Run-Time Component

Functions which run on the CPU to control or communicate with the GPU

- Device Management
- Memory Allocation
- Memory Copy (CPU→GPU, GPU→GPU, GPU→CPU)
- Thread Launch
- Synchronization

```
e.g:

VectorAddKernel<<<SIZE/128,128>>>(GPUVector1, GPUVector2, GPUOutputVector);

cudaMemcpy(HostOutputVector,GPUOutputVector, SIZE*sizeof(float),cudaMemcpyDeviceToHost);
```

Device Management

- CPU can query and select GPU devices
  - `cudaGetDeviceCount( int* count )`
  - `cudaSetDevice( int device )`
  - `cudaGetDevice( int  *current_device )`
  - `cudaGetDeviceProperties( cudaDeviceProp* prop, int device )`
- Multi-GPU setup:
  - device 0 is used by default
  - one CPU thread can control only one GPU

GPU Memory Allocation / Release

- `cudaMalloc(void ** pointer, size_t nbytes)`
- `cudaMemset(void * pointer, int value, size_t count)`
- `cudaFree(void* pointer)`

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**)&d_a,  nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Data Copies

- `cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
  - returns after the copy is complete
  - blocks CPU thread
  - doesn't start copying until previous CUDA calls complete

- `enum cudaMemcpyKind`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`

Host Pinned Memory

- Facilitates DMA transfers between CPU & GPU – can double bandwidth
- Necessary for Asynchronous transfers
  - `cudaMallocHost(void ** pointer, size_t nbytes)`
  - `cudaFreeHost(void* pointer)`

- CUDA 2.2 Adds new Pinned Memory API
  - `cudaHostAlloc( void ** pointer, size_t nbytes, unsigned int flags )`
  - Flags:

`cudaHostAllocPortable`	Allows pinned buffers to be shared between CUDA contexts
`cudaHostAllocMapped`	Enables memory for Zero-Copy
`cudaHostAllocWriteCombined`	Allocates write-combined memory. WC memory is uncached by CPU and allows PCIe speeds, but reduced CPU read performance.

“Zero-Copy”

- Threads directly access pinned CPU memory as if it was regular GPU GMEM
- Read/Write over the PCIe bus is slower, but no copy overhead
- Can Overlap PCIe transfers time with compute
- SM 1.2 or later only
- API:
  - Allocate CPU memory with **`cudaHostAlloc`**. Use **`cudaHostAllocMapped`** flag.
  - Retrieve a GPU memory pointer to the host memory:
    `cudaHostGetDevicePointer``( void **pDevice, void *pHost, unsigned int Flags );`
- When to use
  - Reading / writing each data item only once
  - Very small data size (constants, state updates, coefficients)
  - Status updates & flag to or from running GPU kernels
  - Memory can be read from pinned cpu memory in the kernel and then written to GMEM for future fast access

Launching kernels on GPU

- Launch parameters:
  - grid dimensions: x and y
  - thread-block dimensions: x, y, and z
  - shared memory: number of bytes per block, for extern smem variables declared without size
    - optional, **0** by default
  - stream ID
    - optional, **0** by default

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block, 0, 0>>>(...);
kernel<<<32, 512>>>(...);
```

Host Synchronization

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel starts executing once all previous CUDA calls have completed
- Memcopies are synchronous
  - control returns to CPU once the copy is complete
  - copy starts once all previous CUDA calls have completed
- `cudaThreadSynchronize()`
  - blocks until all previous CUDA calls complete
- Async API provides:
  - GPU CUDA-call streams
  - non-blocking memcopies
  - Copy / Compute overlap

Code executed on GPU

- C function with some restrictions:
  - Can only access GPU memory (or Zero Copy mapped mem)
  - No variable number of arguments
  - No static variables

- Must be declared with a qualifier:
  - `__global__` : launched by CPU,
    cannot be called from GPU
    must return void

  - `__device__` : called from other GPU functions,
    cannot be launched by the CPU

Built in Variables

Hardware variables allow threads to know their relative place

- `gridDim` - Grid Dimension (1-D or 2-D)
- `blockDim` - Block Dimension (1-D, 2-D, or 3-D)
- `blockIdx` – Index of the block for the current thread
- `threadIdx` – Index of the current thread

Built-in Vector Types

- Can be used in GPU and CPU code
- **`[u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4]`**
  - Structures accessed with **`x`**, **`y`**, **`z`**, **`w`** fields:

```
uint4 param;
int y = param.y;
```

- **`dim3`**
  - Based on **`uint3`**
  - Used to specify dimensions
  - Default value (1,1,1)

Thread Synchronization Function

- **`void __syncthreads();`**
- Synchronizes all threads in a block
  - Once all threads have reached this point, execution resumes normally
  - Used to avoid RAW / WAR / WAW hazards when accessing shared memory
- Allowed in conditional code only if the conditional is uniform across the entire thread block

Examples:

CUDA “Hello World”,

Image Brightness/Contrast Adjust

Using Shared Memory

- Hundred times faster than global memory
- Cache data to reduce global memory accesses
  - Example: load an image tile into SMEM
- Threads can cooperate via shared memory
  - Divide-and-conquer computations – separable convolution or correlation filter
- Use it to avoid non-coalesced access
  - Stage loads and stores in shared memory to re-order non-coalesceable addressing

SMEM Declaration

Two ways to declare:

- Static Case:

  Use `__shared__` keyword within a kernel program, e.g.:

```
__shared__ float my_smem[128];
```

- Dynamic Case:

  3rd parameter of kernel launch specifies number of SMEM bytes:

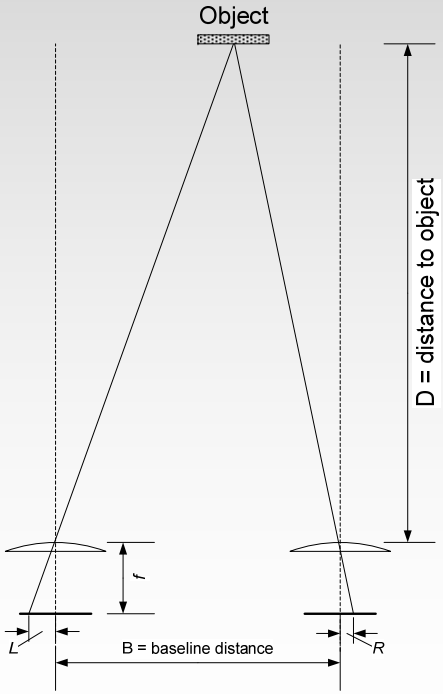
```
KernelFunc<<<GridSz,BlockSz, SMEM_bytes>>>(…);
```

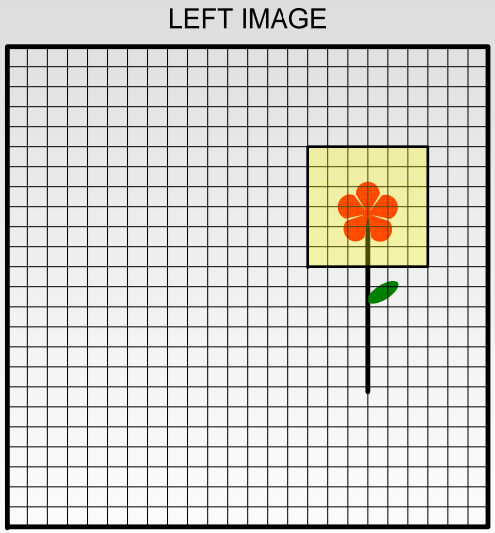
  Within the kernel program:

```
extern __shared__ float * my_smem;
```

SMEM example: Stereo Vision

- Computationally Insane: Need to find correspondence between left and right image for every pixel
- For 640 x 480 pixels, 11x11 block match, this is ~2 Billion computations for 50-pixel search (with no optimization)





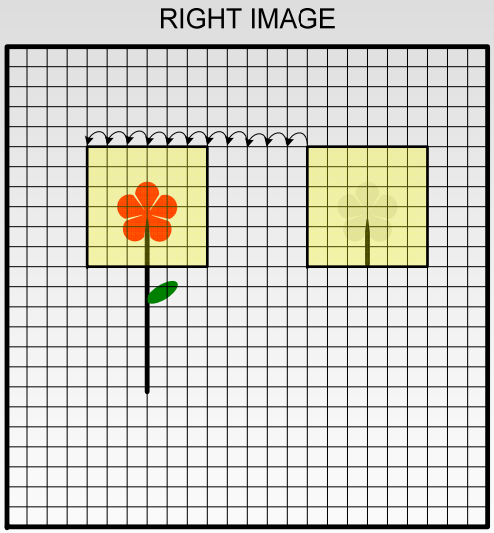
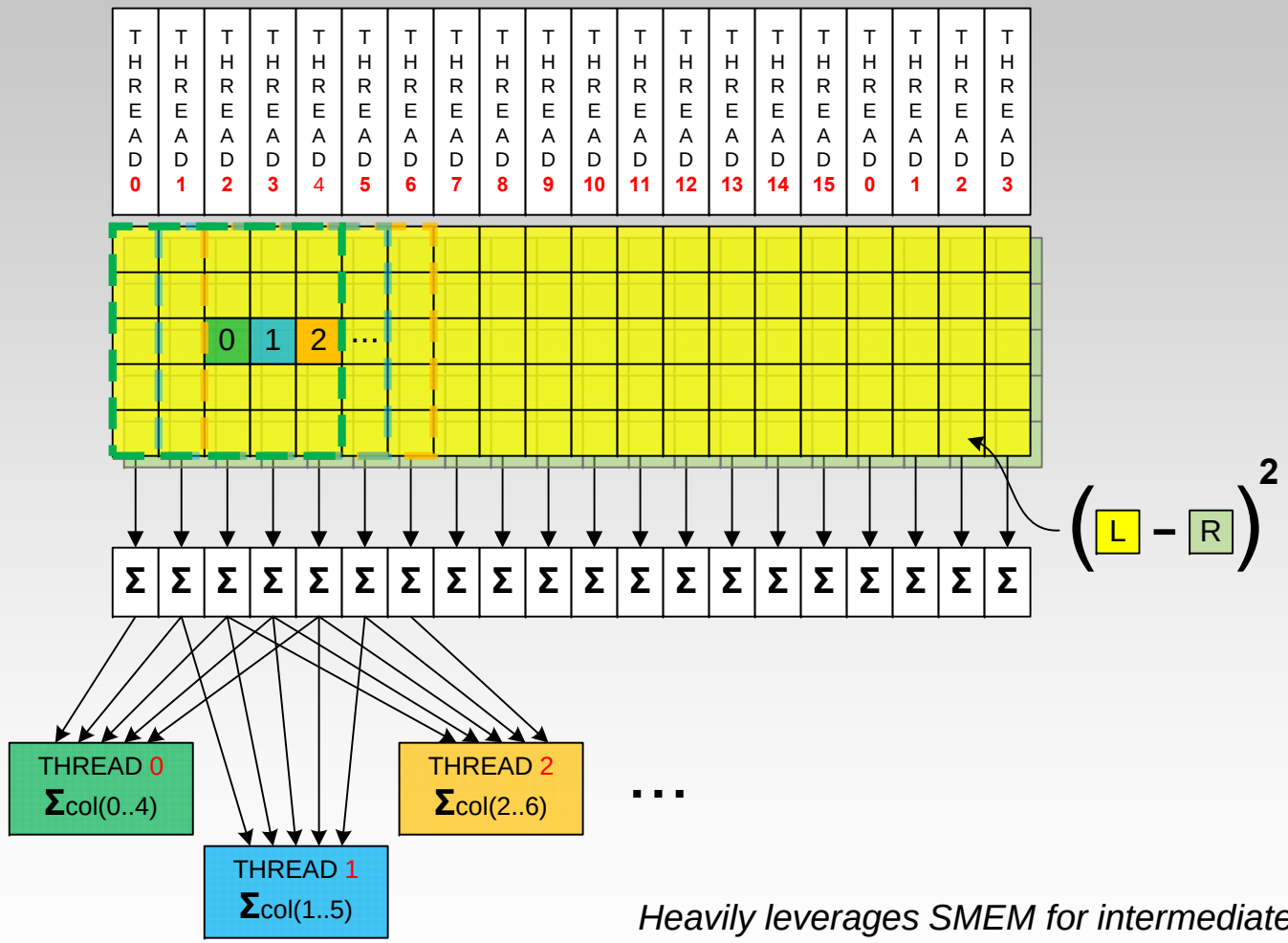


FIGURE 1

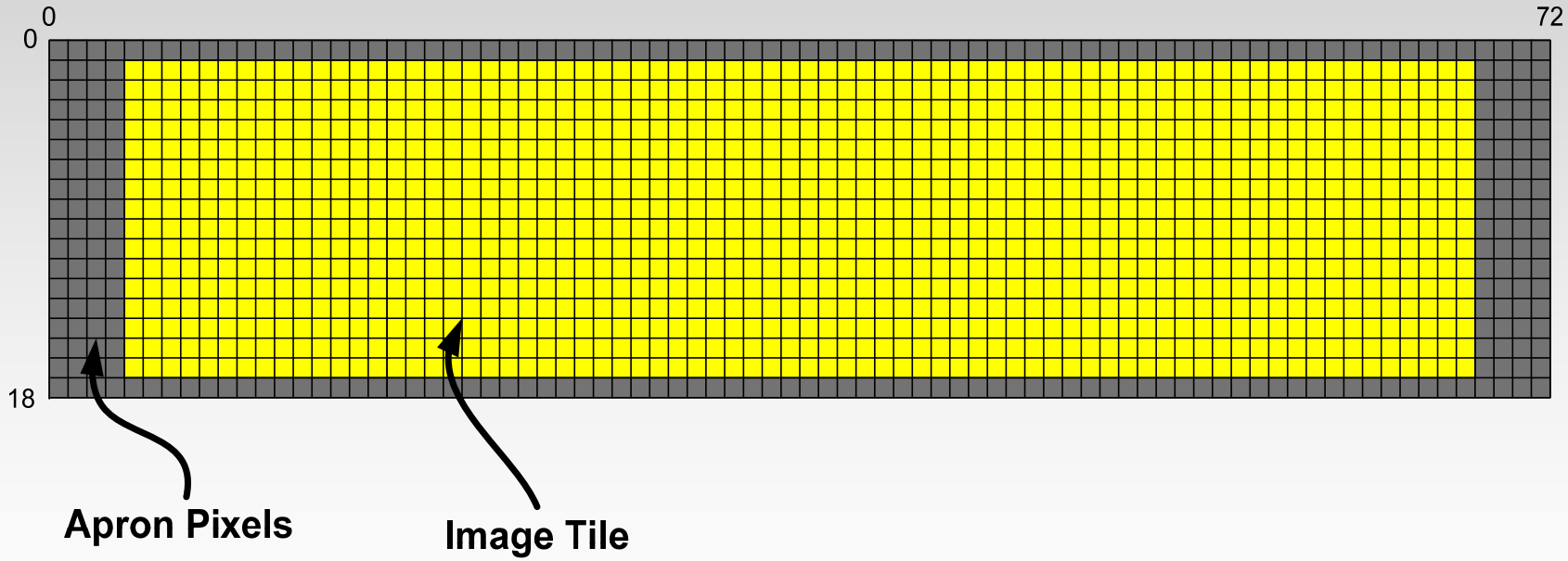
GPU Strategy – Thread Cooperation



Heavily leverages SMEM for intermediate computations to share work between threads

SMEM Convolution

- Load a tile of the image into SMEM, then process
- Problem: need to load apron pixels (and what about coalescing?)

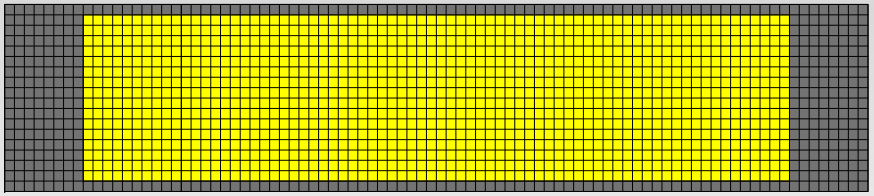


e.g. 16x16 bock. Process 4 8-bit pixels / block, read 72x18 pixels

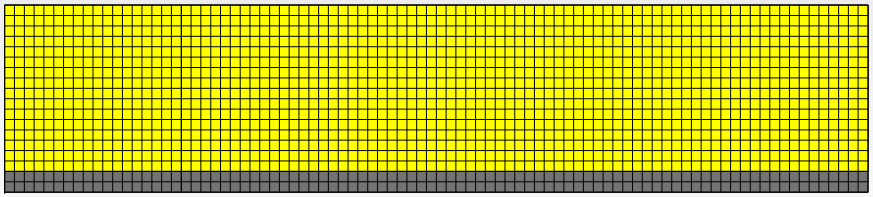
Apron Reading – GT200

New coalescing buffers make it easy!

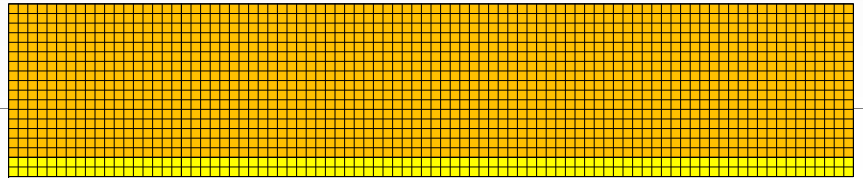
- Use 64-bit reads, only ½ of the threads (plus apron) read, but read 8 pixels per thread instead of 4
- Use a 16x16 thread block, all threads process 4 pixels (or an RGBA pixel)
- Need a slightly larger (80 x 18 pixel) tile in memory



Step 1: Threads with threadIdx.x < 10 read top 16 rows, 8 pixel at once



Step 2: Threads with threadIdx.y < 2 and threadIdx.x < 10 read bottom two rows



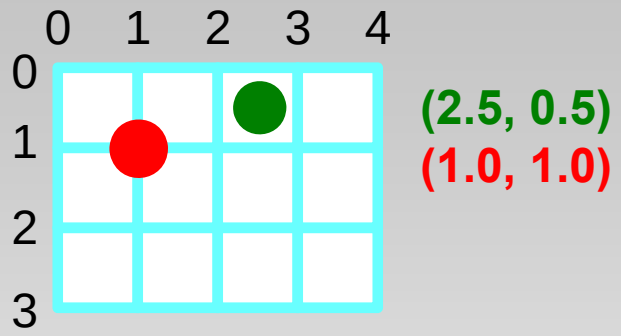
Using Constant Memory

- Fast, Cached, Read-Only Memory
- Data in Constant Memory must be written prior to Kernel Launch
- 64Kb total Constant Memory
- Variables must be declared statically with `__constant__` keyword
- Use `cudaMemcpyToSymbol` to set values of constant memory
- Fastest when all threads read the same value

Using Texture Memory

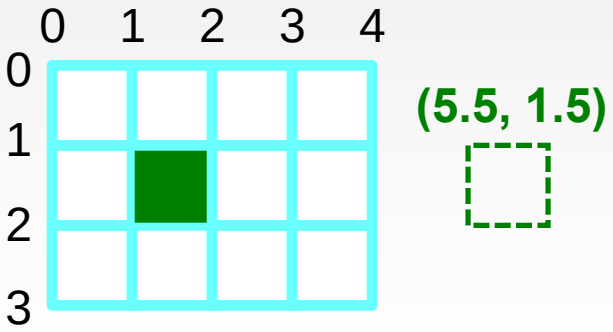
- Texture is an object for ***reading*** data
- Benefits of CUDA textures:
  - Texture fetches are cached
    - optimized for 2D locality
  - Textures are addressable in 2D
    - using integer or normalized coordinates
    - means fewer addressing calculations in code
  - Provide filtering for free
  - Free wrap modes (boundary conditions)
    - clamp to edge / repeat

Texture Addressing



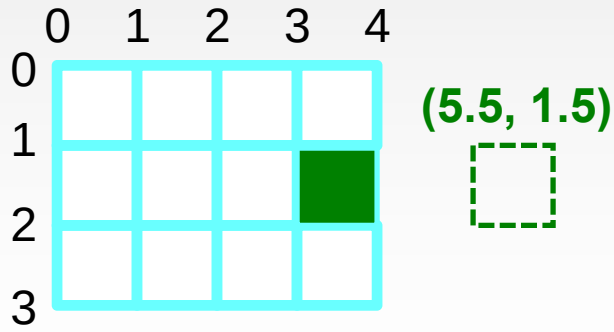
Wrap

- Out-of-bounds coordinate is wrapped (modulo arithmetic)



Clamp

- Out-of-bounds coordinate is replaced with the closest boundary



Two CUDA Texture Types

- Bound to linear memory
  - Global memory is bound to a texture
  - `cudaBindTexture`
  - `cudaBindTexture2D`
- Bound to CUDA arrays
  - CUDA array is bound to a texture
  - `cudaBindTextureToArray`
  - Requires a copy to a cudaArray, but may have higher performance for certain 2D access patterns

CUDA Texturing Steps

- Host (CPU) code:
  - Allocate/obtain memory (global linear, or CUDA array)
  - Create a texture reference object
    - Currently must be at file-scope
  - Bind the texture reference to memory/array
  - When done:
    - Unbind the texture reference, free resources
- Device (kernel) code:
  - Fetch using texture reference
  - **`tex1D(), tex2D() or tex3D()`** functions

Examples:

Image Rotation

With Textures

OpenGL Interoperability

- OpenGL buffer objects can be mapped into the CUDA address space and then used as global memory
  - Vertex buffer objects
  - Pixel buffer objects
- Direct3D9 Vertex objects can be mapped
- Data can be accessed like any other global data in the device code
- Image data can be displayed from pixel buffer objects using glTexSubImage2D
  - Requires copy in video memory, but still fast

OpenGL Interop Steps

1. Register a buffer object with CUDA
   - `cudaGLRegisterBufferObject(GLuint buffObj);`
   - OpenGL can use a registered buffer only as a source
   - Unregister the buffer prior to rendering to it by OpenGL
2. Map the buffer object to CUDA memory
   - `cudaGLMapBufferObject(void **devPtr, GLuint buffObj);`
   - Returns an address in global memory
   - Buffer must registered prior to mapping
3. Launch a CUDA kernel to process the buffer (or copy into it)
4. Unmap the buffer object prior to use by OpenGL
   - `cudaGLUnmapBufferObject(GLuint buffObj);`
5. Unregister the buffer object
   - `cudaGLUnregisterBufferObject(GLuint buffObj);`
   - ***Optional***: needed if the buffer is a render target
6. Use the buffer object in OpenGL code

GPU Atomic Integer Operations

- Atomic operations on integers
  - Associative operations on signed/unsigned ints
  - add, sub, min, max, …
  - and, or, xor
- Operations on Global memory require SM 1.1 or later
- Operations on Shared memory require SM 1.2 or later

CUDA Event API

- Events are inserted (recorded) into CUDA call streams
- Usage scenarios:
  - measure elapsed time for CUDA calls (clock cycle precision)
  - query the status of an asynchronous CUDA call
  - block CPU until CUDA calls prior to the event are completed

```
cudaEvent_t start, stop;
cudaEventCreate(start);  cudaEventCreate(stop);
cudaEventRecord(start, 0);
kernel<<<grid, block>>>(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(0);
float et;
cudaEventElapsedTime(&et, start, stop);
cudaEventDestroy(start);  cudaEventDestroy(stop);
```

Streams for Asynchronous Operations

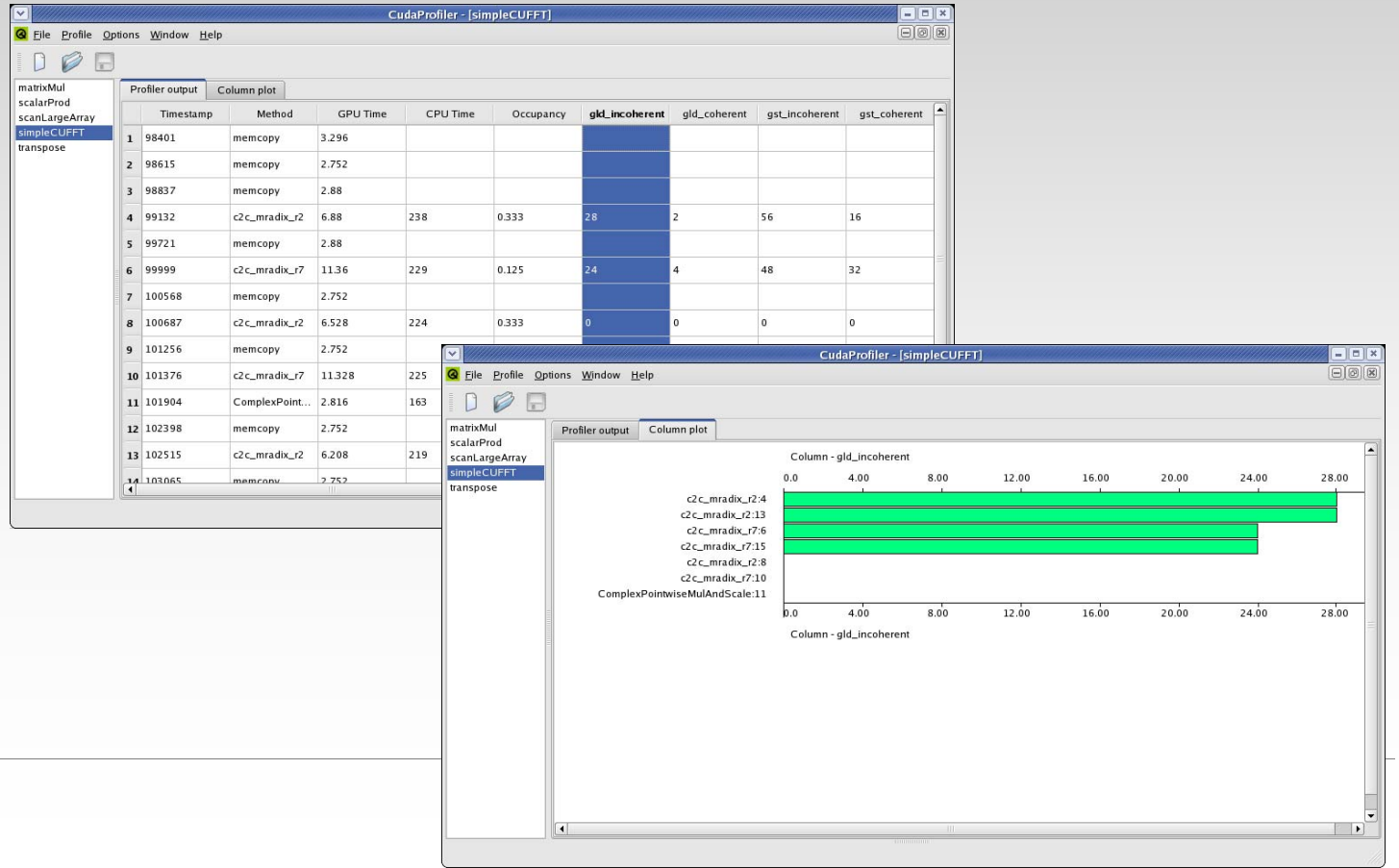
- Asynchronous host ←→ device memory copy for pinned memory (allocated with "cudaMallocHost" in C) frees up CPU on all CUDA capable devices
- Overlap implemented by using a stream
- Stream = Sequence of operations that execute in order
- Stream API:
  - o = default stream
  - `cudaMemcpyAsync(dst, src, size, direction, 0);`

CUDA Optimization

- Start with Optimizing Memory Access
  - Ensure GMEM access is coalesced
  - Eliminate redundant accesses by using SMEM
  - Explore 64-bit and 128-bit reads
- Check Occupancy
  - Multiple threads on the GPU hide memory latency
  - Multiple blocks keep and SM busy when another block is awaiting a `__syncthreads()`

CUDA Profiler

- Uses a special operation mode of the GPU to log important signals
- Best to isolate kernels in a simple application
- Visual Profiler doesn't easily allow user i/o



OpenCL & ATI Radeon™ 4000 Series Overview

Justin Hensley, Ph.D.
Office of the CTO
AMD

Agenda

- OpenCL Introduction & Overview
- ATI Radeon™ HD 4000 Series Overview
- “Tips & Tricks”
- Questions

Quick Recap

1. OpenGL - **cross platform** graphics

2. "Brook for GPU" - project from Stanford
   - brcc translates .br files into .cpp files with pixel shaders
   - Derived from Brook, a "true" stream processing language

3. CTM - Close to the metal
   - First GPGPU API released by a major GPU vendor
   - Steep (near vertical) learning curve
   - Amazing ability to abuse the GPU for things its not supposed to do

4. CAL - Compute Abstraction Layer / Brook+
   - Source for Brook+ available as part of AMD Stream SDK

5. OpenCL - **cross-platform** compute

Video & Computer Vision Demos

Advanced Human Computer Interface Demo

AMD

Video & Computer Vision Demos

Video & Computer Vision Demos

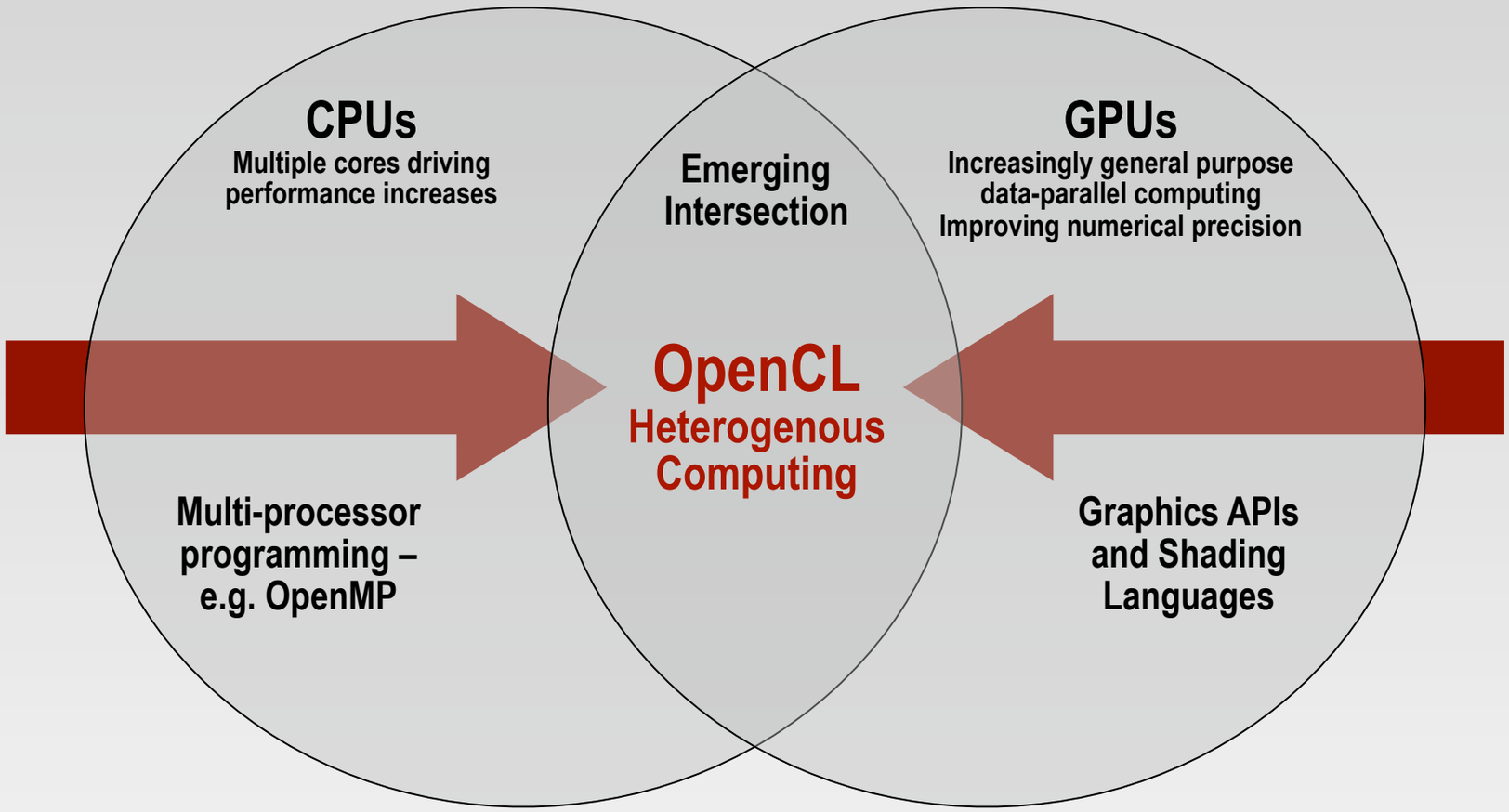
Why OpenCL

- Cross platform, heterogeneous API for applications with "ALU" dense sections of code
- AMD will offer optimized versions of both CPU and GPU "soon"
  - Eases programming of **multi-core CPU**s and **GPU**s
  - With CPU runtime its relatively easy to get get 80-90% of highly optimized SEE-ized multi-threaded application
- ***Balanced*** execution
  - Use the CPU for what its good for
  - Use the GPU for what its good for

OpenCL Working Group

- Diverse industry participation
  - Processor vendors, system OEMs, middleware vendors, application developers
- Many industry-leading experts involved in OpenCL's design
  - A healthy diversity of industry perspectives
  - Apple initially proposed the working group
  - Aftab Munshi from Apple is specification editor

Processor Parallelism



OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

OpenCL
Technical Overview

Overview

A Walk Through the OpenCL Specification:

- Design Requirements
- Architecture
- Framework
- Extended Features

OpenCL Design Requirements

- **Use all computational resources in system**
  - Program GPUs, CPUs, and other processors as peers
  - Support both data- and task- parallel compute models
- **Efficient C-based parallel programming model**
  - Abstract the specifics of underlying hardware
- **Abstraction is low-level, high-performance but device-portable**
  - Approachable – but primarily targeted at expert developers
  - Ecosystem foundation – no middleware or "convenience" functions
- **Implementable on a range of embedded, desktop, and server systems**
  - HPC, desktop, and handheld profiles in one specification
- **Drive future hardware requirements**
  - Floating point precision requirements
  - Applicable to both consumer and HPC applications

Anatomy of OpenCL

- **Language Specification**
  - C-based cross-platform programming interface
  - Subset of ISO C99 with language extensions - familiar to developers
  - Well-defined numerical accuracy - IEEE 754 rounding behavior with specified maximum error
  - Online or offline compilation and build of compute kernel executables
  - Includes a rich set of built-in functions

- **Platform Layer API**
  - A hardware abstraction layer over diverse computational resources
  - Query, select and initialize compute devices
  - Create compute contexts and work-queues

- **Runtime API**
  - Execute compute kernels
  - Manage scheduling, compute, and memory resources

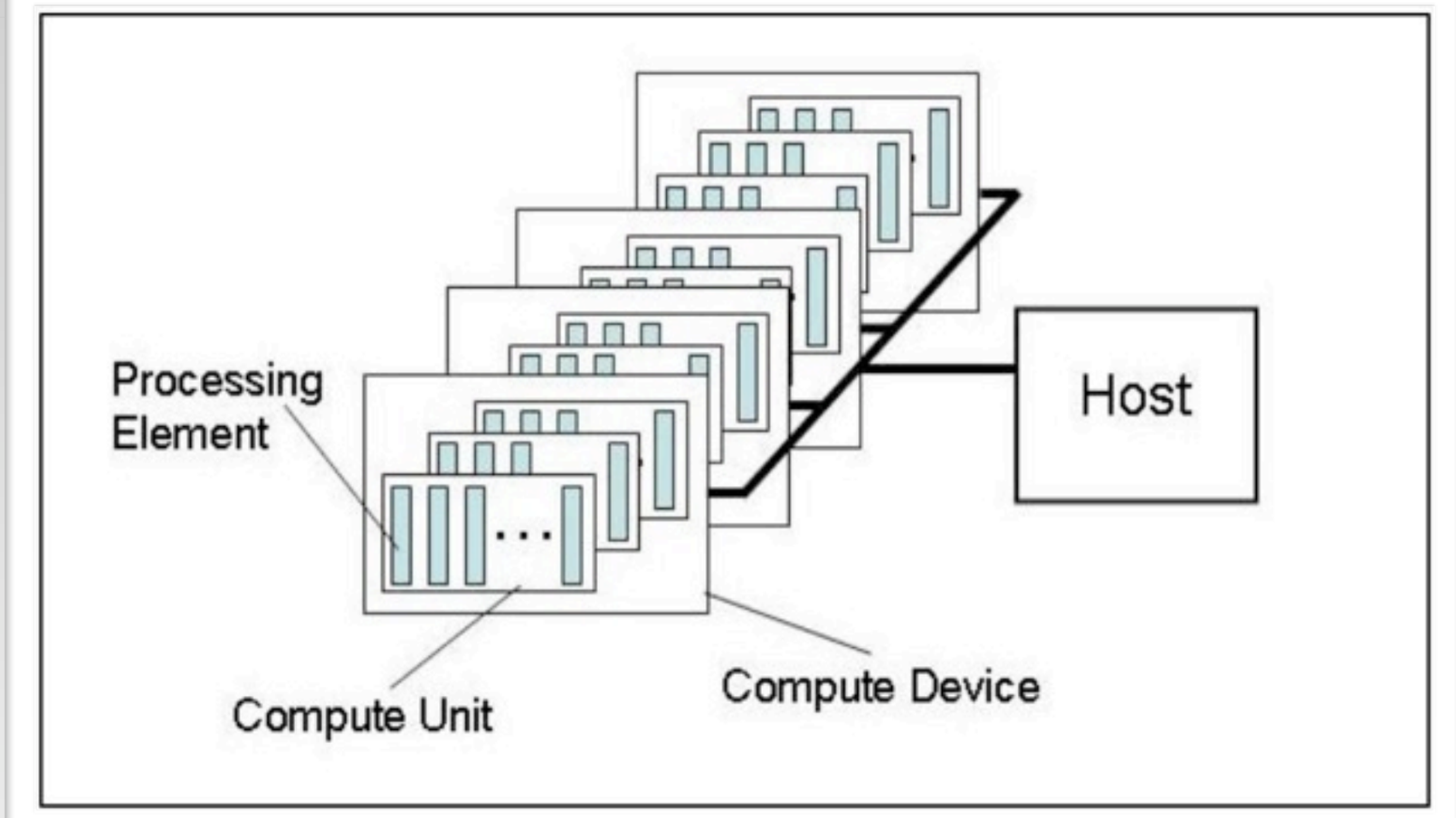
OpenCL Architecture

Chapter 3

Hierarchy of Models

- Platform Model
- Memory Model
- Execution Model
- Programming Model

OpenCL Platform Model (Section 3.1)



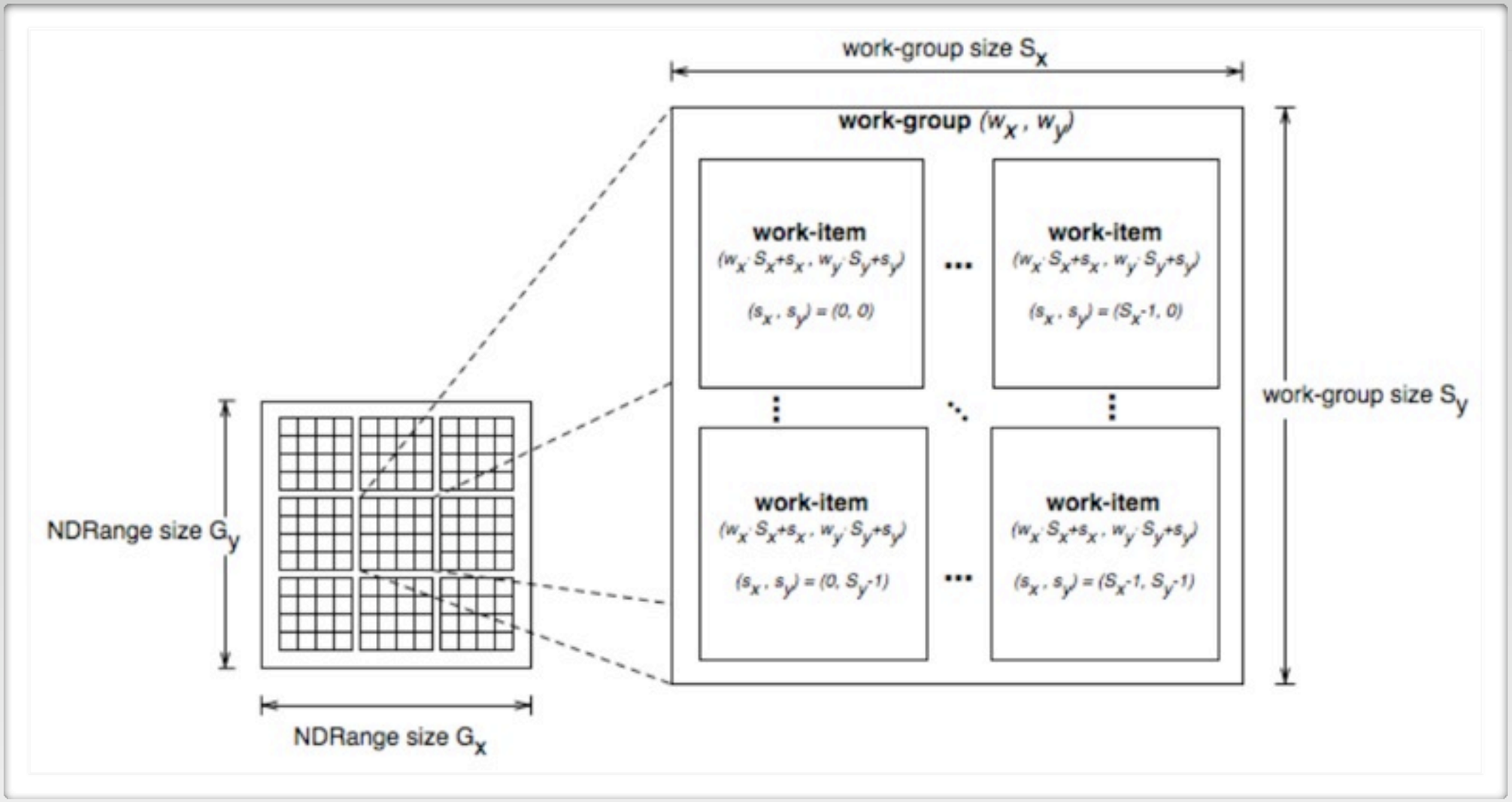
- One Host + one or more Compute Devices
  - Each Compute Device is composed of one or more Compute Units
    - Each Compute Unit is further divided into one or more Processing Elements

OpenCL Execution Model (Section 3.2)

- **OpenCL Program:**
  - Kernels
    - Basic unit of executable code — similar to a C function
    - Data-parallel or task-parallel
  - Host Program
    - Collection of compute kernels and internal functions
    - Analogous to a dynamic library

- **Kernel Execution**
  - The host program invokes a kernel over an index space called an ***NDRange***
    - NDRange = "N-Dimensional Range"
    - NDRange can be a 1, 2, or 3-dimensional space
  - A single kernel instance at a point in the index space is called a ***work-item***
    - Work-items have unique global IDs from the index space
  - Work-items are further grouped into ***work-groups***
    - Work-groups have a unique work-group ID
    - Work-items have a unique local ID within a work-group

Kernel Execution



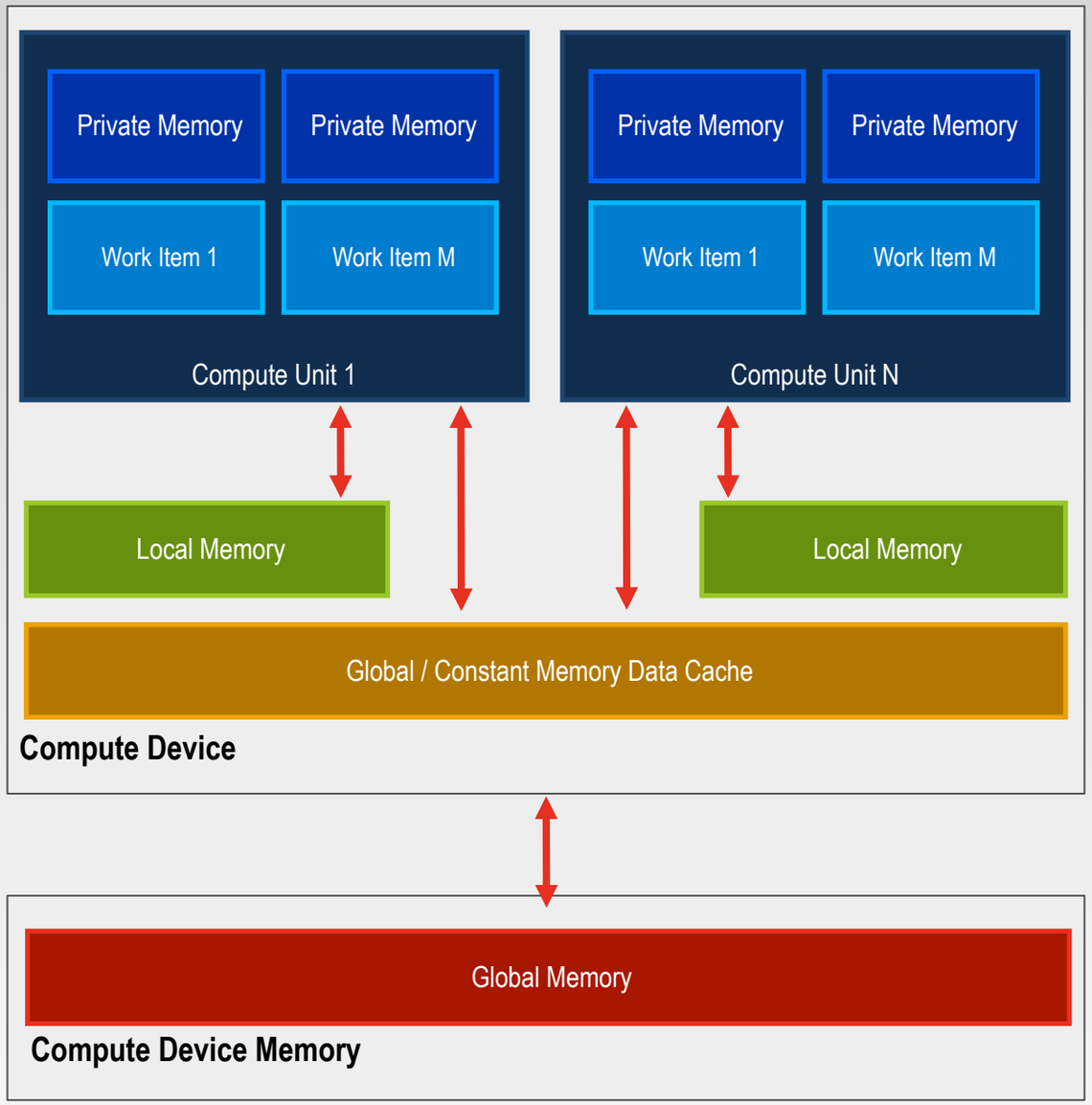
- Total number of work-items = G_x x G_y
- Size of each work-group = S_x x S_y
- Global ID can be computed from work-group ID and local ID

Contexts and Queues (Section 3.2.1)

- Contexts are used to contain and manage the state of the "world"
- Kernels are executed in contexts defined and manipulated by the host
  - Devices
  - Kernels - OpenCL functions
  - Program objects - kernel source and executable
  - Memory objects
- Command-queue - coordinates execution of kernels
  - Kernel execution commands
  - Memory commands - transfer or mapping of memory object data
  - Synchronization commands - constrains the order of commands
- Applications queue compute kernel execution instances
  - Queued in-order
  - Executed in-order or out-of-order
  - Events are used to implement appropriate synchronization of execution instances

OpenCL Memory Model (Section 3.3)

- **Shared memory model**
  - Relaxed consistency
- **Multiple distinct address spaces**
  - Address spaces can be collapsed depending on the device's memory subsystem
- **Address spaces**
  - Private - private to a *work-item*
  - Local - local to a *work-group*
  - Global - accessible by all work-items in all work-groups
  - Constant - read only global space
- **Implementations map this hierarchy**
  - To available physical memories



Memory Consistency (Section 3.3.1)

"OpenCL uses a relaxed consistency memory model; i.e. the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times."

- Within a work-item, memory has load/store consistency
- Within a work-group at a barrier, local memory has consistency across work-items
- Global memory is consistent within a work-group, at a barrier, but not guaranteed across different work-groups
- Consistency of memory shared between commands are enforced through synchronization

Data-Parallel Programming Model

(Section 3.4.1)

- Define N-Dimensional computation domain
  - Each independent element of execution in an N-Dimensional domain is called a *work-item*
  - N-Dimensional domain defines the total number of work-items that execute in parallel
    = *global work size*

- Work-items can be grouped together — ***work-group***
  - Work-items in group can communicate with each other
  - Can synchronize execution among work-items in group to coordinate memory access

- Execute multiple work-groups in parallel
  - Mapping of global work size to work-group can be implicit or explicit

Task-Parallel Programming Model

(Section 3.4.2)

- Data-parallel execution model must be implemented by all OpenCL compute devices
- Some compute devices such as CPUs can also execute task-parallel compute kernels
  - Executes as a single work-item
  - A compute kernel written in OpenCL
  - A native C / C++ function

OpenCL Framework

Language and API Details

Basic OpenCL Program Structure

- Host program
  - Query compute devices
  - Create contexts

  Platform Layer

  - Create memory objects associated to contexts
  - Compile and create kernel program objects
  - Issue commands to command-queue
  - Synchronization of commands
  - Clean up OpenCL resources

  Runtime

- Kernels
  - C code with some restrictions and extensions

  Language

Platform Layer (Chapter 4)

- **Platform layer allows applications to query for platform specific features**
- **Querying platform info (i.e., OpenCL profile)** (Chapter 4.1)
- **Querying devices** (Chapter 4.2)
  - *clGetDeviceIDs*()
    - Find out what compute devices are on the system
    - Device types include CPUs, GPUs, or Accelerators
  - *clGetDeviceInfo*()
    - Queries the capabilities of the discovered compute devices such as:
      - Number of compute cores
      - NDRange limits
      - Maximum work-group size
      - Sizes of the different memory spaces (constant, local, global)
      - Maximum memory object size
- **Creating contexts** (Chapter 4.3)
  - Contexts are used by the OpenCL runtime to manage objects and execute kernels on one or more devices
  - Contexts are associated to one or more devices
    - Multiple contexts could be associated to the same device
  - *clCreateContext*() and *clCreateContextFromType*() returns a *handle* to the created contexts

Command-Queues (Section 5.1)

- Command-queues store a set of operations to perform
- Command-queues are associated to a context
- Multiple command-queues can be created to handle independent commands that don't require synchronization
- Execution of the command-queue is guaranteed to be completed at sync points

Memory Objects (Section 5.2)

- Buffer objects
  - One-dimensional collection of objects (like C arrays)
  - Valid elements include scalar and vector types as well as user defined structures
  - Buffer objects can be accessed via pointers in the kernel
- Image objects
  - Two- or three-dimensional texture, frame-buffer, or images
  - Must be addressed through built-in functions
- Sampler objects
  - Describes how to sample an image in the kernel
    - Addressing modes
    - Filtering modes

Creating Memory Objects

- ***clCreateBuffer()***, ***clCreateImage2D***(), and ***clCreateImage3D***()
- Memory objects are created with an associated context
- Memory can be created as read only, write only, or read-write
- Where objects are created in the platform memory space can be controlled
  - Device memory
  - Device memory with data copied from a host pointer
  - Host memory
  - Host memory associated with a pointer
    - Memory at that pointer is guaranteed to be valid at synchronization points
- Image objects are also created with a channel format
  - Channel order (e.g., RGB, RGBA ,etc.)
  - Channel type (e.g., UNORM INT8, FLOAT, etc.)

Manipulating Object Data

- Object data can be copied to host memory, from host memory, or to other objects
- Memory commands are enqueued in the command buffer and processed when the command is executed
  - *clEnqueueReadBuffer*(), *clEnqueueReadImage*()
  - *clEnqueueWriteBuffer*(), *clEnqueueWriteImage*()
  - *clEnqueueCopyBuffer*(), *clEnqueueCopyImage*()
- Data can be copied between Image and Buffer objects
  - *clEnqueueCopyImageToBuffer*()
  - *clEnqueueCopyBufferToImage*()
- Regions of the object data can be accessed by mapping into the host address space
  - *clEnqueueMapBuffer*(), *clEnqueueMapImage*()
  - *clEnqueueUnmapMemObject*()

Program Objects (Section 5.4)

- Program objects encapsulate:
  - An associated context
  - Program source or binary
  - Latest successful program build, list of targeted devices, build options
  - Number of attached kernel objects
- Build process
  1. Create program object
     - clCreateProgramWithSource()
     - clCreateProgramWithBinary()
  2. Build program executable
     - Compile and link from source or binary for all devices or specific devices in the associated context
     - clBuildProgram()
     - Build options
       - Preprocessor
       - Math intrinsics (floating-point behavior)
       - Optimizations

Kernel Objects (Section 5.5)

- Kernel objects encapsulate
  - Specific kernel functions declared in a program
  - Argument values used for kernel execution
- Creating kernel objects
  - *clCreateKernel*() - creates a kernel object for a single function in a program
  - *clCreateKernelsInProgram*() - creates an object for all kernels in a program
- Setting arguments
  - *clSetKernelArg*(<kernel>, <argument index>)
  - Each argument data must be set for the kernel function
  - Argument values are copied and stored in the kernel object
- Kernel vs. program objects
  - Kernels are related to program execution
  - Programs are related to program source

Kernel Execution (Section 5.6)

- A command to execute a kernel must be enqueued to the command-queue
- ***clEnqueueNDRangeKernel***()
  - Data-parallel execution model
  - Describes the ***index space*** for kernel execution
  - Requires information on NDRange dimensions and work-group size
- ***clEnqueueTask***()
  - Task-parallel execution model (multiple queued tasks)
  - Kernel is executed on a single work-item
- ***clEnqueueNativeKernel***()
  - Task-parallel execution model
  - Executes a native C/C++ function not compiled using the OpenCL compiler
  - This mode does not use a kernel object so arguments must be passed in

Command-Queues and Synchronization

- **Command-queue execution**
  - Execution model signals when commands are complete or data is ready
  - Command-queue could be explicitly flushed to the device
  - Command-queues execute in-order or out-of-order
    - In-order - commands complete in the order queued and correct memory is consistent
    - Out-of-order - no guarantee when commands are executed or memory is consistent without synchronization

- **Synchronization**
  - Signals when commands are completed to the host or other commands in queue
  - Blocking calls
    - Commands that do not return until complete
    - clEnqueueReadBuffer() can be called as blocking and will block until complete
  - *Event objects*
    - Tracks execution status of a command
    - Some commands can be blocked until event objects signal a completion of previous command
      - clEnqueueNDRangeKernel() can take an event object as an argument and wait until a previous command (e.g., clEnqueueWriteBuffer) is complete
    - Profiling
  - Queue barriers - queued commands that can block command execution

Language for Compute Kernels

(Chapter 6)

- **OpenCL C Programming Language**
- **Derived from ISO C99**
  - A few restrictions: recursion, function pointers, functions in C99 standard headers …
  - Preprocessing directives defined by C99 are supported
- **Built-in Data Types**
  - Scalar and vector data types, Pointers
  - Data-type conversion functions: convert_type<_sat><_roundingmode>
  - Image types: image2d_t, image3d_t and sampler_t
- **Built-in Functions** — Required
  - work-item functions, math.h, read and write image
  - Relational, geometric functions, synchronization functions
- **Built-in Functions** — Optional
  - double precision, atomics to global and local memory
  - selection of rounding mode, writes to image3d_t surface

Language Highlights

- **Function qualifiers**
  - "__kernel" qualifier declares a function as a kernel
  - Kernels can call other kernel functions
- **Address space qualifiers**
  - __global, __local, __constant, __private
  - Pointer kernel arguments must be declared with an address space qualifier
- **Work-item functions**
  - Query work-item identifiers
  - get_work_dim()
  - get_global_id(), get_local_id(), get_group_id()
- **Image functions**
  - Images must be accessed through built-in functions
  - Reads/writes performed through sampler objects from host or defined in source
- **Synchronization functions**
  - Barriers - all work-items within a work-group must execute the barrier function before any work-item can continue
  - Memory fences - provides ordering between memory operations

Restrictions

- Pointers to functions are not allowed
- Pointers to pointers allowed within a kernel, but not as an argument
- Bit-fields are not supported
- Variable length arrays and structures are not supported
- Recursion is not supported
- Writes to a pointer of types less than 32-bit are not supported
- Double types are not supported, but reserved
- 3D Image writes are not supported
- Some restrictions are addressed through extensions

OpenCL Extended Features

Optional Extensions (Chapter 9)

- Extensions are optional features exposed through OpenCL
- The OpenCL working group has already approved many extensions that are supported by the OpenCL specification:
- Double precision floating-point types (Section 9.3)
- Built-in functions to support doubles
- Atomic functions (Section 9.5, 9.6, 9.7)
- 3D Image writes (Section 9.8)
- Byte addressable stores (write to pointers with types < 32-bits) (Section 9.9)
- Built-in functions to support half types (Section 9.10)

OpenGL Interoperability (Appendix B)

- **Both standards under one IP framework**
  - Enables very close collaborative design
- **Efficient, inter-API communication**
  - While still allowing both APIs to handle the types of workloads for which they were designed
- **OpenCL can efficiently share resources with OpenGL**
  - Textures, Buffer Objects and Renderbuffers
  - Data is shared, not copied
  - OpenCL objects are created *from* OpenGL objects
    - *clCreateFromGLBuffer*(), *clCreateFromGLTexture2D*(), *clCreateFromGLRenderbuffer*()
- **Applications can select compute device(s) to run OpenGL and OpenCL**
  - Efficient queuing of OpenCL and OpenGL commands into the hardware
  - Flexible scheduling and synchronization
- **Examples**
  - Vertex and image data generated with OpenCL and then rendered with OpenGL
  - Images rendered with OpenGL and post-processed with OpenCL kernels

Any Questions?

OpenCL Source Code Examples

Example 1: Vector Addition

- Compute c = a + b
  - a, b, and c are vectors of length *N*

- Basic OpenCL concepts
  - Simple kernel code
  - Basic context management
  - Memory allocation
  - Kernel invocation

- Example shows code "from kernel out"
  - Kernel code first, then the code that calls it

Simple Vector Addition in "C"

```
void sum(float A[], float B[], float C[], int n)
{
    for(int i=0; i<n; i++)
    {
        C[i] = A[i] + B[i];
    }
}
```

Simple Vector Addition in "C"

```
void sum(float A[], float B[], float C[], int n)
{
    for(int i=0; i<n; i++)
    {
        C[i] = A[i] + B[i];
    }
}
```

1D ***Index Space***

Kernel of Computation

Vector Addition Kernel

```
__kernel void vec_add (__global const float *a,
                       __global const float *b,
                       __global       float *c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
```

Spec

`__kernel`: Section 6.7.1
`__global`: Section 6.5.1
`get_global_id()`: Section 6.11.1
Data types: Section 6.1

Vector Addition Kernel

```
__kernel void vec_add (__global const float *a,
                       __global const float *b,
                       __global       float *c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
```

Original C code

```
void sum(float A[], float B[], float C[], int n)
{
    for(int i=0; i<n; i++)
    {
        C[i] = A[i] + B[i];
    }
}
```

Spec

`__kernel`: Section 6.7.1
`__global`: Section 6.5.1
`get_global_id()`: Section 6.11.1
Data types: Section 6.1

Host Code Outline

- Host program
  - Query compute devices
  - Create contexts

Platform Layer

  - Create memory objects associated to contexts
  - Compile and create kernel program objects
  - Issue commands to command-queue
  - Synchronization of commands
  - Clean up OpenCL resources

Runtime

VecAdd: Context, Devices, Queue

```
// create the OpenCL context on a GPU device
cl_context context = clCreateContextFromType(0,     // (must be 0)
                                             CL_DEVICE_TYPE_GPU,
                                             NULL,  // error callback
                                             NULL,  // user data
                                             NULL); // error code

// get the list of GPU devices associated with context
size_t cb;
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
cl_device_id *devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);

// create a command-queue
cl_cmd_queue cmd_queue = clCreateCommandQueue(context,
                                              devices[0],
                                              0,     // default options
                                              NULL); // error code
```

Spec

Contexts and context creation: Section 4.3
Command Queues: Section 5.1

VecAdd: Create Memory Objects

```
cl_mem memobjs[3];

// allocate input buffer memory objects
memobjs[0] = clCreateBuffer(context,
                            CL_MEM_READ_ONLY |    // flags
                            CL_MEM_COPY_HOST_PTR,
                            sizeof(cl_float)*n,   // size
                            srcA,                 // host pointer
                            NULL);                // error code

memobjs[1] = clCreateBuffer(context,
                            CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                            sizeof(cl_float)*n, srcB, NULL);

// allocate input buffer memory object
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                            sizeof(cl_float)*n, NULL, NULL);
```

Spec

Creating buffer objects: Section 5.2.1

VecAdd: Program and Kernel

```
// create the program
cl_program program = clCreateProgramWithSource(
                                        context,
                                        1,                  // string count
                                        &program_source, // program
  strings
                                        NULL,               // string lengths
                                        NULL);              // error code
// build the program
cl_int err = clBuildProgram(program,
                            0,      // num devices in device list
                            NULL,   // device list
                            NULL,   // options
                            NULL,   // notifier callback function ptr
                            NULL); // error code
// create the kernel
cl_kernel kernel = clCreateKernel(program, “vec_add”, NULL);
```

Spec

Creating program objects:	Section 5.4.1
Building program executables:	Section 5.4.2
Creating kernel objects:	Section 5.5.1

VecAdd: Set Kernel Arguments

```
// set "a" vector argument
err  = clSetKernelArg(kernel,
                      0,                    // argument index
                      (void *)&memobjs[0], // argument data
                      sizeof(cl_mem));      // argument data size

// set "b" vector argument
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
sizeof(cl_mem));

// set "c" vector argument
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
sizeof(cl_mem));
```

Spec

Setting kernel arguments: Section 5.5.2
Executing Kernels: Section 6.1
Reading, writing, and copying buffer objects: Section 5.2.2

VecAdd: Invoke Kernel, Read Result

```
size_t global_work_size[1] = n; // set work-item dimensions

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel,
                             1,          // Work dimensions
                             NULL,       // must be NULL (work offset)
                        global_work_size,
                             NULL,       // automatic local work size
                             0,          // no events to wait on
                             NULL,       // event list
                             NULL);      // event for this kernel
// read output array
err = clEnqueueReadBuffer(          context, memobjs[2],
                          CL_TRUE,              // blocking
                          0,                    // offset
                          n*sizeof(cl_float), // size
                          dst,                  // pointer
                          0, NULL, NULL);       // events
```

Spec

Setting kernel arguments: Section 5.5.2
Executing Kernels: Section 6.1
Reading, writing, and copying buffer objects: Section 5.2.2

Supplemental Information

- OpenCL Specification
  - http://www.khronos.org/opencl/

- Additional OpenCL examples
  - 3x3 median filter
  - Summed-Area Table (vertical phase)

(Simple) Median Filter

Compute my index

Handle edge cases

Load neighborhood

Sort along rows

Sort along cols

Sort diagonal

Output median

```
__kernel void medianfilter_rgba( __global uint *id, __global uint *od, int width, int h, int r )
{
    const int posx = get_global_id(0);
    const int posy = get_global_id(1);
    const int idx = posy*width + posx;

    if( posx == 0 || posy == 0 || posx == width-1 || posy == h-1 )
    {
        od[ idx ] = id[ idx ];
    } else {
        uint row00, row01, row02, row10, row11, row12, row20, row21, row22;
        row00 = id[ idx - 1 - width ];  row01 = id[ idx - width ];  row02 = id[ idx + 1 - width ];
        row10 = id[ idx - 1];           row11 = id[ idx ];          row12 = id[ idx + 1 ];
        row20 = id[ idx - 1 + width ];  row21 = id[ idx + width ];  row22 = id[ idx + 1 + width ];

        // sort the rows
        sort3( &(row00), &(row01), &(row02) );
        sort3( &(row10), &(row11), &(row12) );
        sort3( &(row20), &(row21), &(row22) );
        // sort the cols
        sort3( &(row00), &(row10), &(row20) );
        sort3( &(row01), &(row11), &(row21) );
        sort3( &(row02), &(row12), &(row22) );
        // sort the diagonal
        sort3( &(row00), &(row11), &(row22) );

        // median is the the middle value of the diagonal
        od[ idx ] = row11;
    }
}
```

Median Filter - Auxiliary Functions

Convert to luminance

```
float rgbToLum( uint a )
{
    return (0.3f*(a & 0xff) + 0.59*((a>>8) & 0xff) + 0.11*((a>>16) & 0xff));
}
```

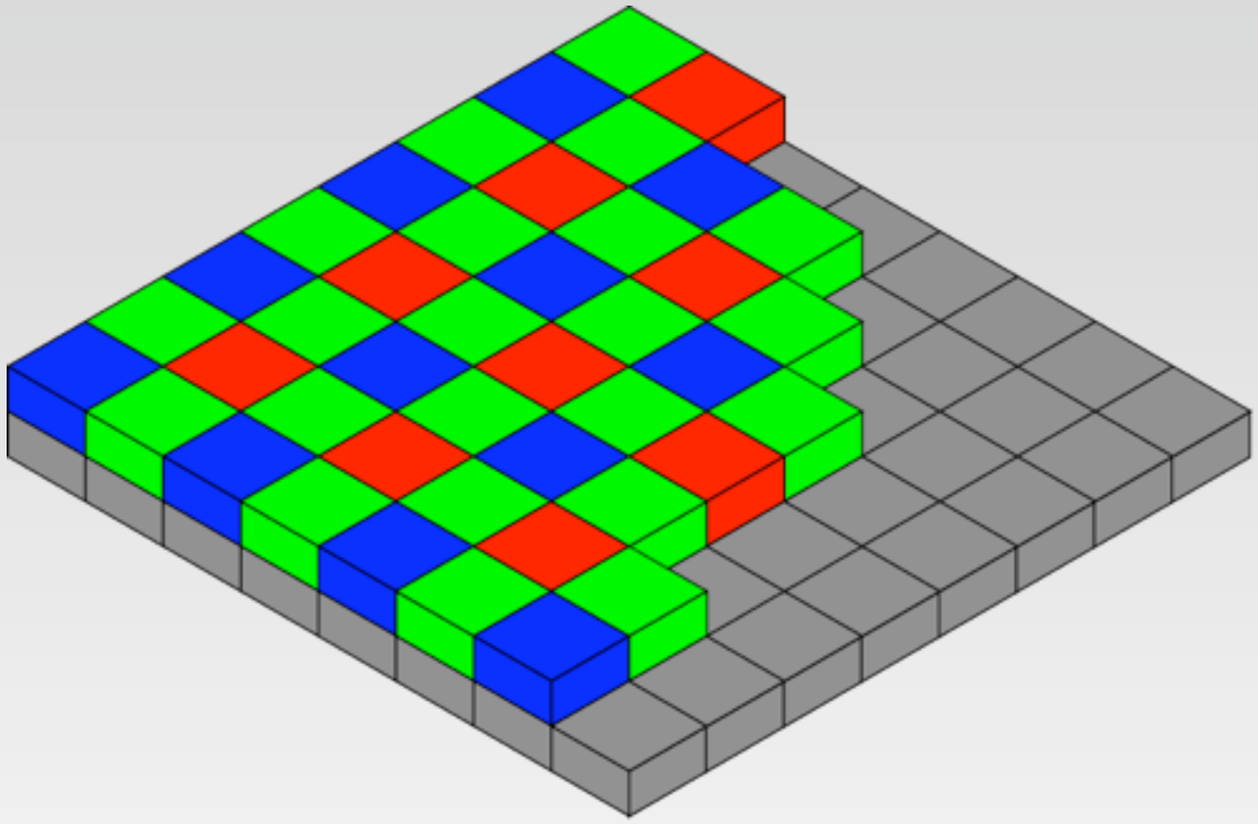
Swap A and B

```
void swap( uint *a, uint *b )
{
    uint tmp = *b;
    *b = *a;
    *a = tmp;
}
```

Comparison sort

```
void sort3( uint *a, uint *b, uint *c )
{
    if( rgbToLum( *a ) > rgbToLum( *b ) )
        swap( a, b );
    if( rgbToLum( *b ) > rgbToLum( *c ) )
        swap( b, c );
    if( rgbToLum( *a ) > rgbToLum( *b ) )
        swap( a, b );
}
```

Bayer Pattern



- Cameras typically sense only a single color at each ‘pixel’
  - Patterns can vary
- RGB images computed by interpolating nearby samples
- “BGGR” pattern shown
  - Emphasis on green chosen to mimic eye’s sensitivity to green light

*Image from http://en.wikipedia.org/wiki/File:Bayer_pattern_on_sensor.svg

Simple Bayer-Pattern Interpolation

- *Nd-range = <image width-1, image height-1>*

```
// Simple kernel to convert BGGR bayer pattern data to RGB
__kernel void convertBayerImage(
    __global uchar4 *output,
    __global const uchar *input,
    const int width )
{
    const int posx = get_global_id(0);
    const int posy = get_global_id(1);

    uchar in00 = input[ posy*width + posx ];
    uchar in10 = input[ posy*width + posx + 1 ] ;
    uchar in01 = input[ (posy+1)*width + posx ] ;
    uchar in11 = input[ (posy+1)*width + posx + 1 ] ;
```

Simple Bayer-Pattern Interpolation

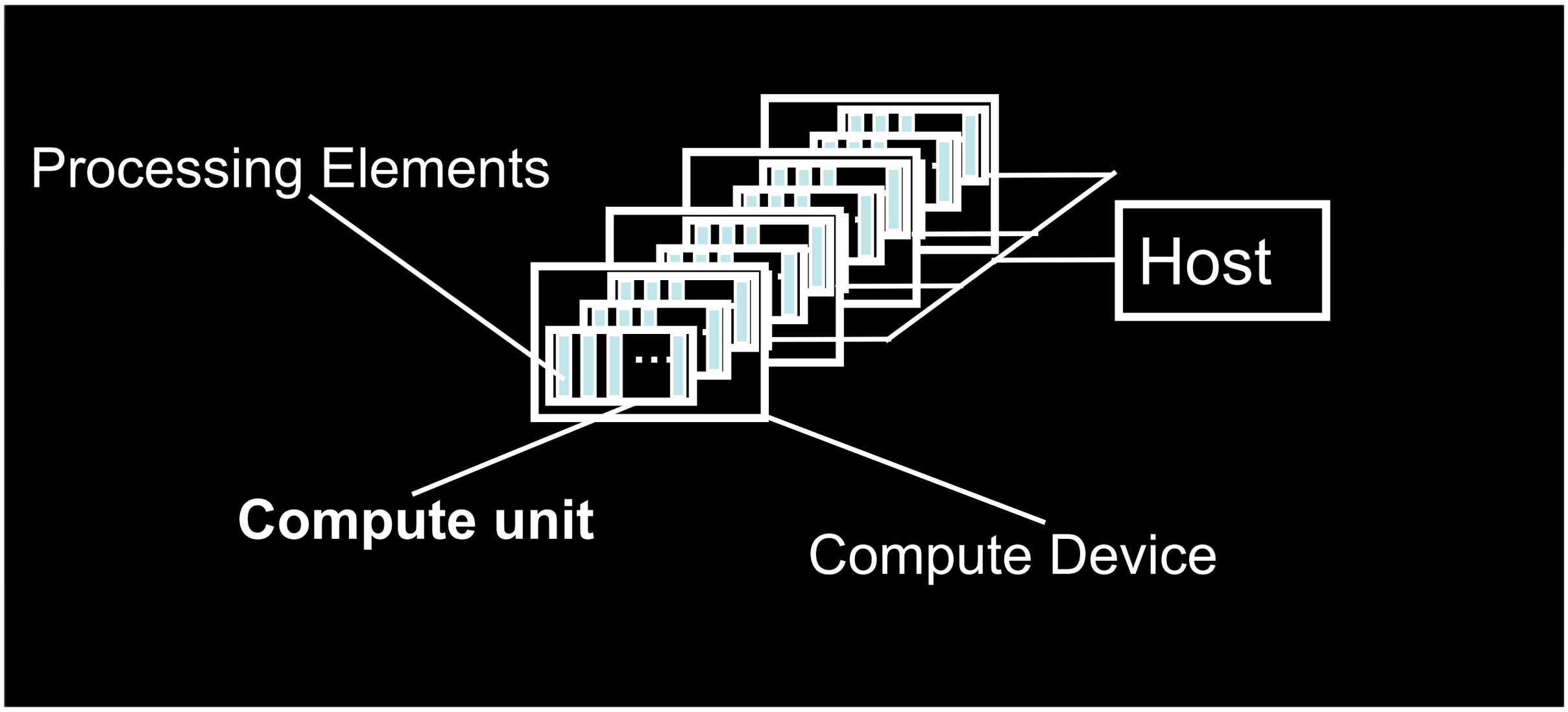
```
// permute the loaded values based on position in 2x2 quad
int xmod = posx%2;
int ymod = posy%2;

uchar imgB, imgG0, imgG1, imgR;
if( xmod == 0 && ymod == 0 )
{
    imgB = in00; imgG0 = in10; imgG1 = in01; imgR = in11;
} else if( xmod == 1 && ymod == 0 ) {
    imgG0 = in00; imgB = in10; imgR = in01; imgG1 = in11;
} else if( xmod == 0 && ymod == 1 ) {
    imgG0 = in00; imgR = in10; imgB = in01; imgG1 = in11;
} else {
    imgR = in00; imgG0 = in10; imgG1 = in01; imgB = in11;
}
// perform simple interpolation
output[ posy*width + posx ] =
    (uchar4)( imgR, 0.5 * ( imgG0 + imgG1 ), imgB, 255 );
return;
}
```

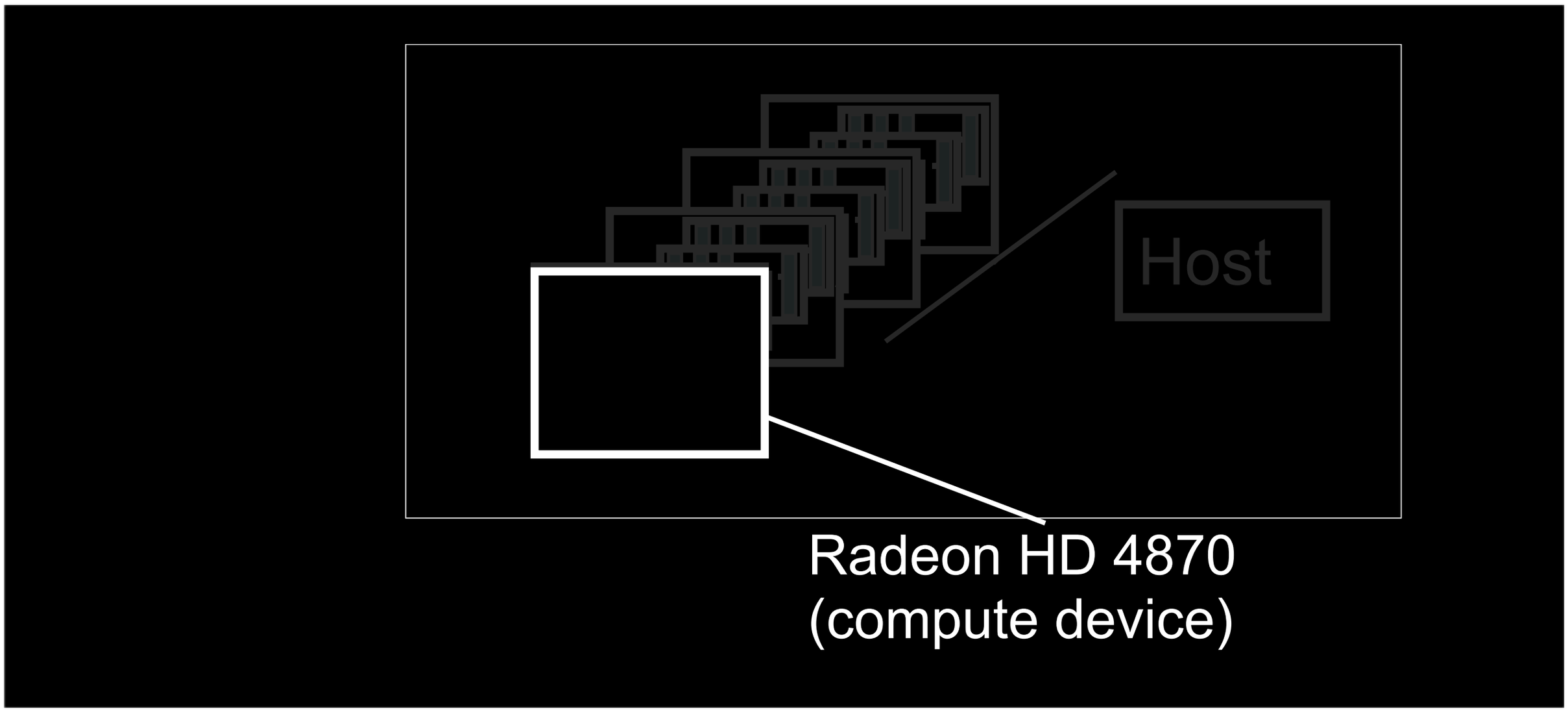
ATI Radeon™ 4000 Series

Technical Overview

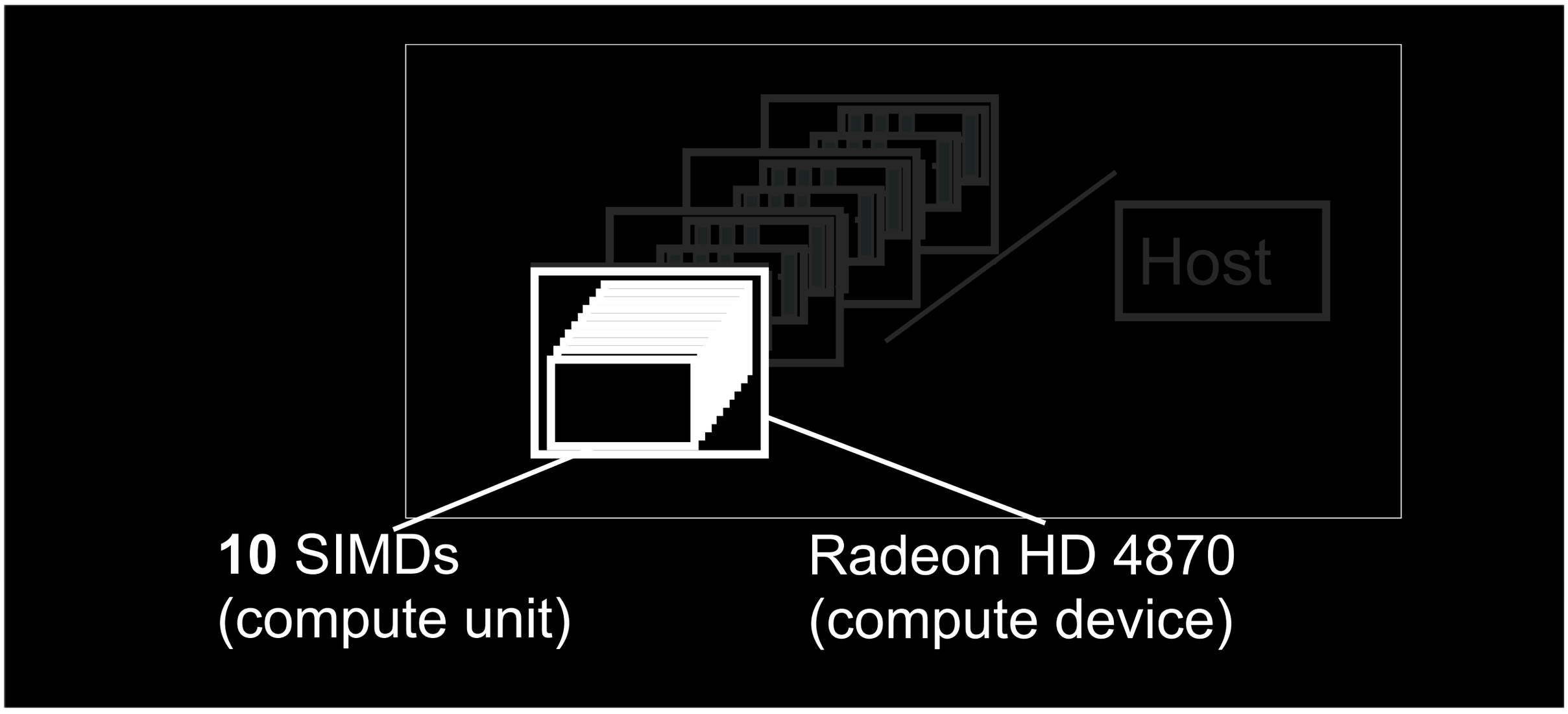
ATI Radeon HD 4870 : Logical View



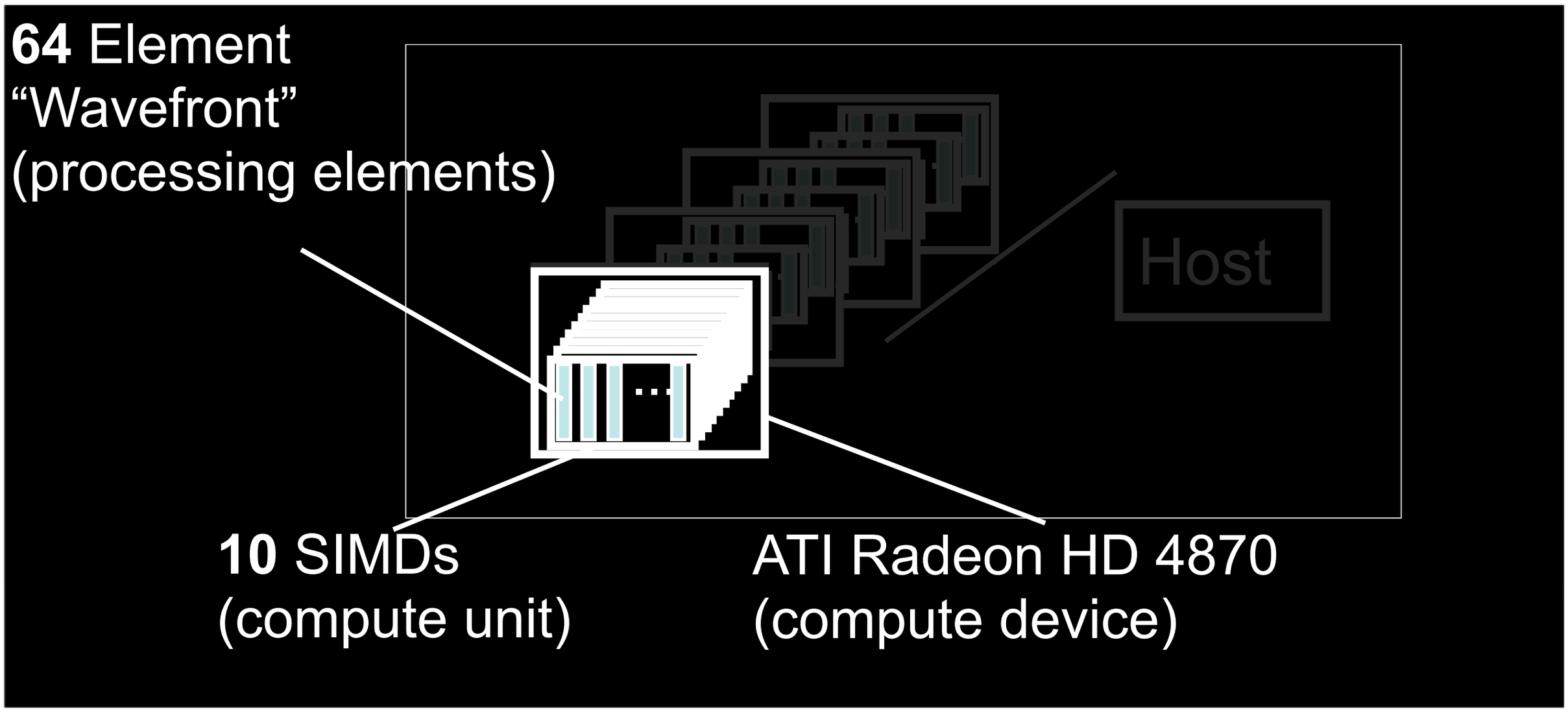
ATI Radeon HD 4870 : Logical View



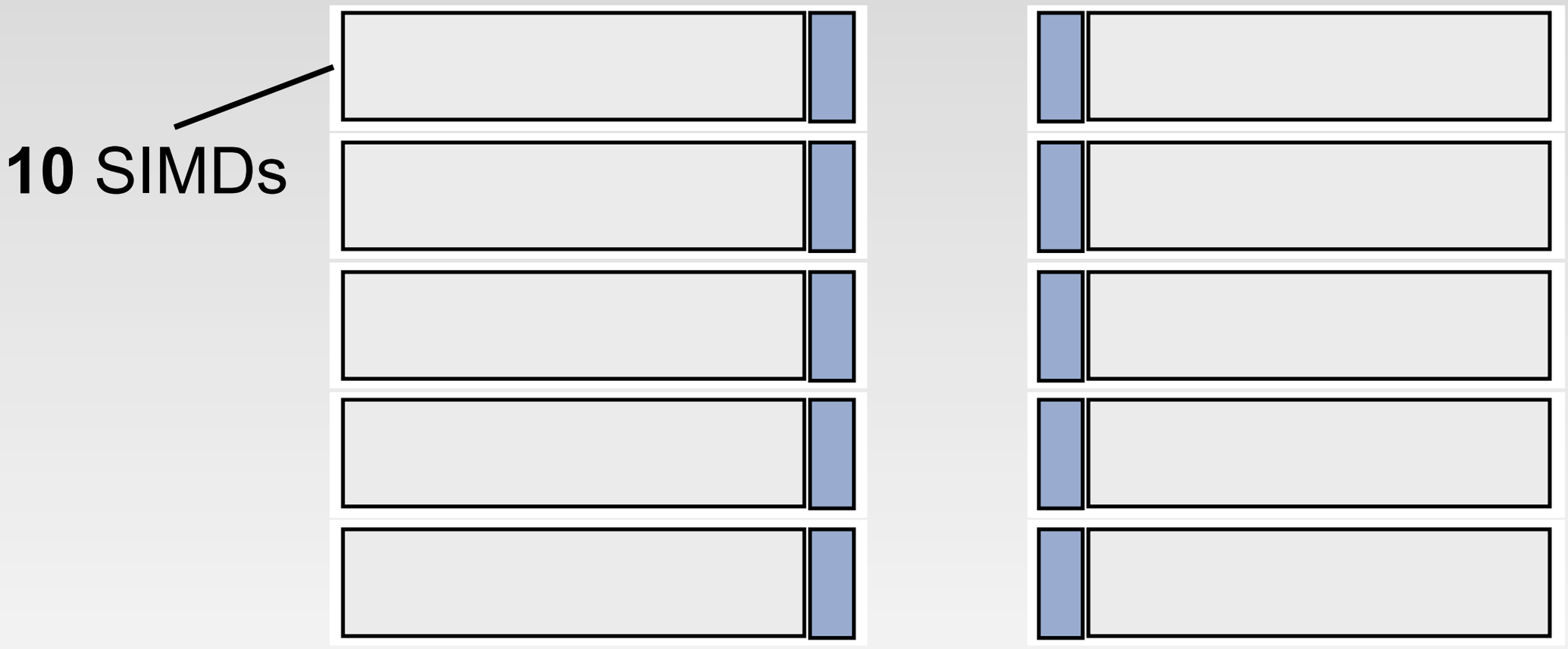
ATI Radeon HD 4870 : Logical View



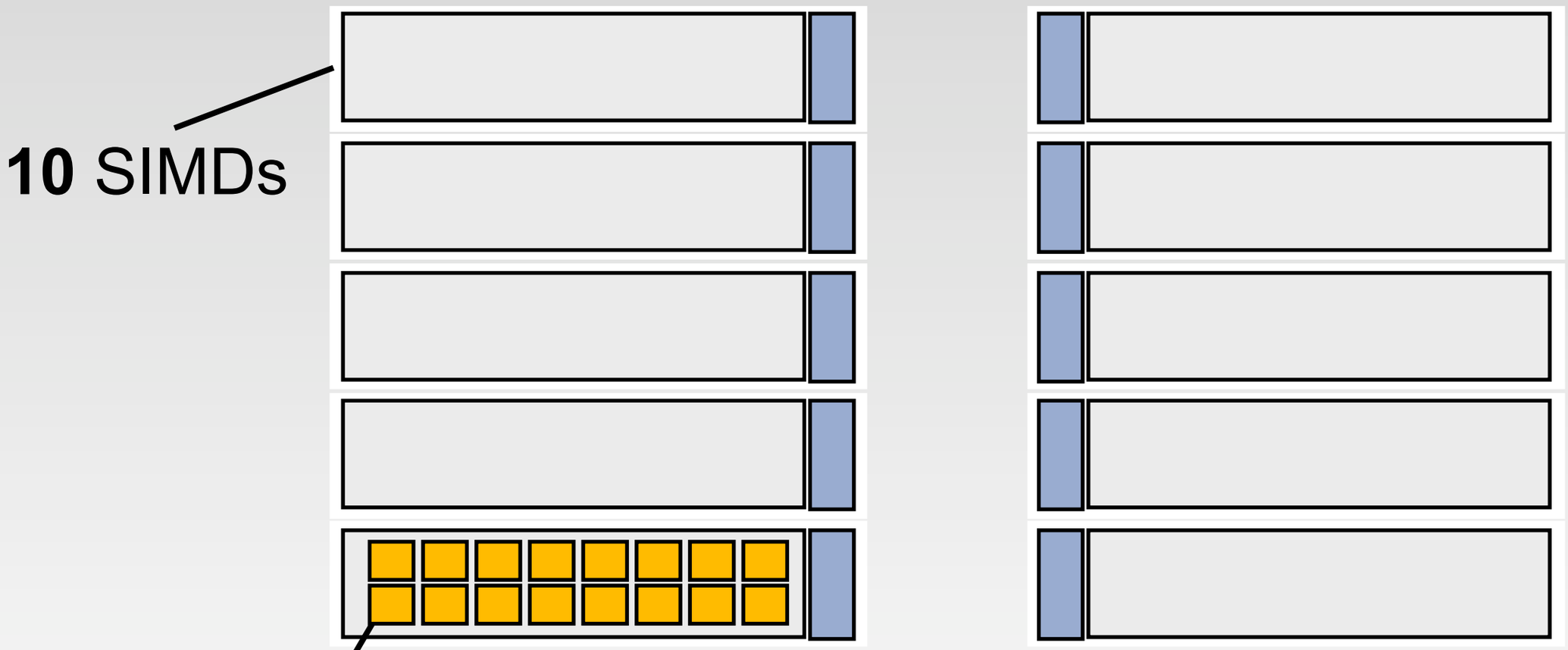
ATI Radeon HD 4870 : Logical View



ATI Radeon HD 4870 : *Reality*

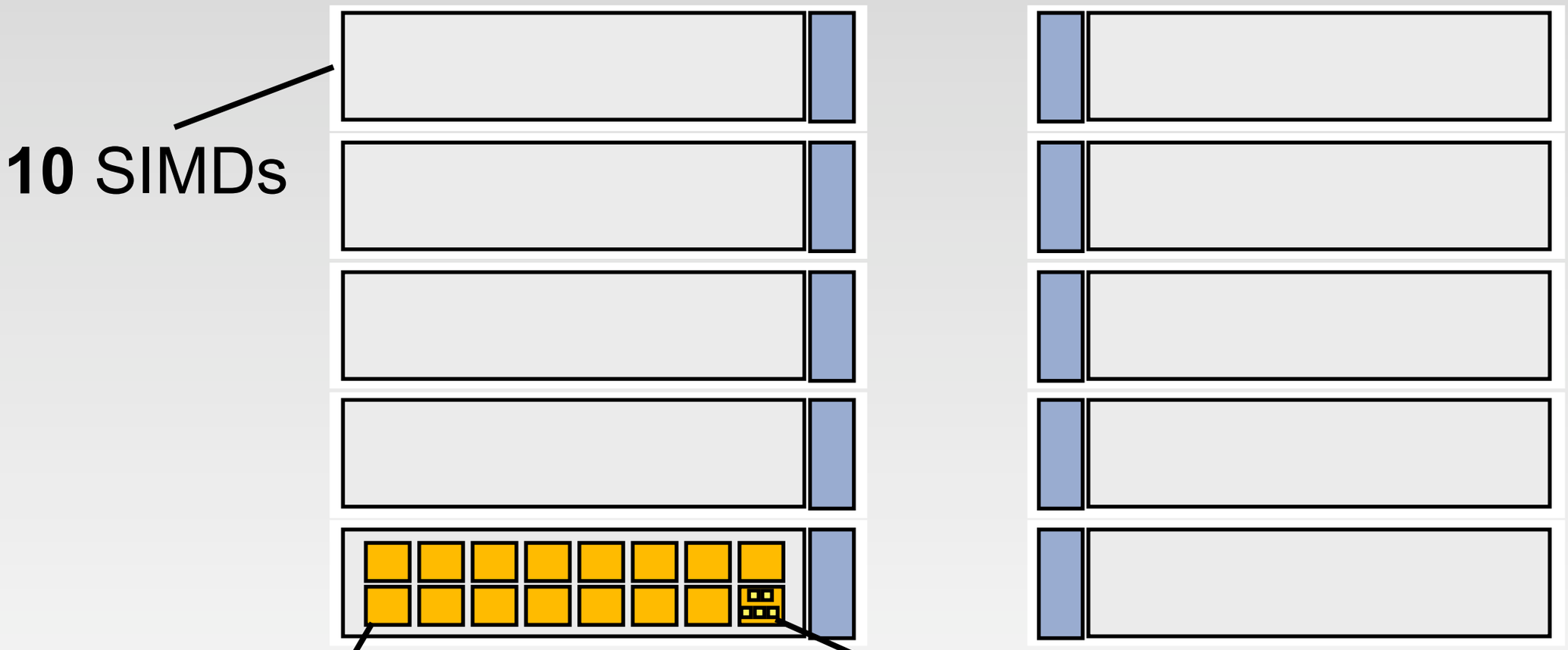


ATI Radeon HD 4870 : *Reality*

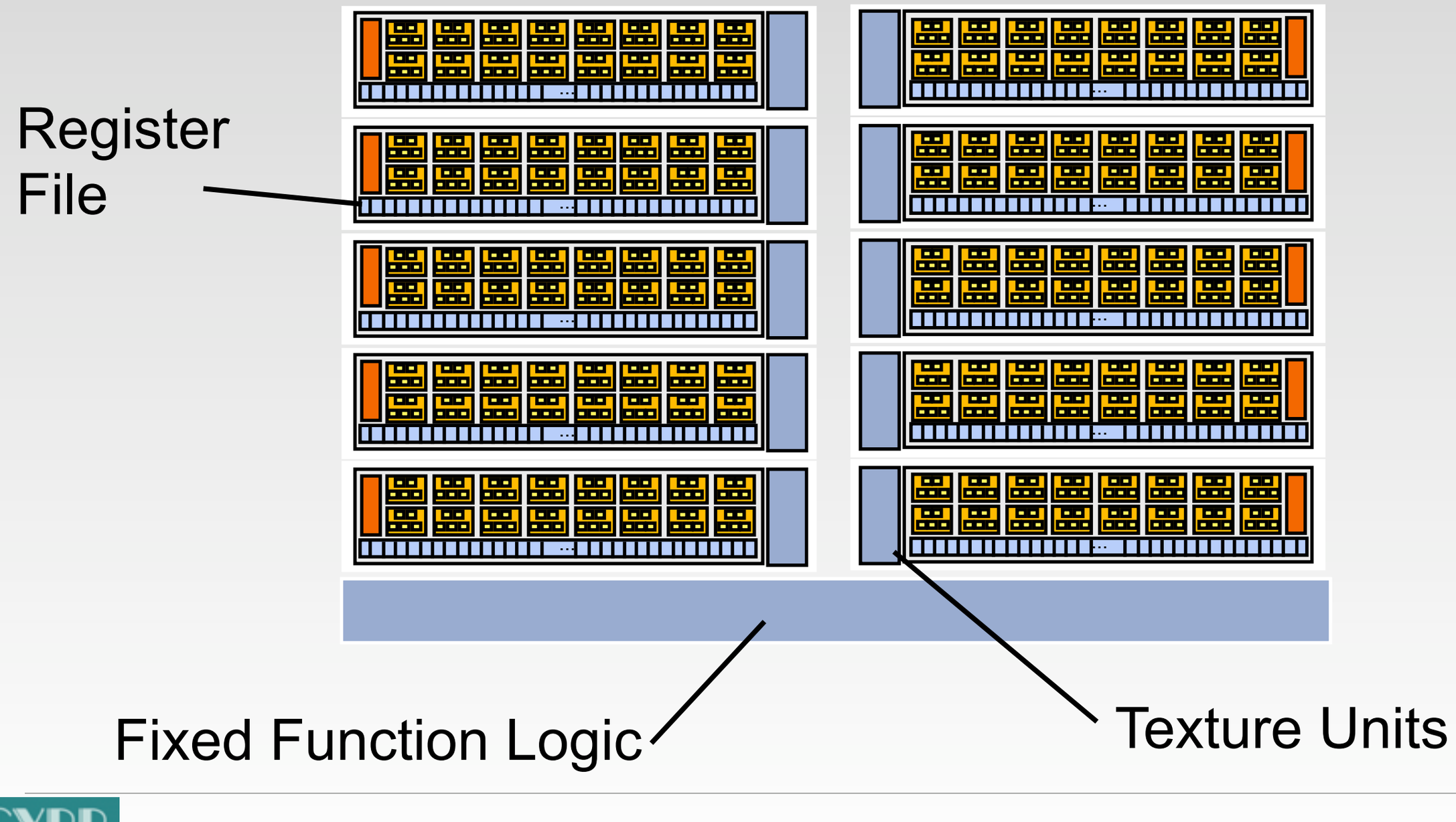


16 Processing “cores” per SIMD
(**64** Elements over 4 cycles)

ATI Radeon HD 4870 : *Reality*

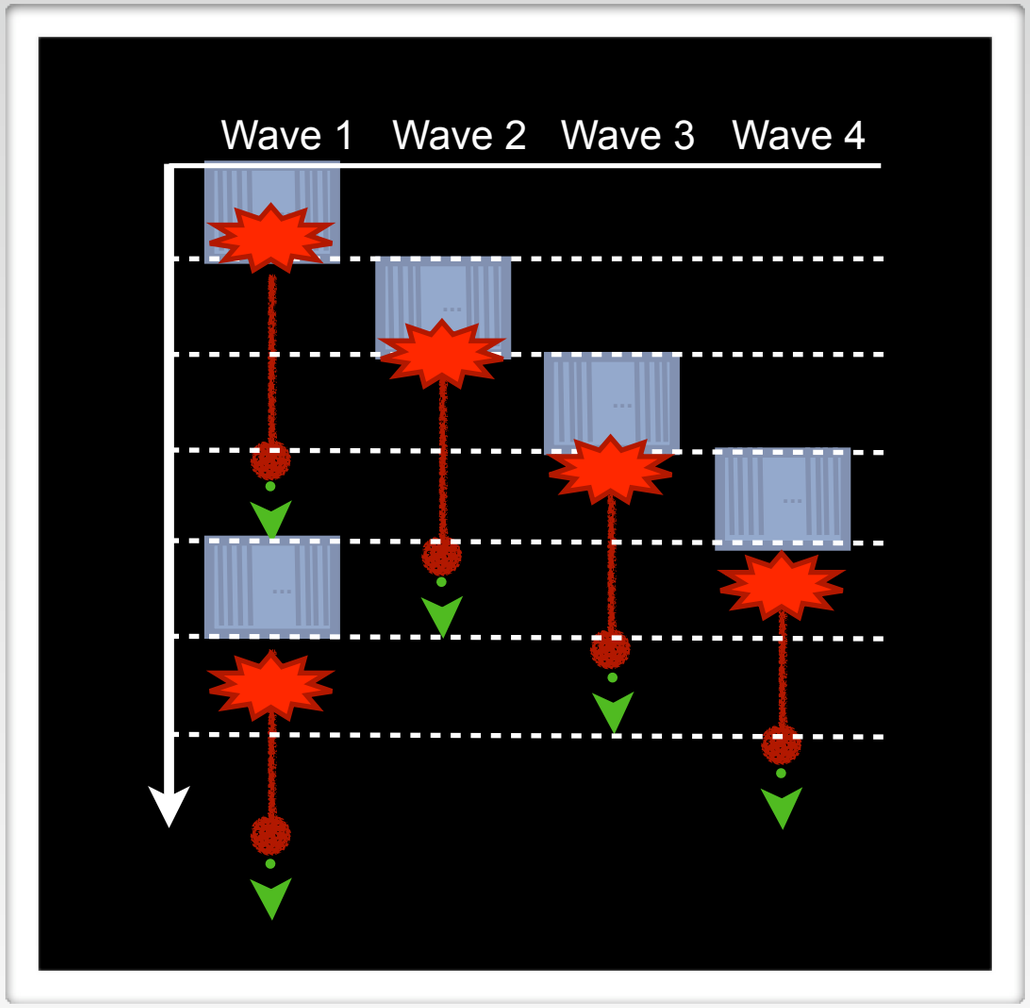


ATI Radeon HD 4870 : *Reality*



Latency Hiding

- GPUs eschew large caches for large register files
- Ideally launch more "work" than available ALUs
- Register file partitioned amongst active wavefronts
- Fast switching on long latency operations



“Spreadsheet Analysis”

- Valuable to estimate performance of kernels
  - Helps identify when something is “wrong”
  - AMD’s GPU Shader Analyzer / Stream Kernel Analyzer
- First-order bottlenecks: ALU, TEX, or MEM
  - $T_{kernel} = \max(\text{ALU}, \text{TEX}, \text{MEM})$

$$T_{alu} = \#elements * \#ALU / (10*16) / 750\ Mhz$$

global_size

VLIW instructions

SIMDs

“cores” per SIMD

Engine Clock

Practical Implications

- Workgroup size should be a multiple of 64
  - Remember: *Wavefront* is 64 elements
  - Smaller workgroups ➙ SIMDs will be underutilized

- SIMDs operate on pairs of wavefronts

Minimum Global Size

- **10** SIMDs * **2** waves * **64** elements = **1280** elements
  - Minimum global size to utilize GPU with one kernel
  - Does not allow for any latency hiding!
- For **minimum** latency hiding: **2560 elements**

Register Usage

- Recall GPUs hide latency by switching between large number of wavefronts
- Register usage determines maximum number of wavefronts in flight
- More wavefronts ➙ better latency hiding
- Fewer wavefronts ➙ worse latency hiding
- Long runs of ALU instructions can compensate for low number of wavefronts

Kernel Guidelines

- Prefer **int4 / float4** when possible
  - Processor "cores" are 5-wide VLIW machines
  - Memory system prefers 128-bit load/stores
- Consider access patterns - e.g. access along rows
- AMD GPUs have large register files
  - Perform "more work per element"

More Work per Work-item

- Prefer read/write 128-bit values
- Compute more than one output per work-item
- Better Algorithm (further optimizations possible):
  1. Load neighborhood 8x3 via six 128-bit loads
  2. Sort pixels for each of four pixels
  3. Output median values via 128-bit write
- 20% faster than simple method on ATI Radeon HD 4870

More Work per Work-item

```
__kernel void medianfilter_x4( __global uint *id, __global uint *od, int width, int h, int r )
{
    const int posx = get_global_id(0); // global width is 1/4 image width
    const int posy = get_global_id(1); // global height is image height
    const int width_d4 = width >> 2;   // divide width by 4
    const int idx_4 =  posy*(width_d4) + posx;

    uint4 left0, right0, left1, right1, left2, right2, output;
    // ignoring edge cases for simplicity
    left0  = ((__global uint4*)id)[ idx_4 - width_d4 ];
    right0 = ((__global uint4*)id)[ idx_4 - width_d4 + 1];

    left1  = ((__global uint4*)id)[ idx_4 ];
    right1 = ((__global uint4*)id)[ idx_4 + 1];

    left2  = ((__global uint4*)id)[ idx_4 + width_d4 ];
    right2 = ((__global uint4*)id)[ idx_4 + width_d4 + 1];

    // now compute four median values
    output.x = find_median( left0.x, left0.y, left0.z,
                            left1.x, left1.y, left1.z, left2.x, left2.y, left2.z );
    output.y = find_median( left0.y, left0.z, left0.w,
                            left1.y, left1.z, left1.w, left2.y, left2.z, left2.w );
    output.z = find_median( left0.z, left0.w, right0.x,
                            left1.z, left1.w, right1.x, left2.z, left2.w, right2.x );
    output.w = find_median( left0.w, right0.x, right0.y,
                            left1.w, right1.x, right1.y, left2.w, right2.x, right2.y );

    ((__global uint4*)od)[ idx_4 ] = output;
}
```

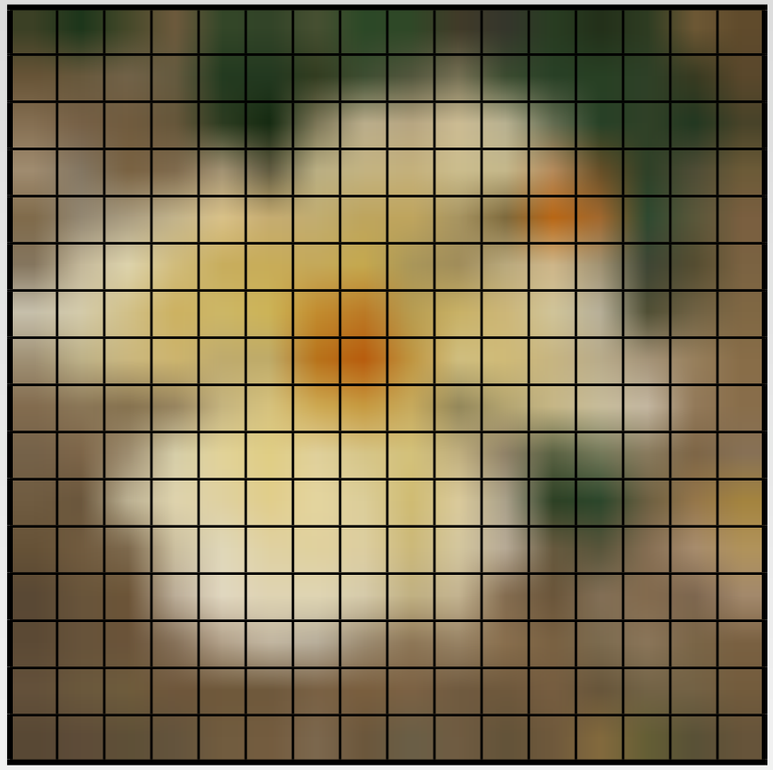
Memory Accesses

- Summed-area tables - a.k.a. “integral images”
  - 2D Scan of image
- Useful in computer vision and graphics
- OpenGL way - recursive doubling
  - Compute height 1D horizontal SATs
  - Compute width 1D vertical SATs
  - 2D texture *really* helps here.
- OpenCL way (for sufficiently large images)
  - Just perform n-sequential scans in parallel
  - For smaller images - need to block image to get “enough” threads in flight

Summed-Area Table Generation

Vertical Phase

Image (*logical*)

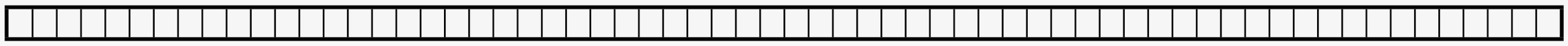


```
__kernel void verticalSAT( __global float *out,
                           __global float *in,
                                      int width  )
{
    const int idx = get_global_id( 0 );

    int i, index = idx;
    float sum = 0.0;

    for( i = 0;i < height;i++ )
    {
        sum = sum + in[index];
        out[index] = sum;
        index = index + width;
    }
}
```

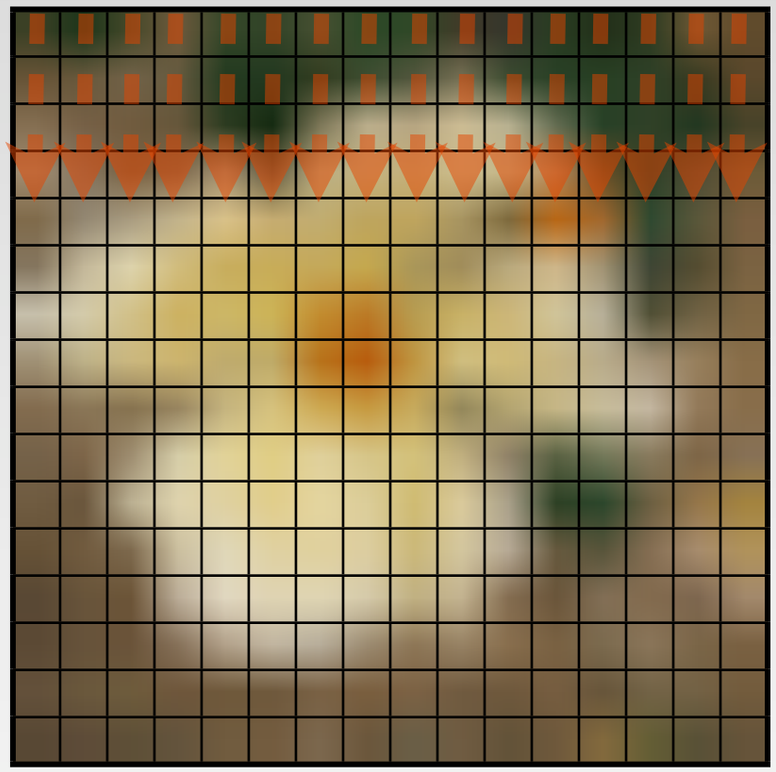
Memory (*reality*)



Summed-Area Table Generation

Vertical Phase

Image (*logical*)

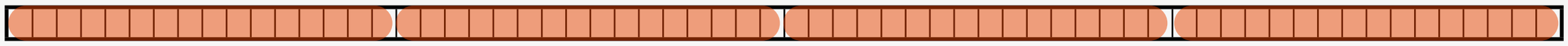


```
__kernel void verticalSAT( __global float *out,
                           __global float *in,
                                      int width  )
{
    const int idx = get_global_id( 0 );

    int i, index = idx;
    float sum = 0.0;

    for( i = 0;i < height;i++ )
    {
        sum = sum + in[index];
        out[index] = sum;
        index = index + width;
    }
}
```

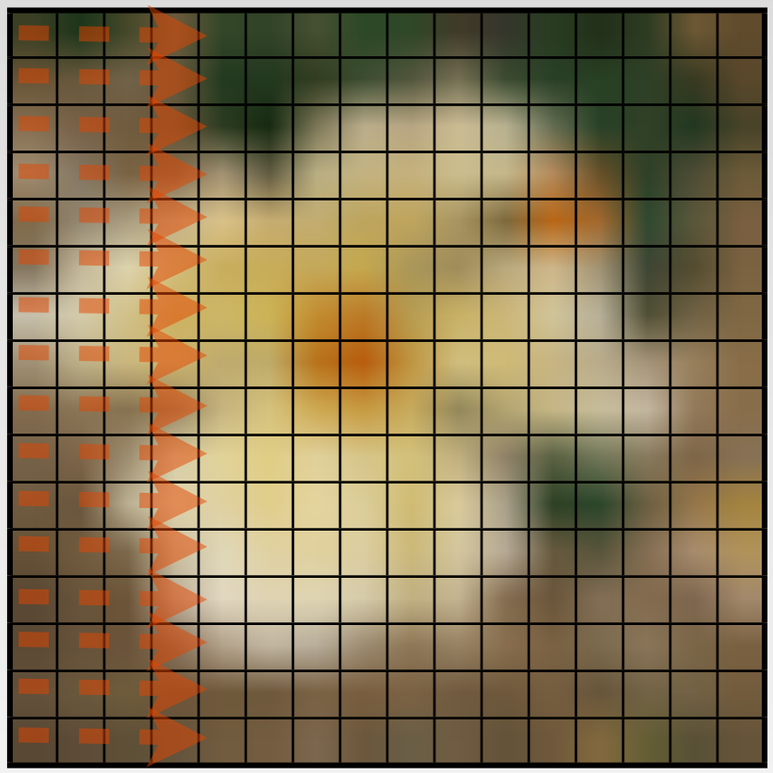
Memory (*reality*)



Summed-Area Table Generation

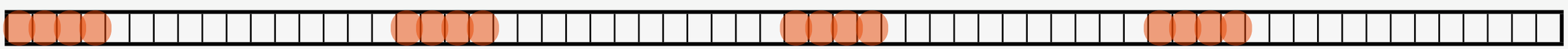
Horizontal Phase

Image (*logical*)



BAD, BAD, BAD!

Memory (*reality*)



2D SAT

1. Compute vertical SAT
2. Compute transpose
3. Compute vertical SAT
4. Compute transpose

- Could combine SAT and transpose.

Optimization Summary

- Optimization is a balancing act
  - Almost every rule has an exception
- How important is the last 20%, 10%, 5%?
- Things to consider
  - Register usage / number of Wavefronts in flight
  - ALU to memory access ratio
    - Sometimes better re-compute something
  - Workgroup size a multiple of 64
  - Global size at least 2560 for a single kernel

Further Reading & A "Shout Out"

- Example Image processing in practice
- Exclusive anti-aliasing mode available on AMD's GPUs
  - CFAA - custom filter anti-aliasing
  - Upcoming "High Performance Graphics" paper describes implementation

- "High Performance Graphics" - combination of **Graphics Hardware** and the **Symposium on Interactive Ray Tracing**
  - Co-located with SIGGRAPH
  - 1/3 "GPGPU", 1/3 rasterization, graphics hardware, 1/3 ray tracing

Trademark Attribution

AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Other names used in this presentation are for identification purposes only and may be trademarks of their respective owners.

Computer Vision on Larrabee

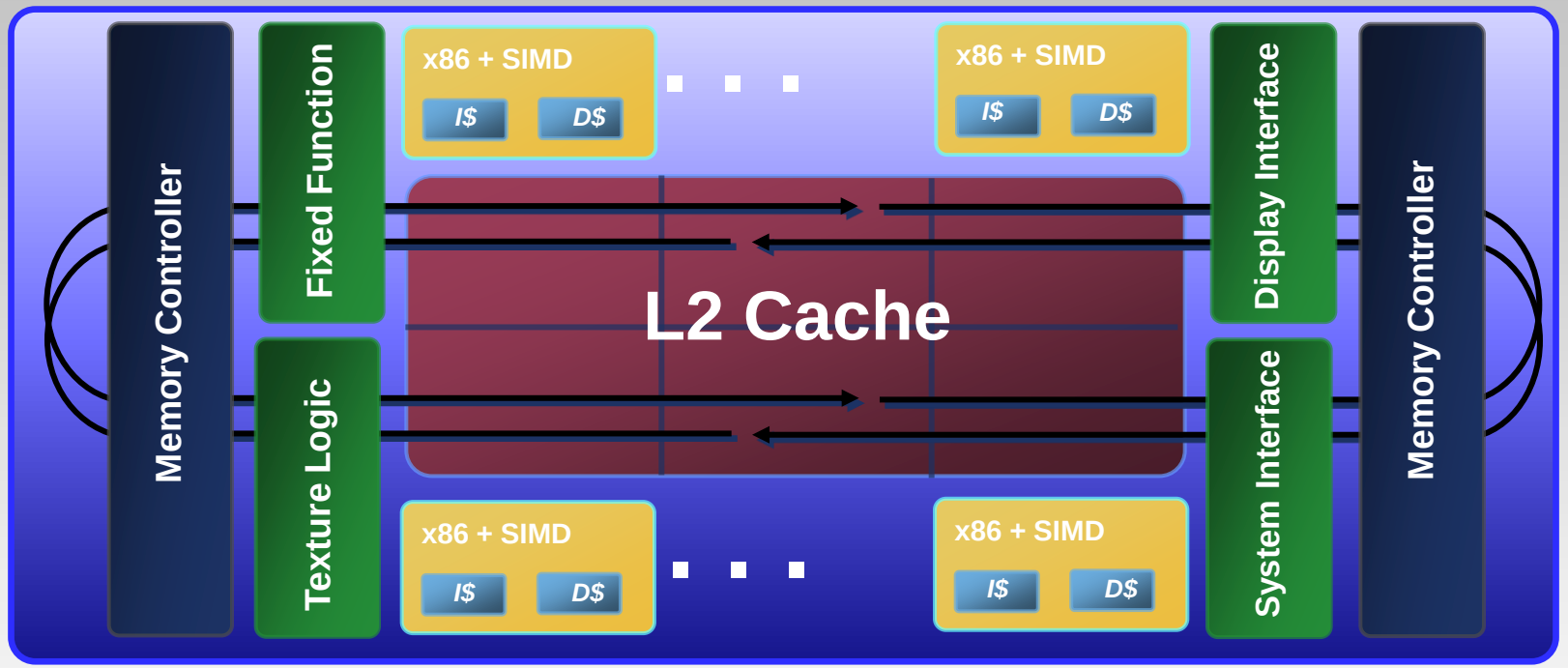
Oleg Maslov,
Konstantin Rodyushkin,
Intel

Outline

- **Short tour of**
  - Larrabee architecture
  - Low-level programming
  - Paradigms for parallel programming on Larrabee
- **Case studies**
  - Efficient parallel execution of CV kernels
  - Scalability results on some of them

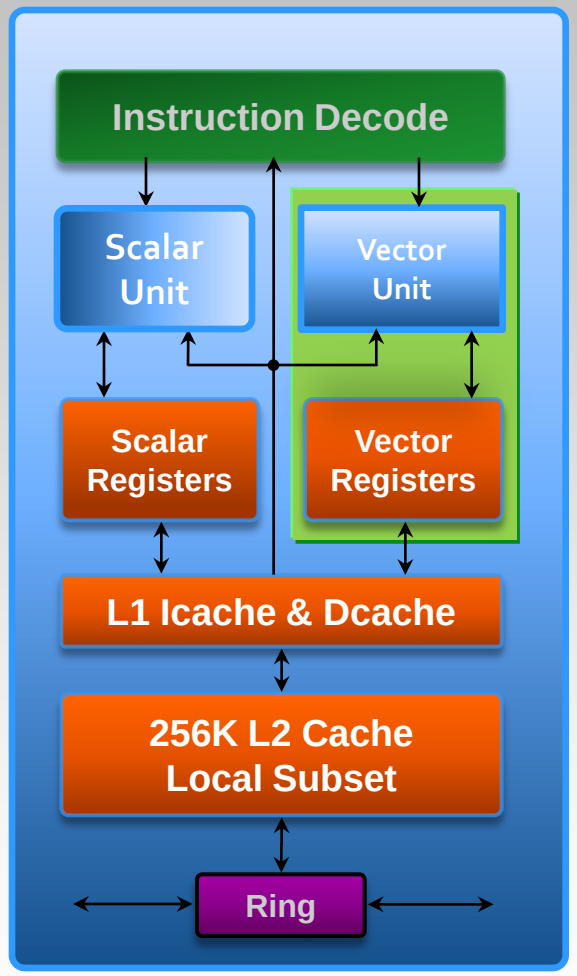
Larrabee architecture

Obligatory block diagram



- Lots of x86 cores with 16-wide SIMD
- Fully coherent caches
- Wide ring bus
- Fixed-function texture hardware

Larrabee core



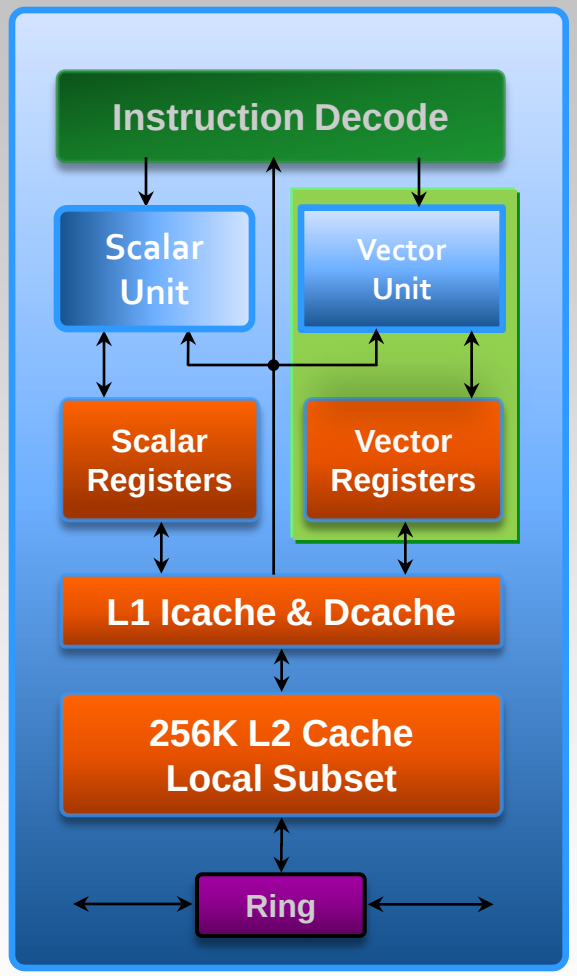
Larrabee based on x86 ISA

- All of the left "scalar" half
- Four threads per core
- No surprises, except that there's LOTS of cores and threads

New right-hand vector unit

- Larrabee New Instructions
- 512-bit SIMD vector unit
- 32 vector registers
- Pipelined one-per-clock throughput
- Dual issue with scalar instructions

Larrabee core



Fully coherent L1 and L2 caches

Short in-order pipeline

- No latency on scalar ops, low latency on vector
- Cheap branch mispredict

Connected to fast bidirectional ring

- L2 caches can share data with each other

Vector Unit Data Types

512-bit vector register

Bits	480	448	416	384	352	320	288	256	224	192	160	128	96	64	32	0
float	15.0f	14.0f	13.0f	12.0f	11.0f	10.0f	9.0f	8.0f	7.0f	6.0f	5.0f	4.0f	3.0f	2.0f	1.0f	0.0f
int	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
double	7.0		6.0		5.0		4.0		3.0		2.0		1.0		0.0	
int64	7		6		5		4		3		2		1		0	

16-wide float32/int32

8-wide float64 vector

32 vector registers v0-v31

Larrabee New Instructions

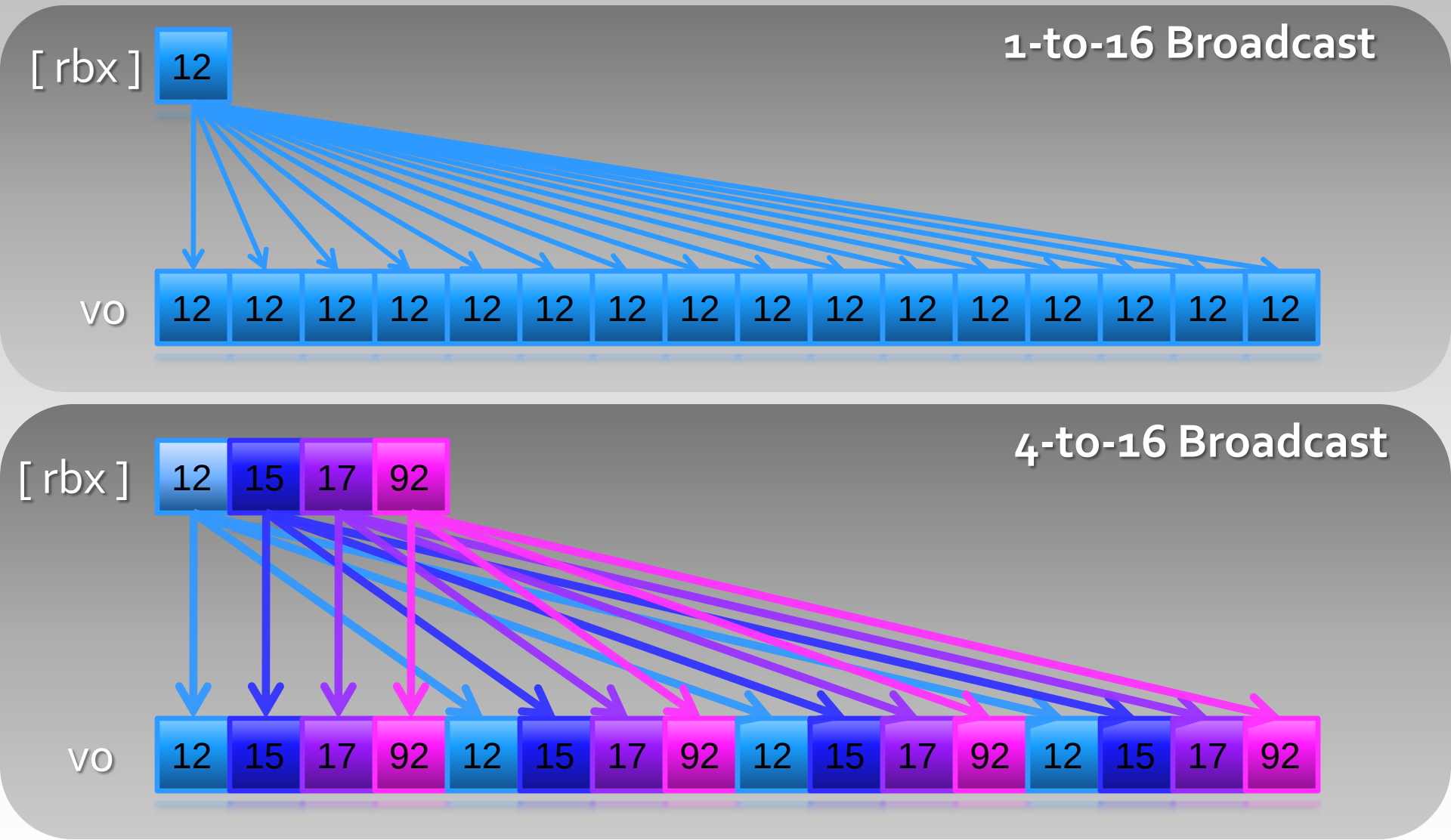
All SIMD math at 32- or 64-bit

- Ternary ops
- Multiply-add
- One source from memory

Broadcast/swizzle/format conversion

- {s,u}int{8,16}, float16,etc - allows more efficient use of caches and memory
- Almost free conversion

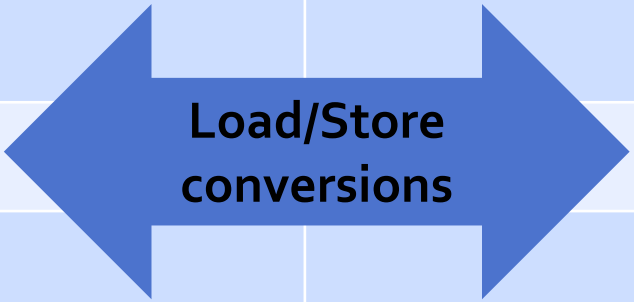
Broadcast example



Larrabee Format Conversions

Memory

float16	float32
uint8	float32
sint8	float32
uint16	float32
sint16	float32
uint8	uint32
sint8	sint32
uint16	uint32
sint16	sint32



And more …

Larrabee

Predication

- **Eight 16-bit mask registers k0-k7**
- **Every instruction can take a mask**
- **Act as write masks – bit=0 preserves dest**

  vaddps v1{k6}, v2, v3

  - Bits in k6 enable/disable writes to v1
  - Preserves existing register contents in bit=0 lanes
- **Memory stores also take a write mask**

Predication

- **Predication allows per-lane conditional flow**

- **Vector compare does 16 parallel compares**
  - Writes results into a write mask
  - Mask can be used to protect some of the 16 elements from being changed by instructions

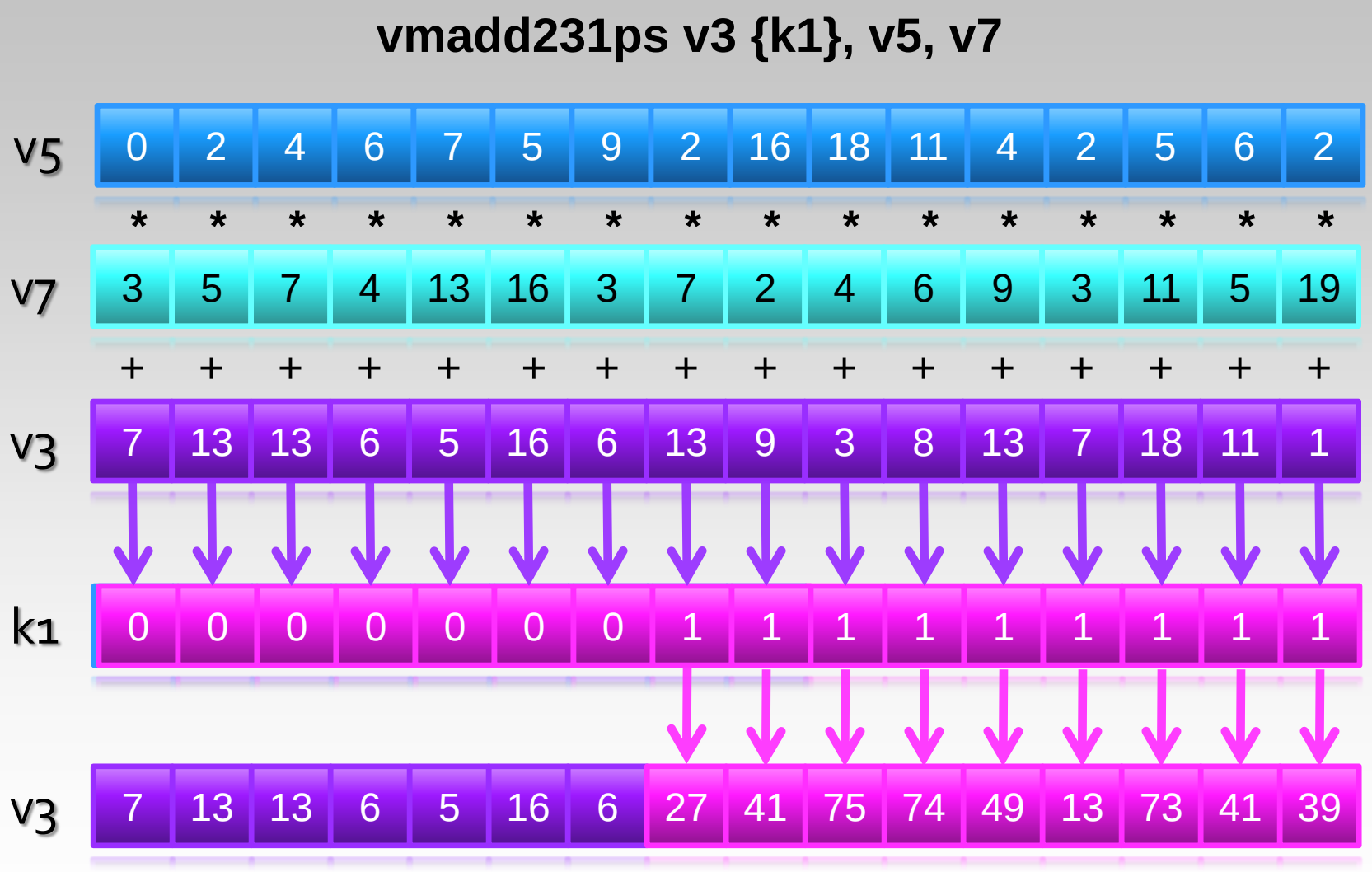
- **Simple predication example:**

  ;if (v5<v6) {v1 += v3;}

  vcmppi_lt k7, v5, v6

  vaddpi v1{k7}, v1, v3

Multiply-Add with Predication Example



Gather/Scatter

- **Important part of a wide vector ISA**
  - Structure-Of-Array(SOA) mode is difficult to get data into
  - Most data structures are Array-Of-Structures (AOS)
  - Natural format for indirections – pointer to each structure
- **Gather/scatter allows sparse read/write**
  - Gather gets data into the 16-wide SOA format in registers
  - Process data 16-wide
- **Scatter stores data back out into AOS**

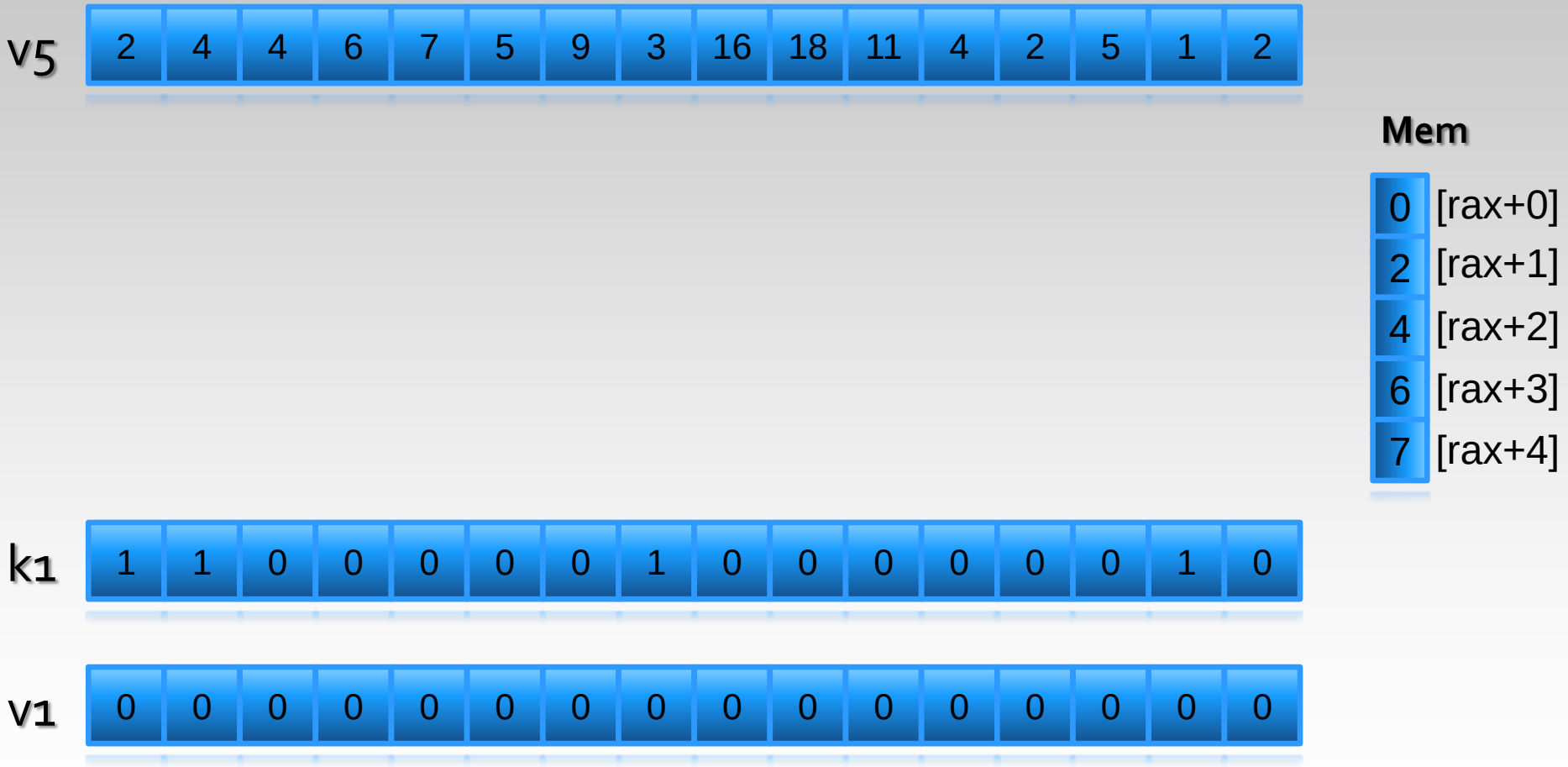
Gather Example

vgather v1{k1},[rax+v5]

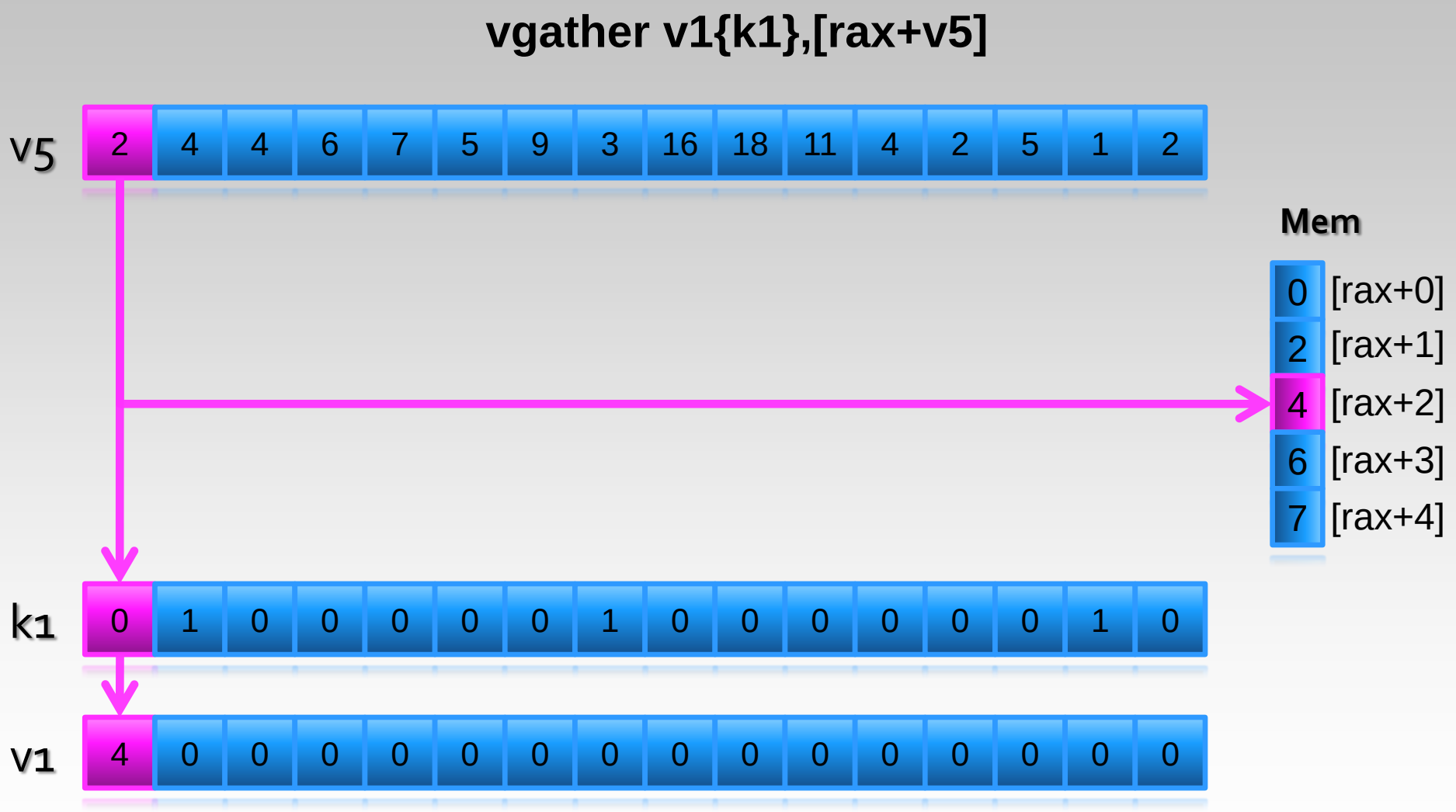
- **Gather is effectively 16 loads, one per lane**
- **As usual, a mask register (k1) disables some lanes**
  - Gather/scatter are special – they MUST take a mask
  - Afterwards, mask will be all-zeros
- **Vector of offsets (v5)**
  - Offsets may be optionally scaled by 2, 4 or 8 bytes
  - Added to a standard x86 base pointer (rax)
  - Offsets can point anywhere in memory
  - Multiple offsets can point to the same place

Gather showcase

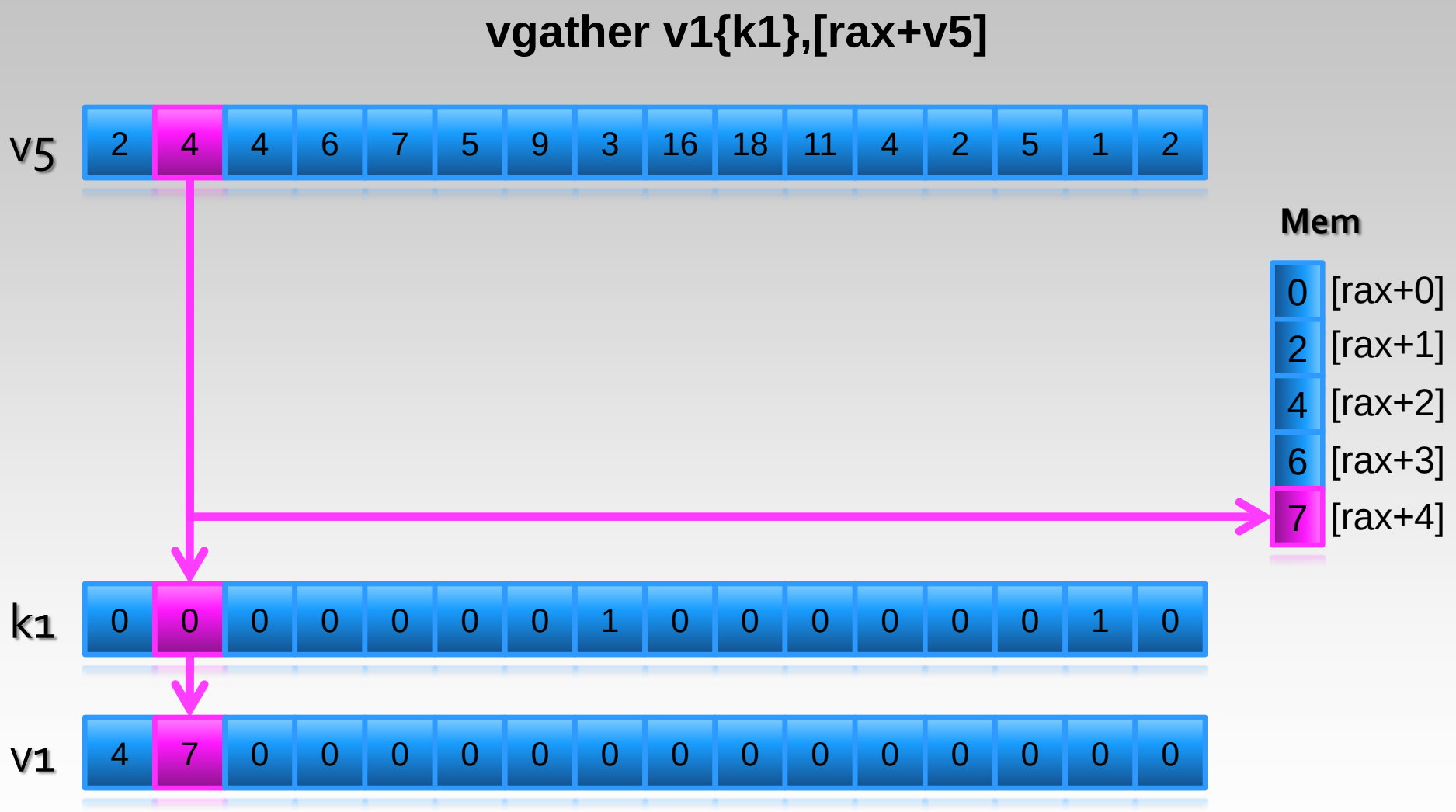
vgather v1{k1},[rax+v5]



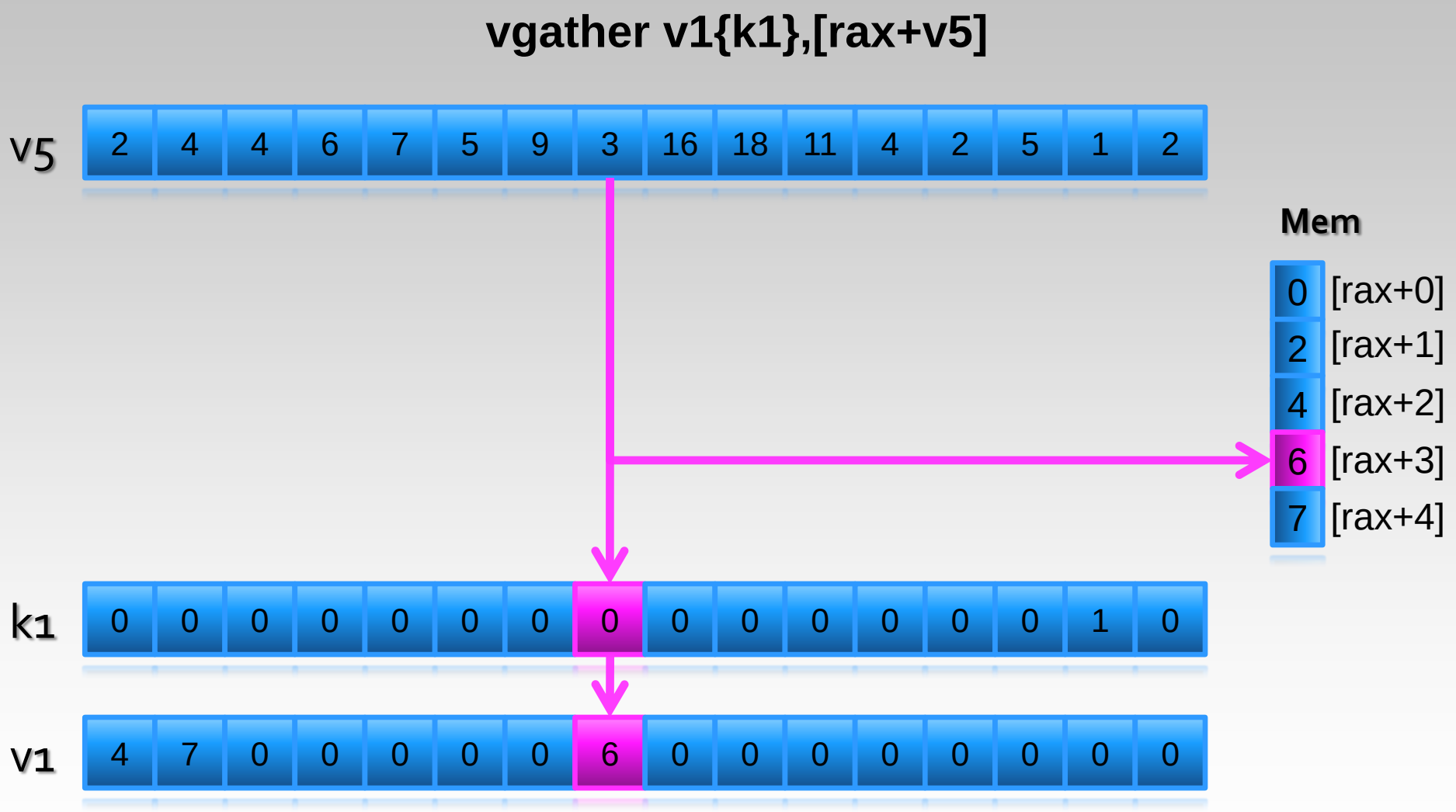
Gather showcase



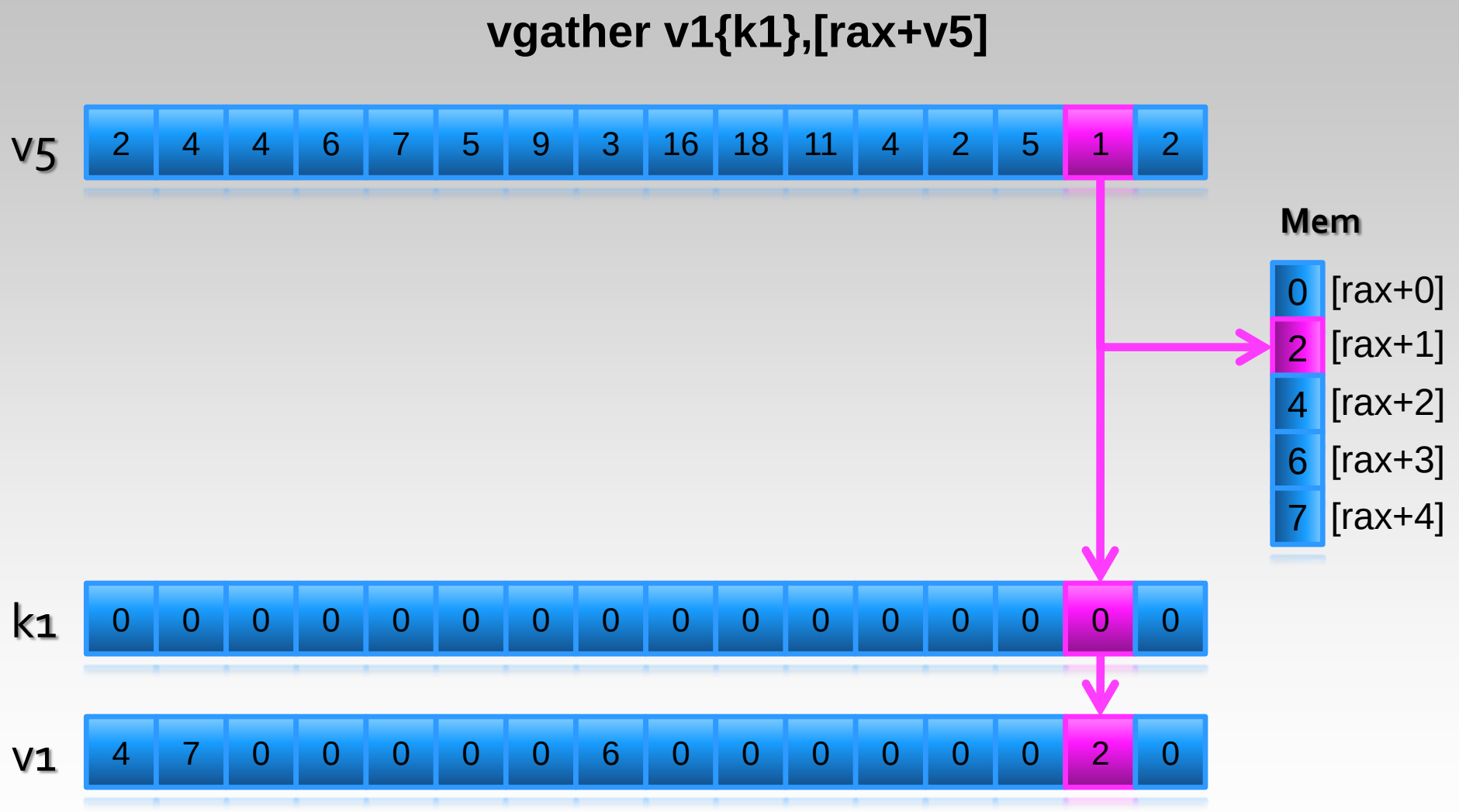
Gather showcase



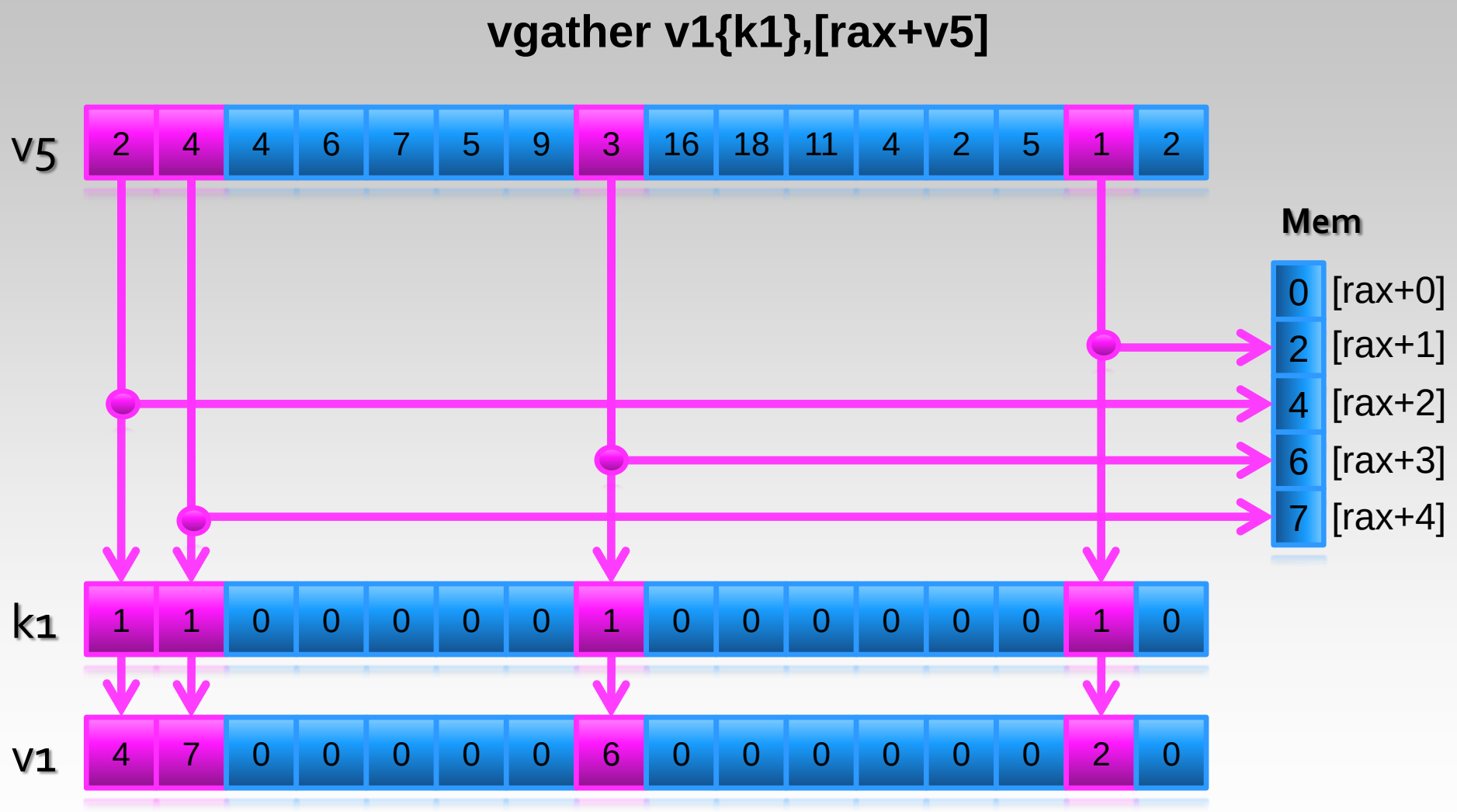
Gather showcase



Gather showcase



All happens at once



Programming Larrabee

- **Larrabee can support industry-standard GPU programming models such as**
  - Direct3D
  - OpenGL
  - Other standard APIs
- **Larrabee Native**
  - Access full power and flexibility of LRB architecture

Larrabee Native Programming

- **Two tightly paired binaries**
  - Host CPU
  - Larrabee
- **Host CPU code**
  - Load Larrabee binary
  - Data transfers & message passing to/from Larrabee
- **Larrabee code**
  - Data transfers & message passing to/from Host
  - Access fixed-function graphics HW
  - x86 + SIMD

Building Larrabee code

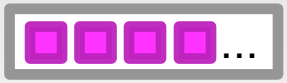
- **Assembly**
  - More compact, easier to understand underlying machine
  - Difficult to integrate with existing codebase
  - Difficult to maintain

- **Full C/C++ compiler + LRBni intrinsics**
  - Parallel libraries
  - More robust language constructs, more debuggable
  - What we expect most people to use
  - With 32 vector registers, the compiler can do a good job of register allocation and scheduling

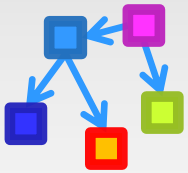
For more info…

- **"A First Look at the Larrabee New Instructions (LRBni)"**
  - by Michael Abrash, Dr. Dobb's Journal, Apr'09
  - http://www.ddj.com/architect/216402188
- **C++ Larrabee Prototype Library Guide**
  - http://software.intel.com/en-us/articles/prototype-primitives-guide/

Larrabee in parallel



Data-Parallel



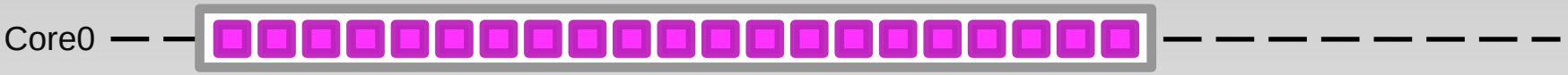
Task-Parallel

Pipeline-Parallel

Data Parallel

Run a single kernel over many elements

- Sequential workload



- Parallel workload

Data Parallel

Run a single kernel over many elements

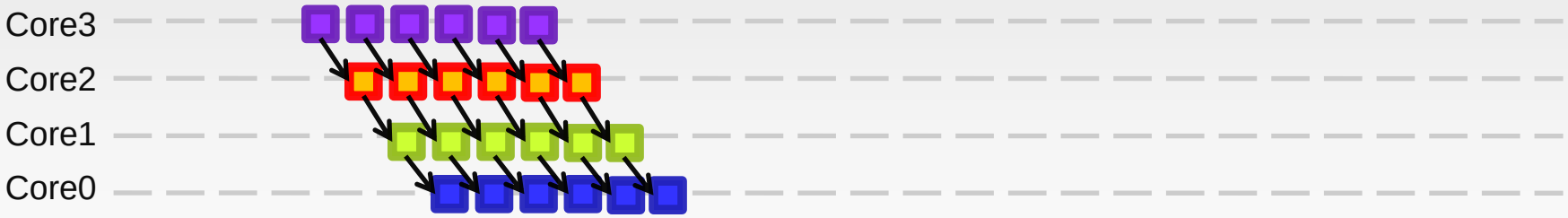
- **Can exploit throughput architecture well**
  - Amortize per-element cost with SIMD/SIMT
  - Hide memory latency with lightweight threads
- **Algorithm**
  - Launch N independent work items
  - All running the same program code (kernel)
  - Data operated on is function of 0 <= i < N

Pipeline parallel

Increase throughput by running multiple stages of an algorithm in parallel

- Sequential workload

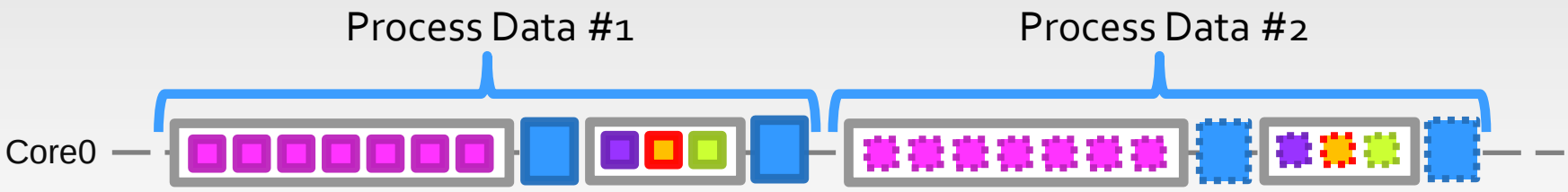
- Parallel workload



Task parallel

Achieve scalability for heterogeneous and irregular work by expressing dependencies directly

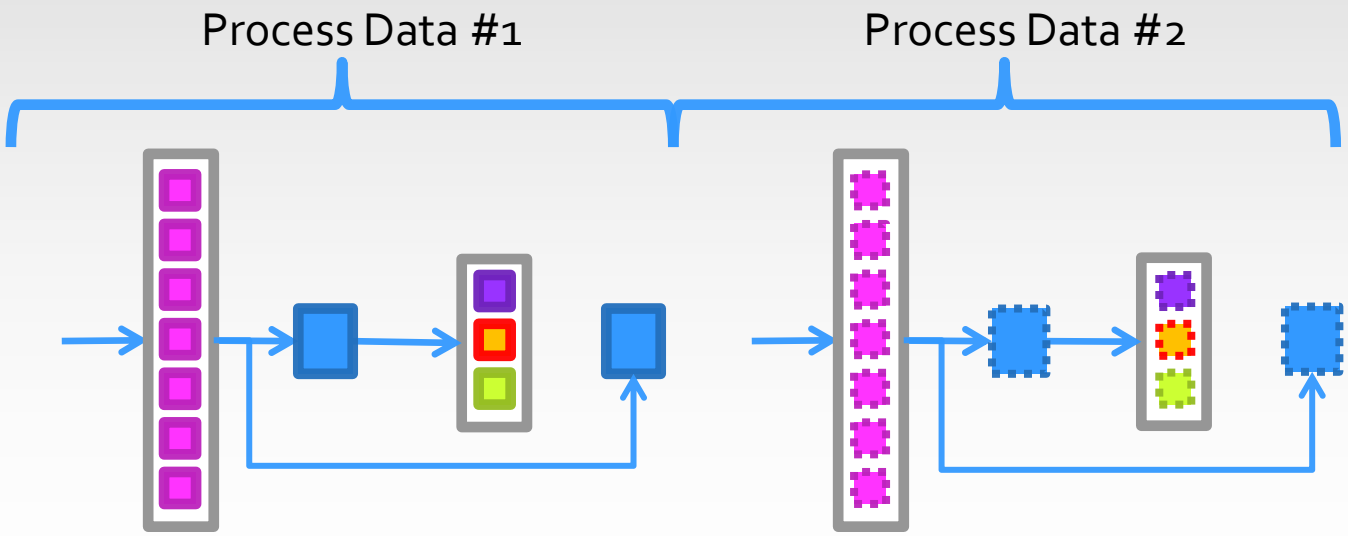
- Sequential workload



Task parallel

Achieve scalability for heterogeneous and irregular work by expressing dependencies directly

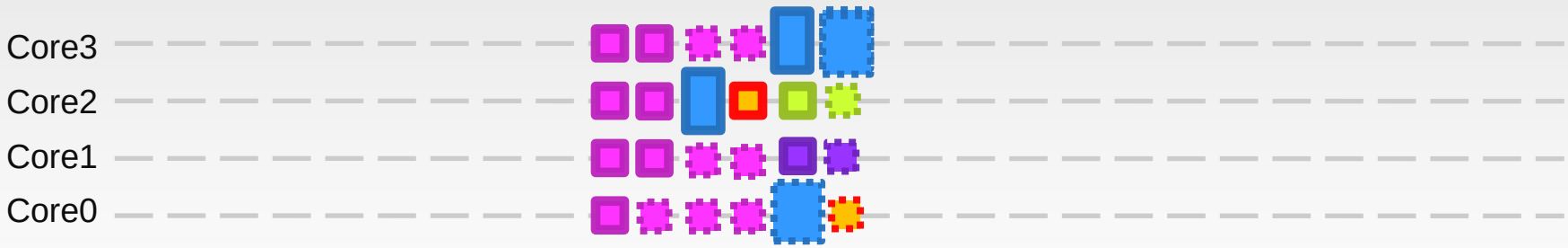
- Break into asynchronous stages
- Build dependencies graph



Task parallel

Achieve scalability for heterogeneous and irregular work by expressing dependencies directly

- Parallel workload



Task

A task:

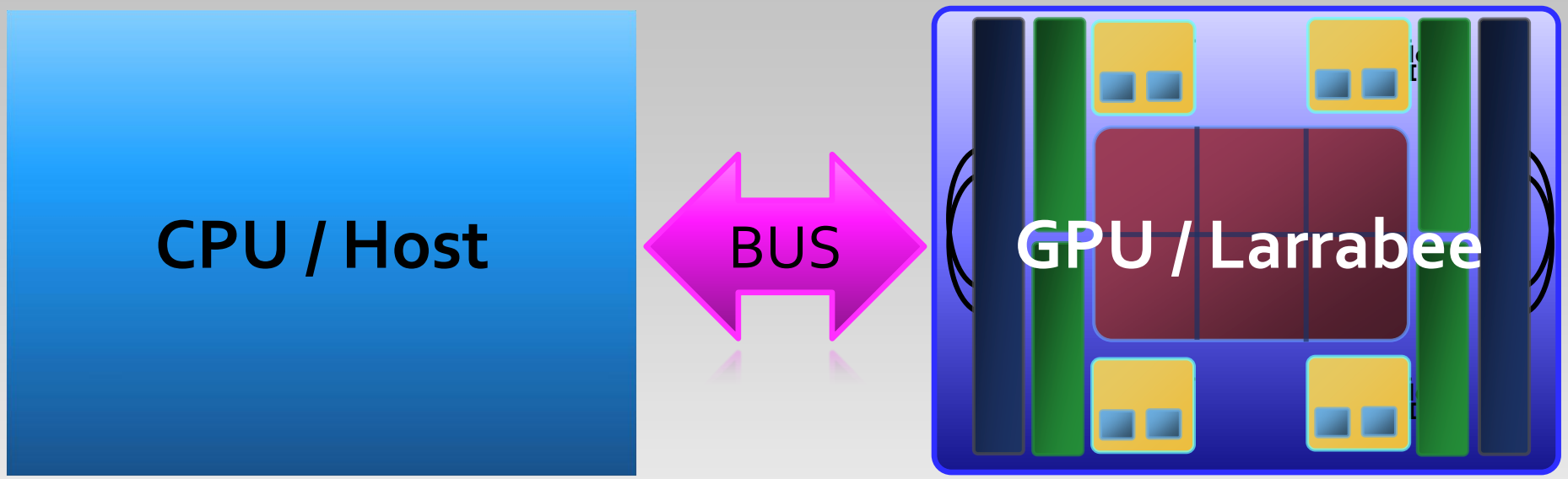
- Think of task as an asynchronous function call
- Implemented as "lightweight, user-space cooperative execution entity that virtualizes HW threads"
- Successful scalable parallel programming model for Xbox 360*, PlayStation 3*, multi-core CPUs, and **Larrabee**
- Tasks automatically scale with increasing core count
- Spawn many more tasks than cores

Task system:

- Handle the code that won't fit other models
- Heterogeneous, irregular
- Dynamically generated work, dependencies
- Provide scalability and load balancing

Case Studies

CPU <-> GPU: Beforehand note



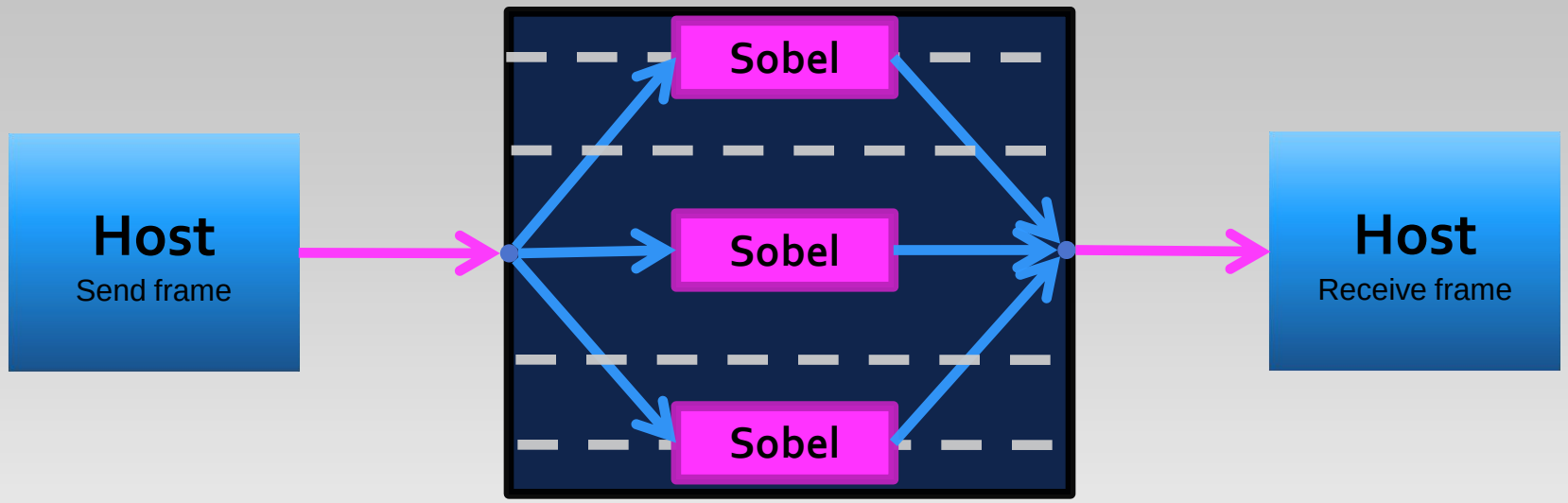
- Data transfer thru bus is **LONG**
- Larrabee ring is **FAST**
- Offload work to Larrabee
- Minimize data transfers to/from Host

Sobel 3x3 filter

$$b(i,j) = \sum_{(k,l)} w(k,l)\,a(i+k, j+l),\ w = \frac{1}{8}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

- **Run a single kernel over multiple elements**
  - Native Data Parallelism
  - Process 16 elements at once with SIMD
- **Exploit throughput architecture**
  - Linear memory access pattern – pre-fetch

Sobel filter implementation



Larrabee

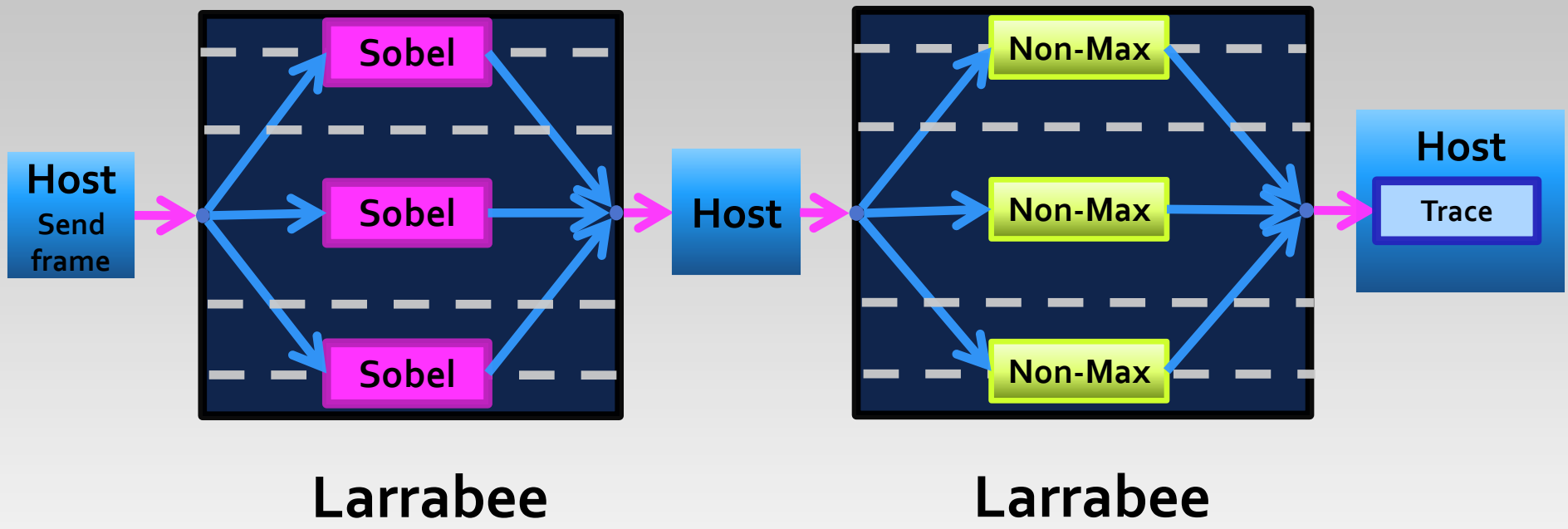
- Send frame to Larrabee
- Distribute work across cores
- Process 16 pixel at once with SIMD
- Send data back to Host

Canny edge detector

- **Algorithm**

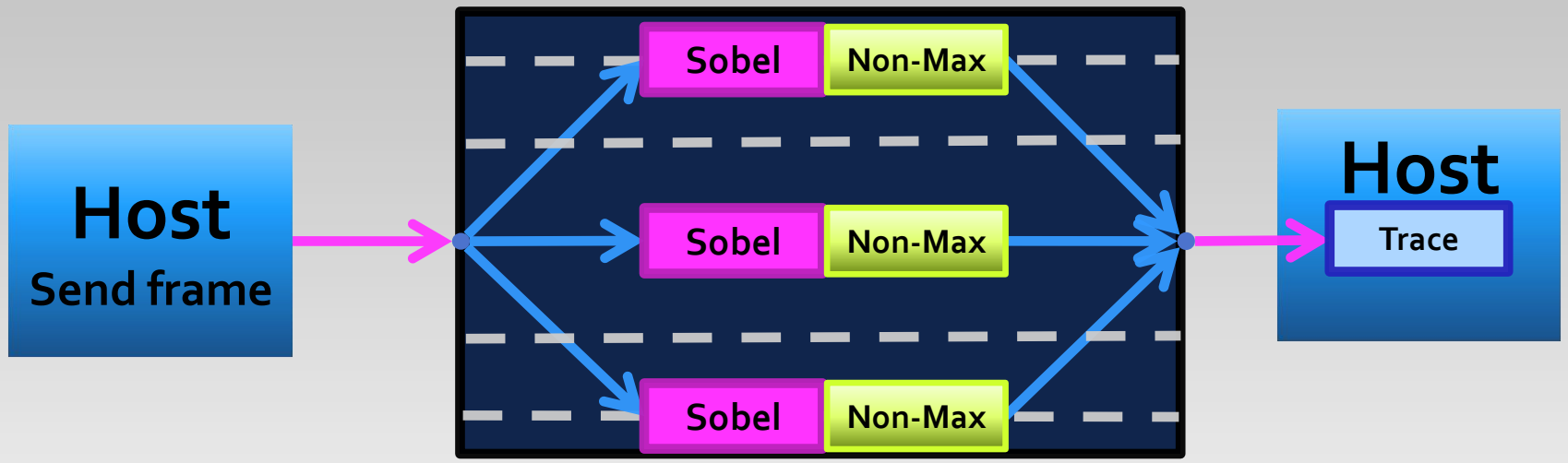
1. Find intensity gradient via Sobel
2. Perform Non-maximum suppression
3. Trace edges

Canny: First iteration



- 2 scans of image
- 2 sync points
- 4 "Host – Larrabee" transfers

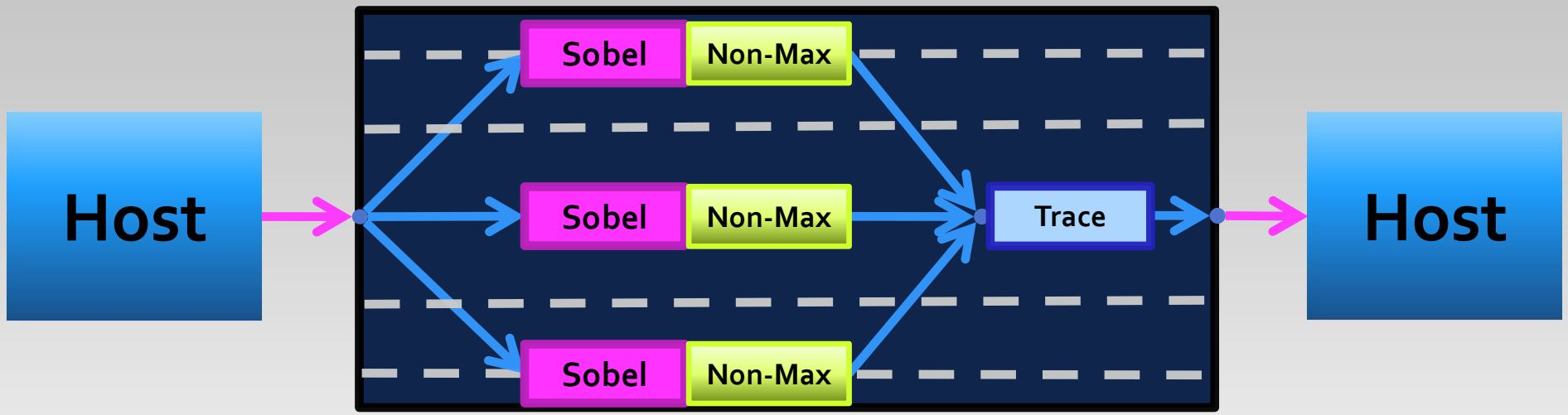
Canny: Second iteration



Larrabee

- Combine 2 kernels into one task
- Use task-local storage for passing data between kernels
- 1 scan of data and 1 sync point
- 2 "Host – Larrabee" transfers

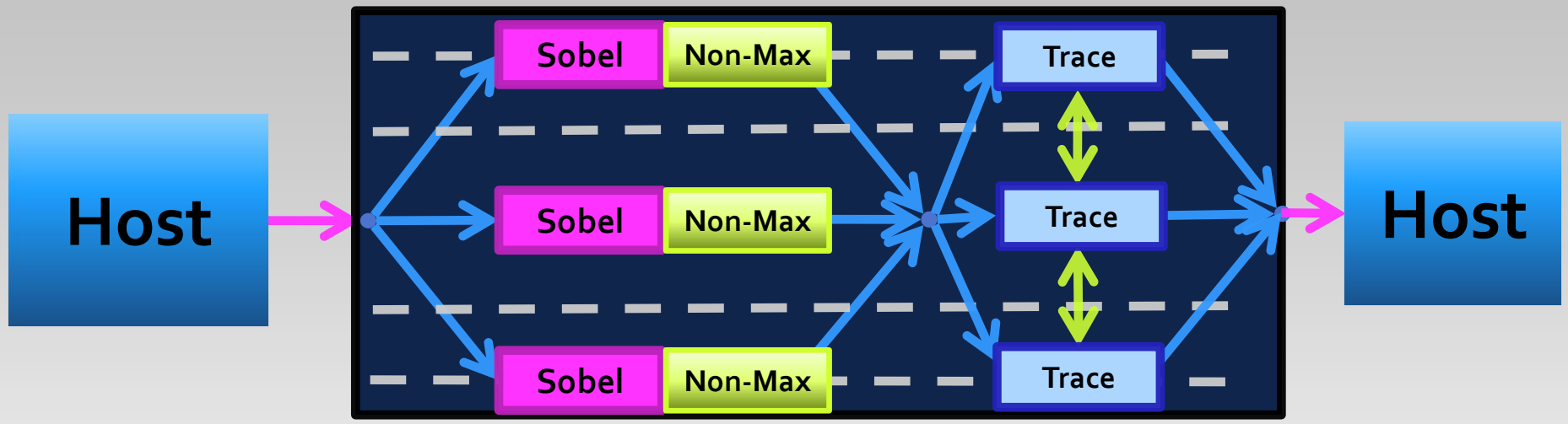
Canny: Third iteration



Larrabee

- Run whole Canny pipeline on Larrabee
- Move "Trace" task to Larrabee
  - Make it dependant task
- 1 scan of data and 1 sync point
- Host CPU is FREE

Canny: Forth iteration



Larrabee

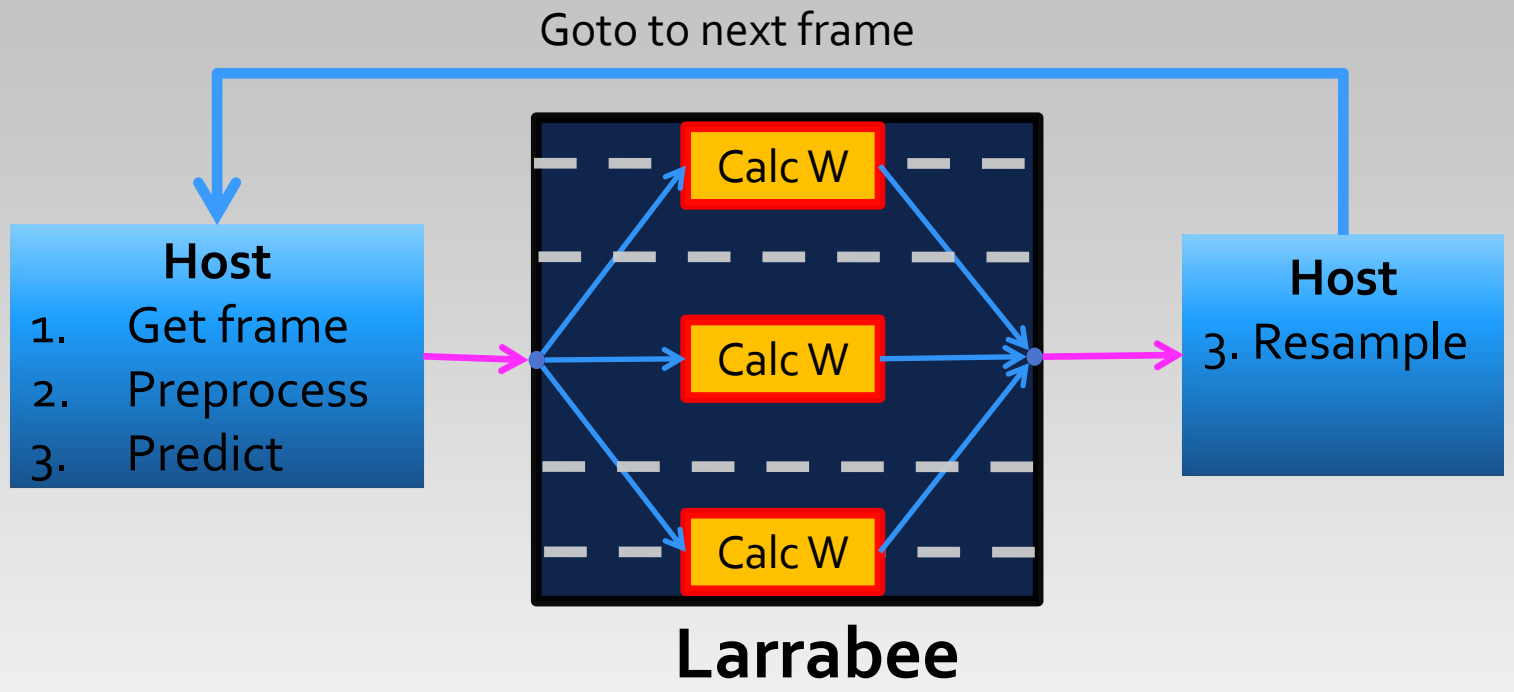
- "Trace" edges is pseudo-asynchronous task
- Make it parallel using fast InterlockIncrement() to share jobs between tasks
- 2 sync points
- 100% pipeline parallel

Particle filtering (PF)

Algorithm:

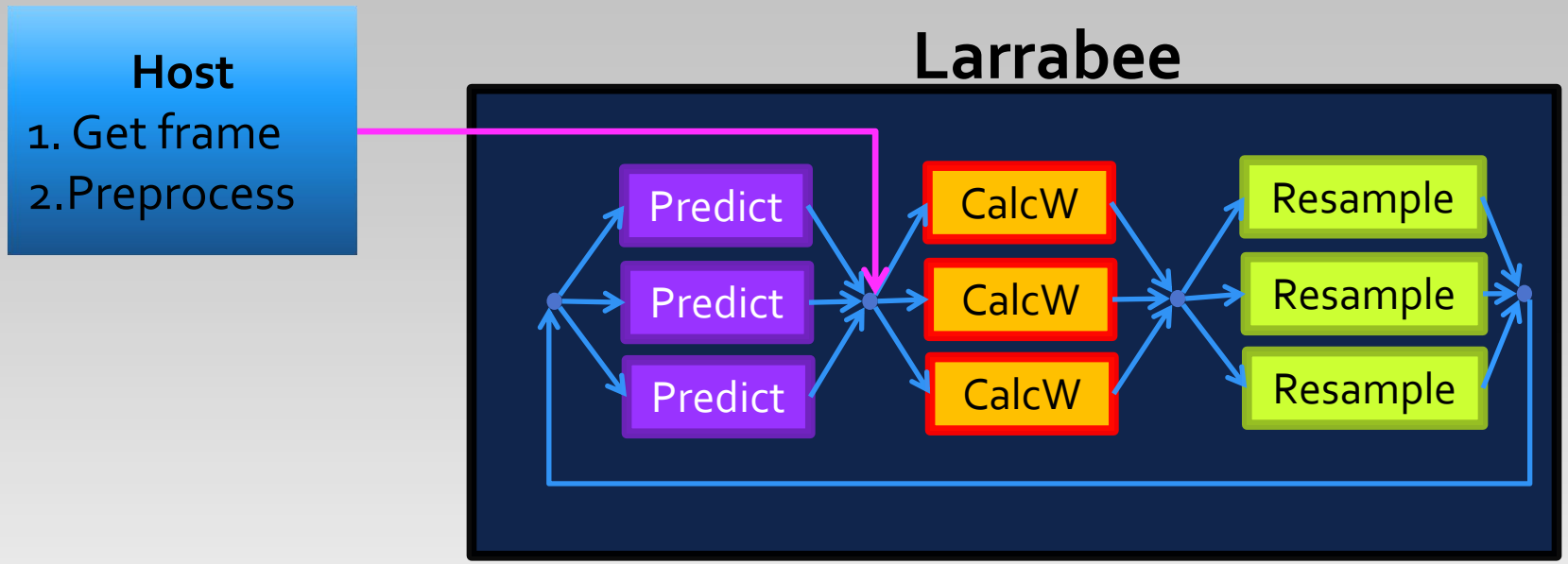
1. Get input data
2. Preprocess input data
3. Predict particles
4. Calc weights for each particle
5. Resample particles
6. Goto step 1

PF: First iteration



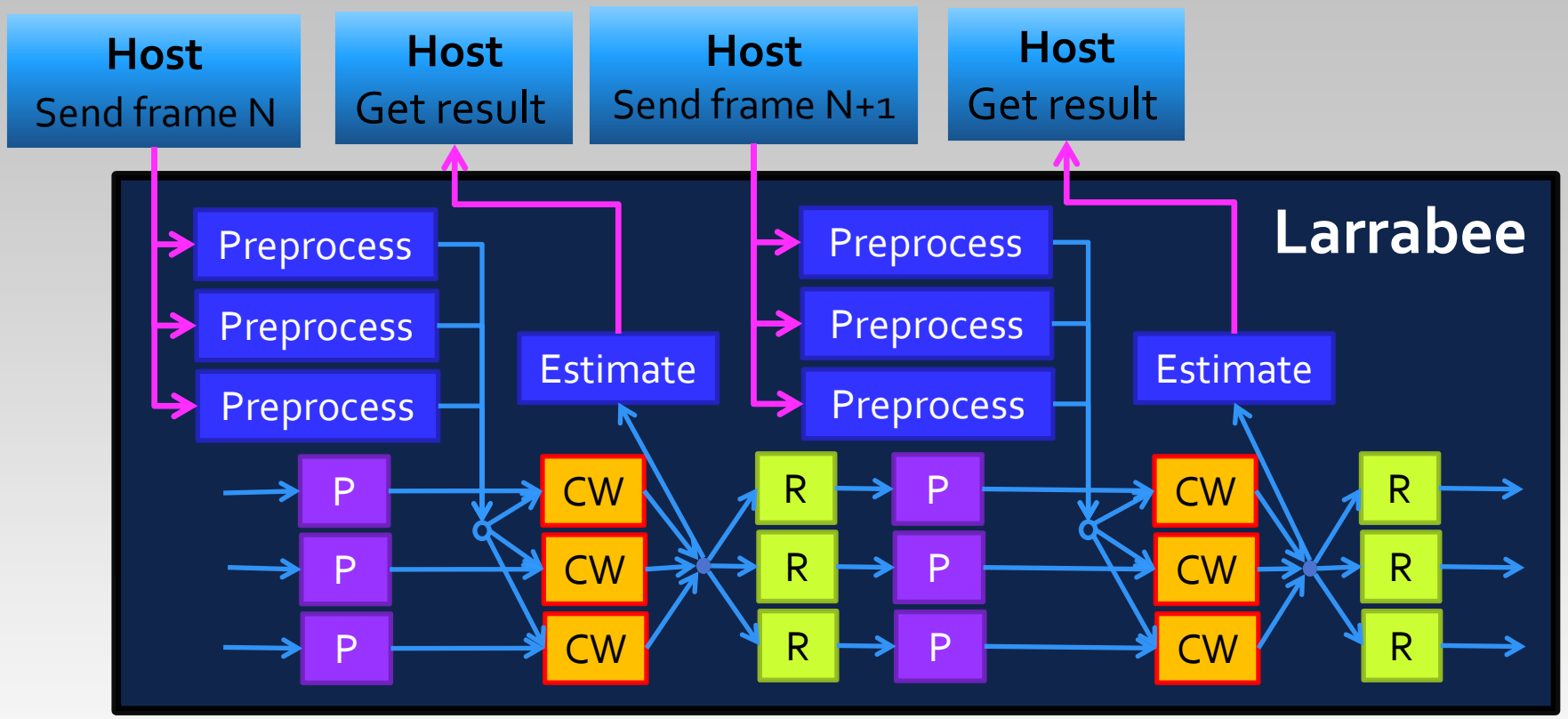
- Most expensive part is weight calculation
- The obvious and easy way to parallelize
- Not all stages parallelized and run on Larrabee side
- Particle and weight data transfer from/to Larrabee on each step

PF: Second iteration



- Implement Predict and Resample as tasks
- Build dependencies graph
- Whole iteration cycle is on the Larrabee side
- Only new data for weight calculation is transferred from HOST

PF: Third iteration



- Implement Preprocess as task
- Implement Parallel pipeline by feeding several frames at once
- 100% Data processing on Larrabee
- Output: estimated vector of parameters

Scalability 4 CV Kernels

CV kernels considered

- **Canny edge detector**
  - 1 complex kernel: Sobel + Non-Maxima suppression
  - *[Canny, A Computational Approach to Edge Detection, IEEE Trans. PAMI'86]*

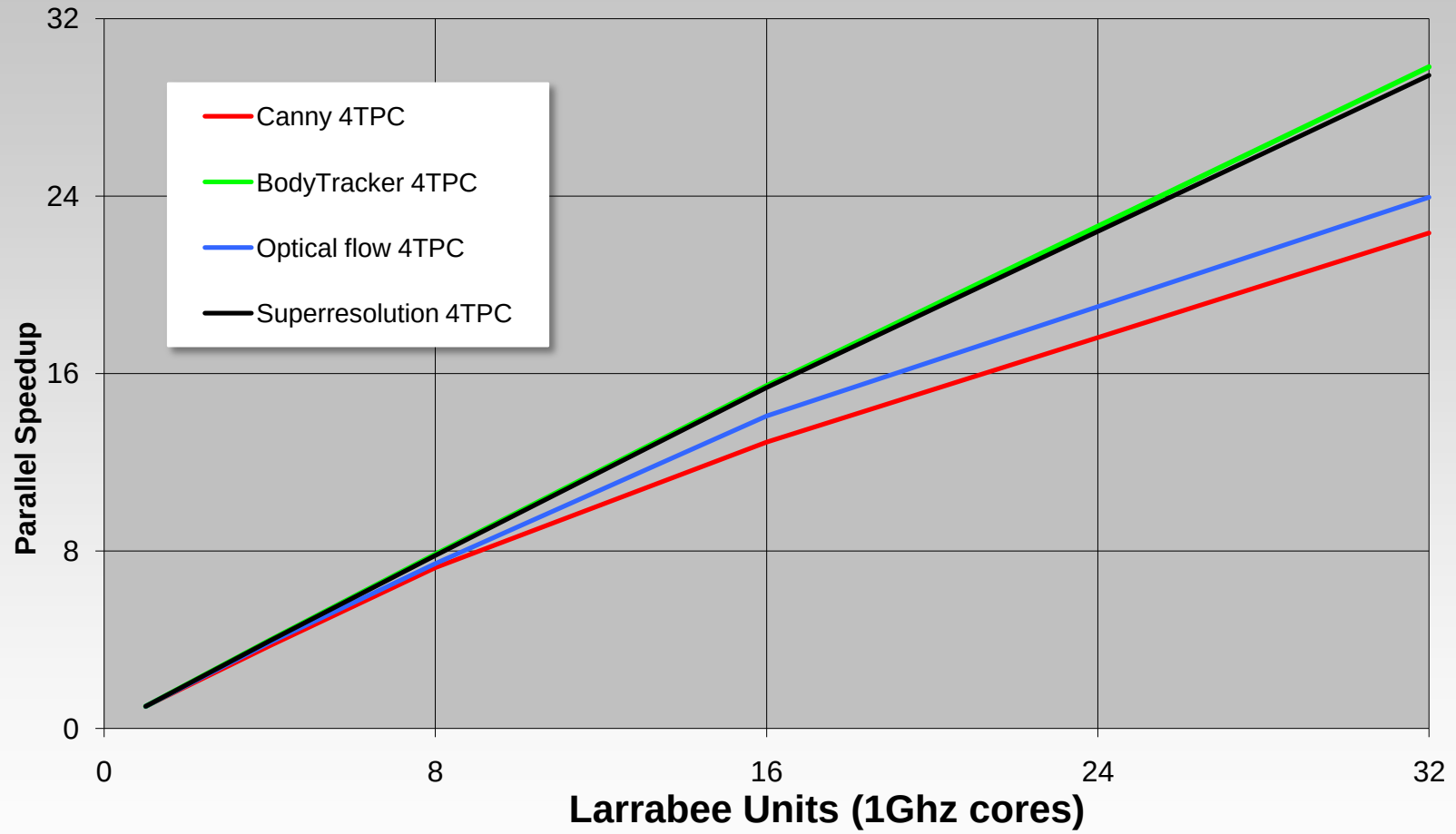
- **Body tracker**
  - Annealing particle filter optimization
  - Complex kernel for weight calculation for 1000 particles
  - Weight is complex function based on foreground, edge and color information
  - *[Deutscher, J.et al, Articulated body motion capture by annealed particle filtering, IEEE CVPR'00]*

CV kernels considered, 2

- **Optical flow**
  - Simulation of one algorithm iteration on image pair
  - *[G. Farnebäck. Polynomial expansion for orientation and motion estimation, Ph.D. thesis, '02].*

- **Super-resolution**
  - Simulation of Gradient calculation - most expensive part
  - *[S. Farsiu, et al: Fast and Robust Multiframe Super Resolution, IEEE IP, Oct'04]*

Scalability



Summary

Larrabee is a many-core x86 architecture

- Anyway, it is optimized for graphics
- But works **FINE** with Computer Vision workloads

Best for parallel

- Data, Pipeline, Task …
- Fast ring interconnect
- Modern memory system, caches

Acknowledgements

- Organizers:
  - P. J. Narayanan, Joe Stam, Jan-Michael Frahm
- Intel:
  - Tom Forsyth, Tim Foley, Aaron Lefohn, Pradeep Dubey, Larry Seiler, Yen-Kuang Chen, Alexei Soupikov

Legal Disclaimer

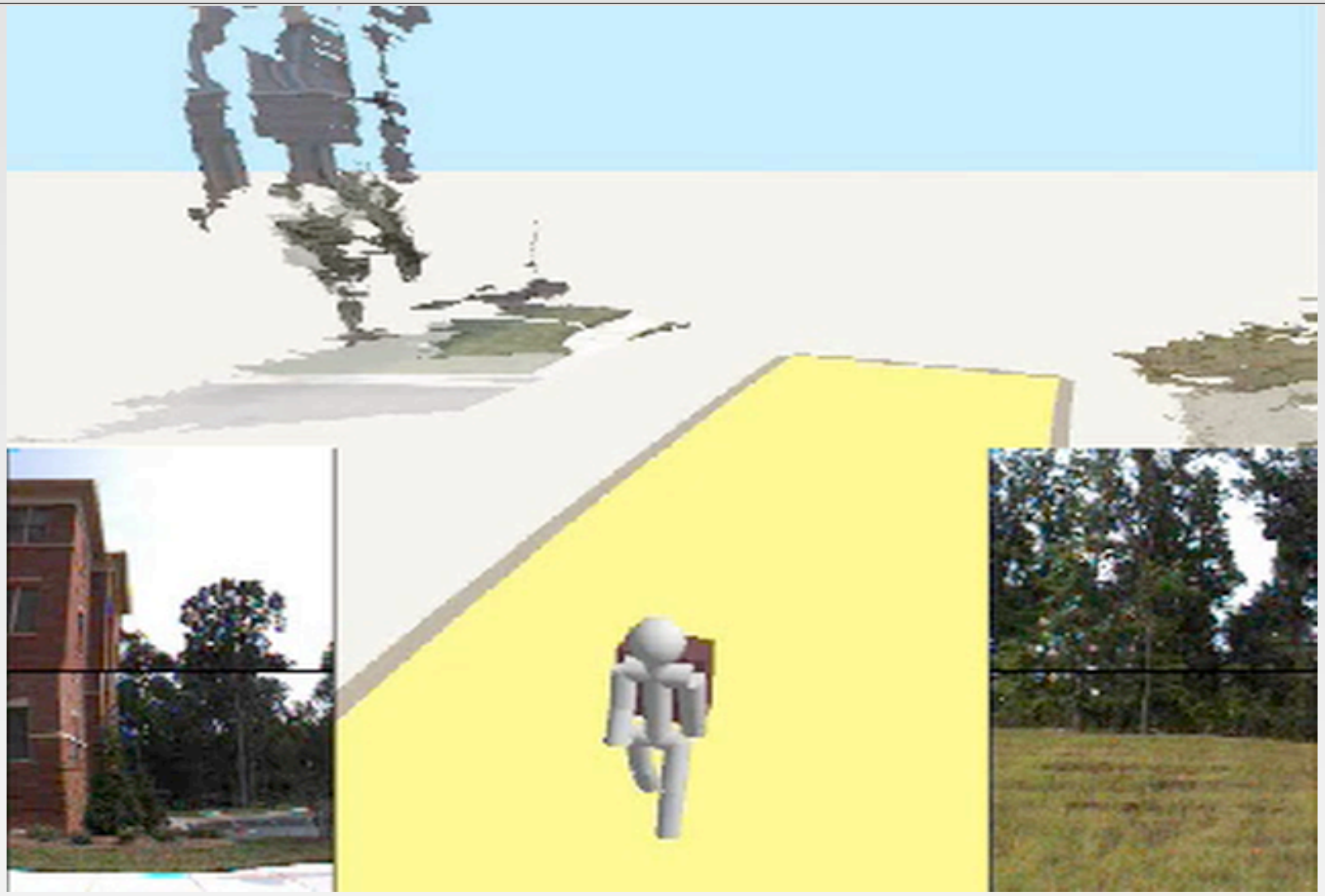
- INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL® PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. INTEL PRODUCTS ARE NOT INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS.
- Intel may make changes to specifications and product descriptions at any time, without notice.
- All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.
- Intel, processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.
- Larrabee and other code names featured are used internally within Intel to identify products that are in development and not yet publicly announced for release. Customers, licensees and other third parties are not authorized by Intel to use code names in advertising, promotion or marketing of any product or services and any such use of Intel's internal code names is at the sole risk of the user
- Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.
- Intel, Intel Inside and the Intel logo are trademarks of Intel Corporation in the United States and other countries.
- *Other names and brands may be claimed as the property of others.
-

Risk factors

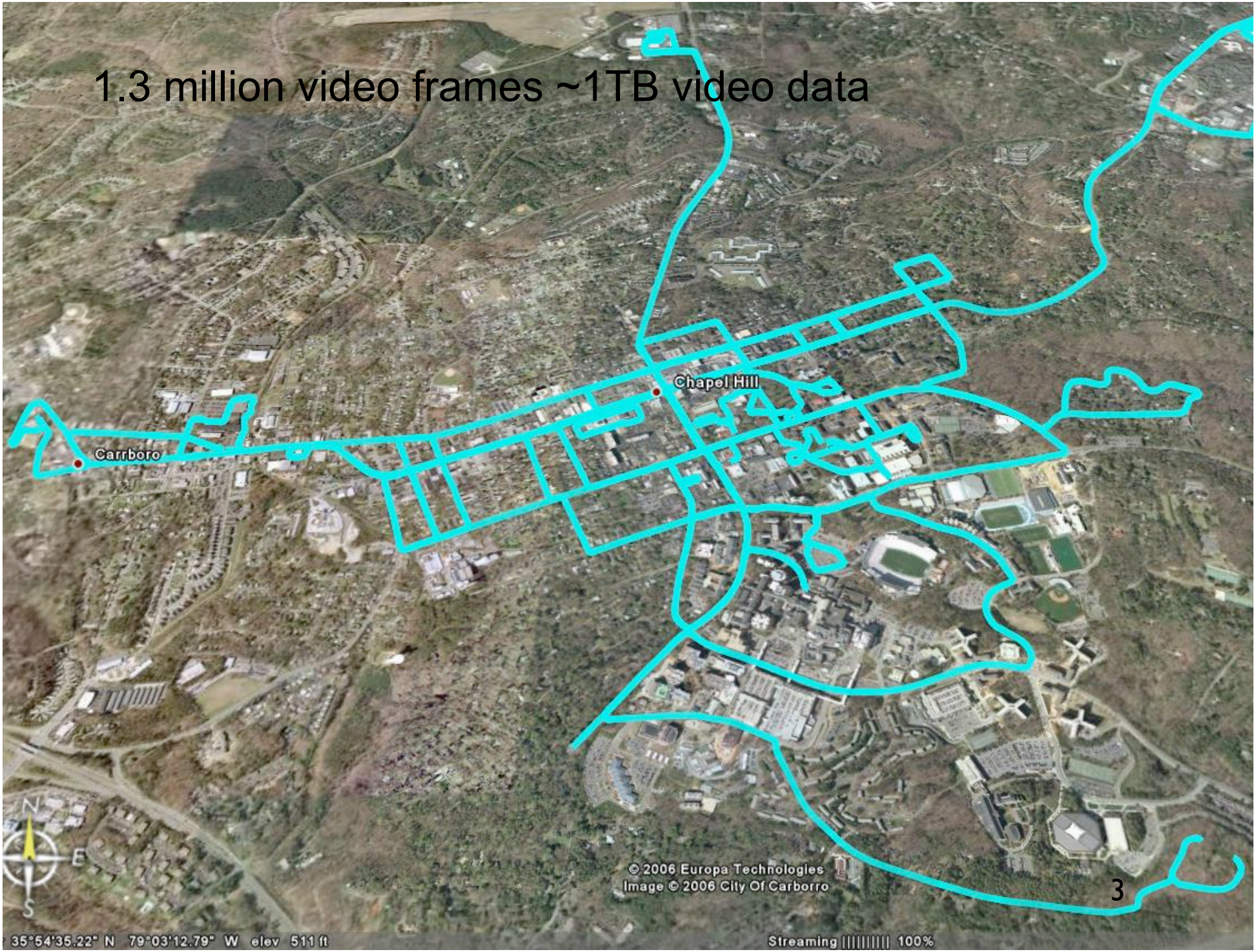
This presentation contains forward-looking statements that involve a number of risks and uncertainties. These statements do not reflect the potential impact of any mergers, acquisitions, divestitures, investments or other similar transactions that may be completed in the future. The information presented is accurate only as of today's date and will not be updated. In addition to any factors discussed in the presentation, the important factors that could cause actual results to differ materially include the following: Demand could be different from Intel's expectations due to factors including changes in business and economic conditions, including conditions in the credit market that could affect consumer confidence; customer acceptance of Intel's and competitors' products; changes in customer order patterns, including order cancellations; and changes in the level of inventory at customers. Intel's results could be affected by the timing of closing of acquisitions and divestitures. Intel operates in intensely competitive industries that are characterized by a high percentage of costs that are fixed or difficult to reduce in the short term and product demand that is highly variable and difficult to forecast. Revenue and the gross margin percentage are affected by the timing of new Intel product introductions and the demand for and market acceptance of Intel's products; actions taken by Intel's competitors, including product offerings and introductions, marketing programs and pricing pressures and Intel's response to such actions; Intel's ability to respond quickly to technological developments and to incorporate new features into its products; and the availability of sufficient supply of components from suppliers to meet demand. The gross margin percentage could vary significantly from expectations based on changes in revenue levels; product mix and pricing; capacity utilization; variations in inventory valuation, including variations related to the timing of qualifying products for sale; excess or obsolete inventory; manufacturing yields; changes in unit costs; impairments of long-lived assets, including manufacturing, assembly/test and intangible assets; and the timing and execution of the manufacturing ramp and associated costs, including start-up costs. Expenses, particularly certain marketing and compensation expenses, vary depending on the level of demand for Intel's products, the level of revenue and profits, and impairments of long-lived assets. Intel is in the midst of a structure and efficiency program that is resulting in several actions that could have an impact on expected expense levels and gross margin. Intel's results could be impacted by adverse economic, social, political and physical/infrastructure conditions in the countries in which Intel, its customers or its suppliers operate, including military conflict and other security risks, natural disasters, infrastructure disruptions, health concerns and fluctuations in currency exchange rates. Intel's results could be affected by adverse effects associated with product defects and errata (deviations from published specifications), and by litigation or regulatory matters involving intellectual property, stockholder, consumer, antitrust and other issues, such as the litigation and regulatory matters described in Intel's SEC reports. A detailed discussion of these and other factors that could affect Intel's results is included in Intel's SEC filings, including the report on Form 10-Q for the quarter ended June 28, 2008.

Applications

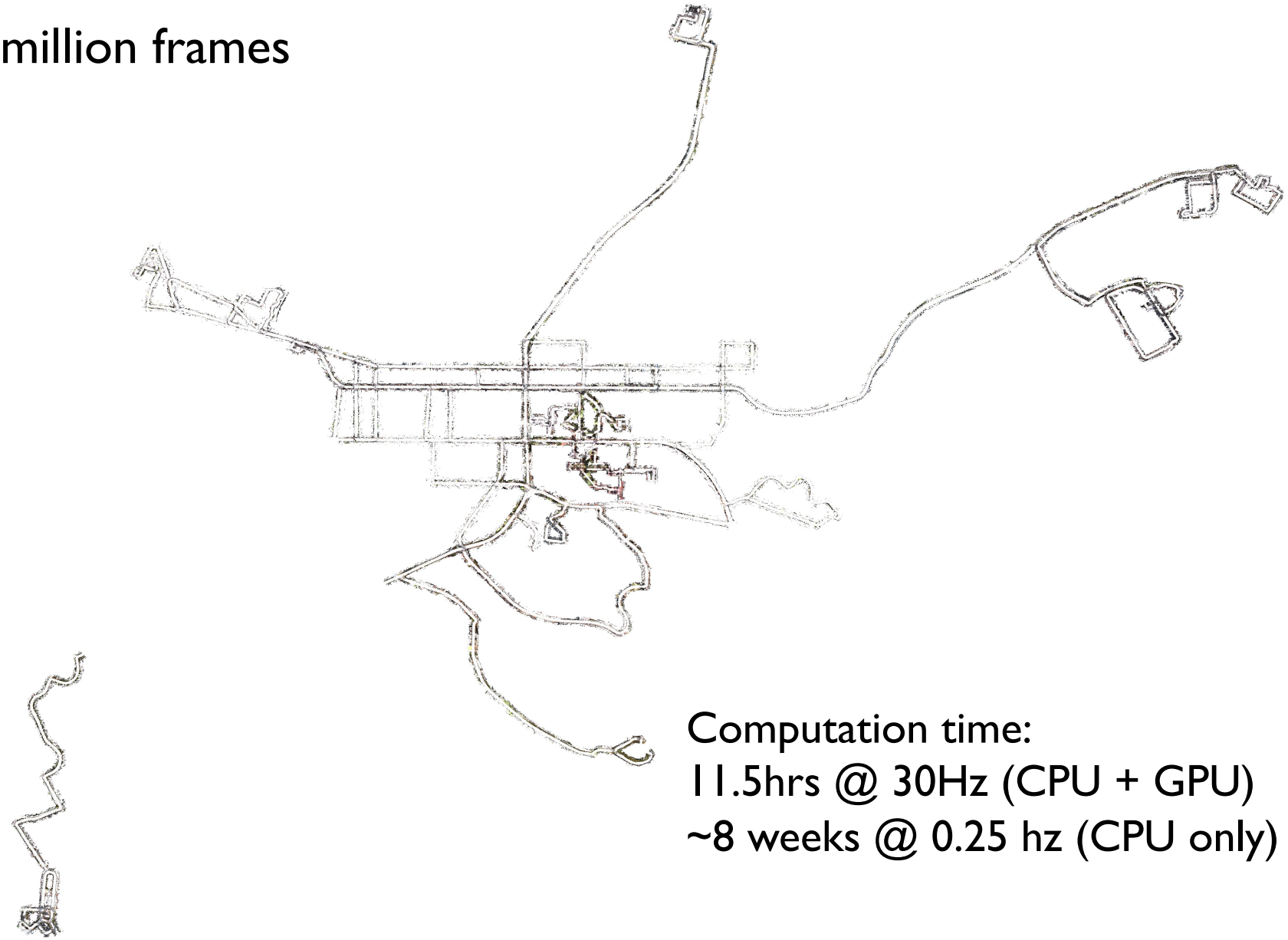
Real-time Scene Reconstruction



UNC (Pollefeys, Frahm) & UK (Nister, Yang) in UrbanScape project


1.3 million video frames ~1TB video data
Chapel Hill
Carrboro
© 2006 Europa Technologies
Image © 2006 City Of Carborro
3
35°54'35.22" N 79°03'12.79" W elev 511 ft
Streaming 100%

- 1.3 million frames



Computation time:
11.5hrs @ 30Hz (CPU + GPU)
~8 weeks @ 0.25 hz (CPU only)

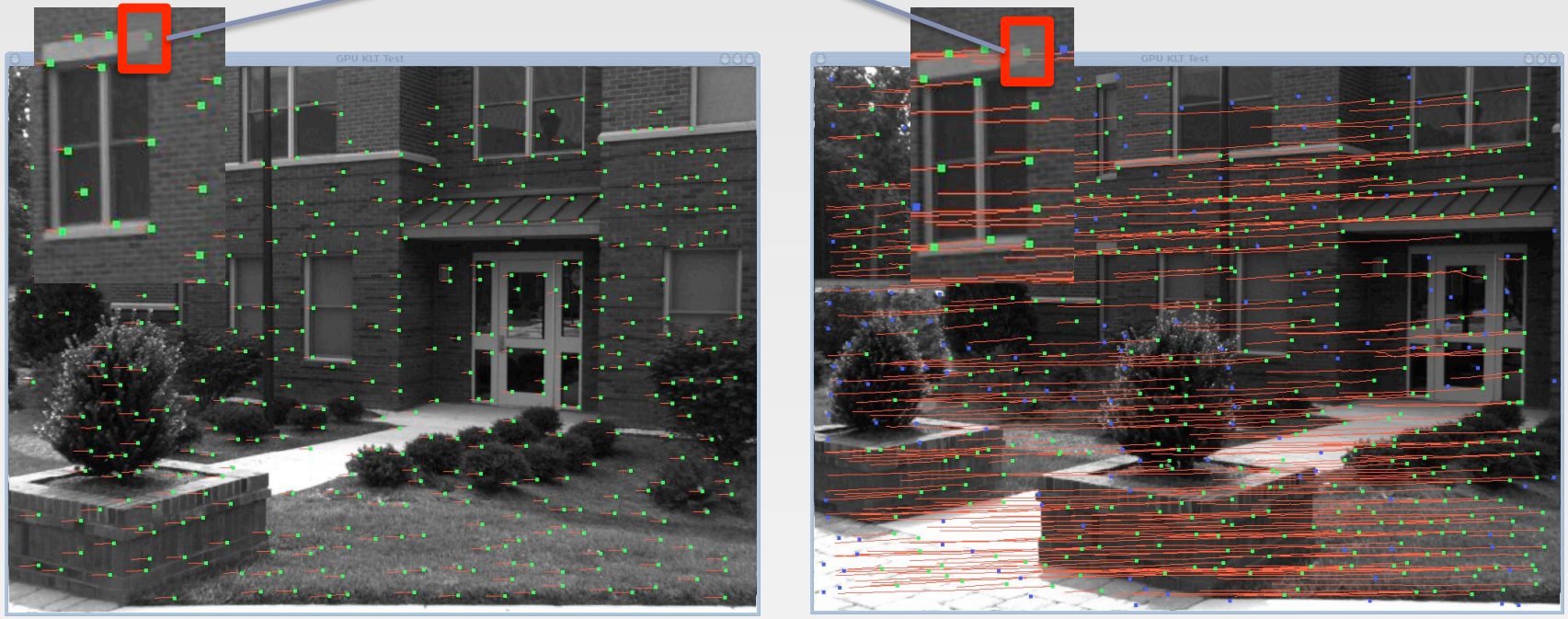
Real-Time Stereo KLT using CUDA

Jan-Michael Frahm and Brian Clipp
Department of Computer Science
University of North Carolina *at* Chapel Hill

Correspondence Estimation

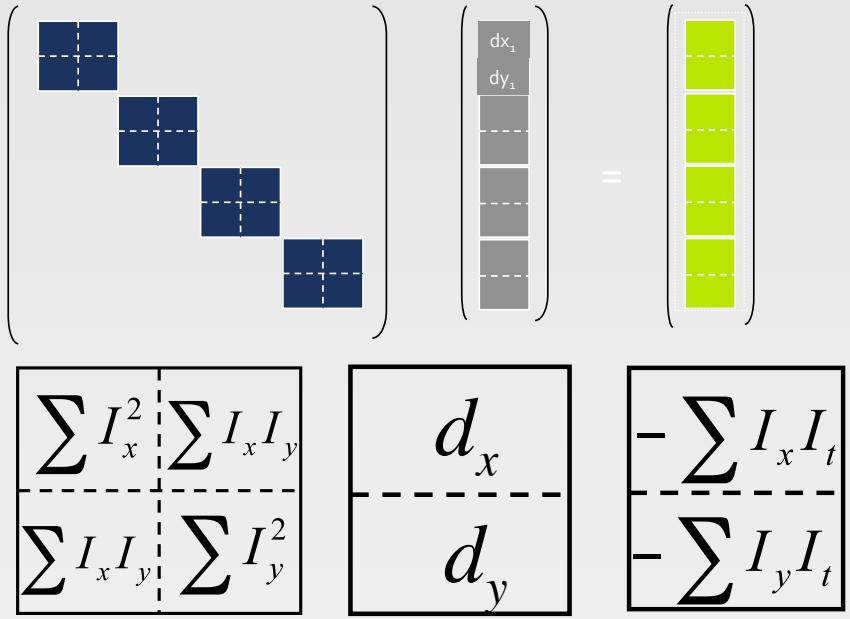
- Assumption: image content varies slowly/smoothly
  - Video sequences

$$I_0(x+d) = I_1(x)$$



KLT tracker [Lucas & Kanade. '81]

Correspondence Estimation



Data parallel implementation on GPU [Sinha, Frahm, Pollefeys, Genc MVA'07]
With camera gain estimation on GPU [Zach, Gallup, Frahm CVGPU'08]

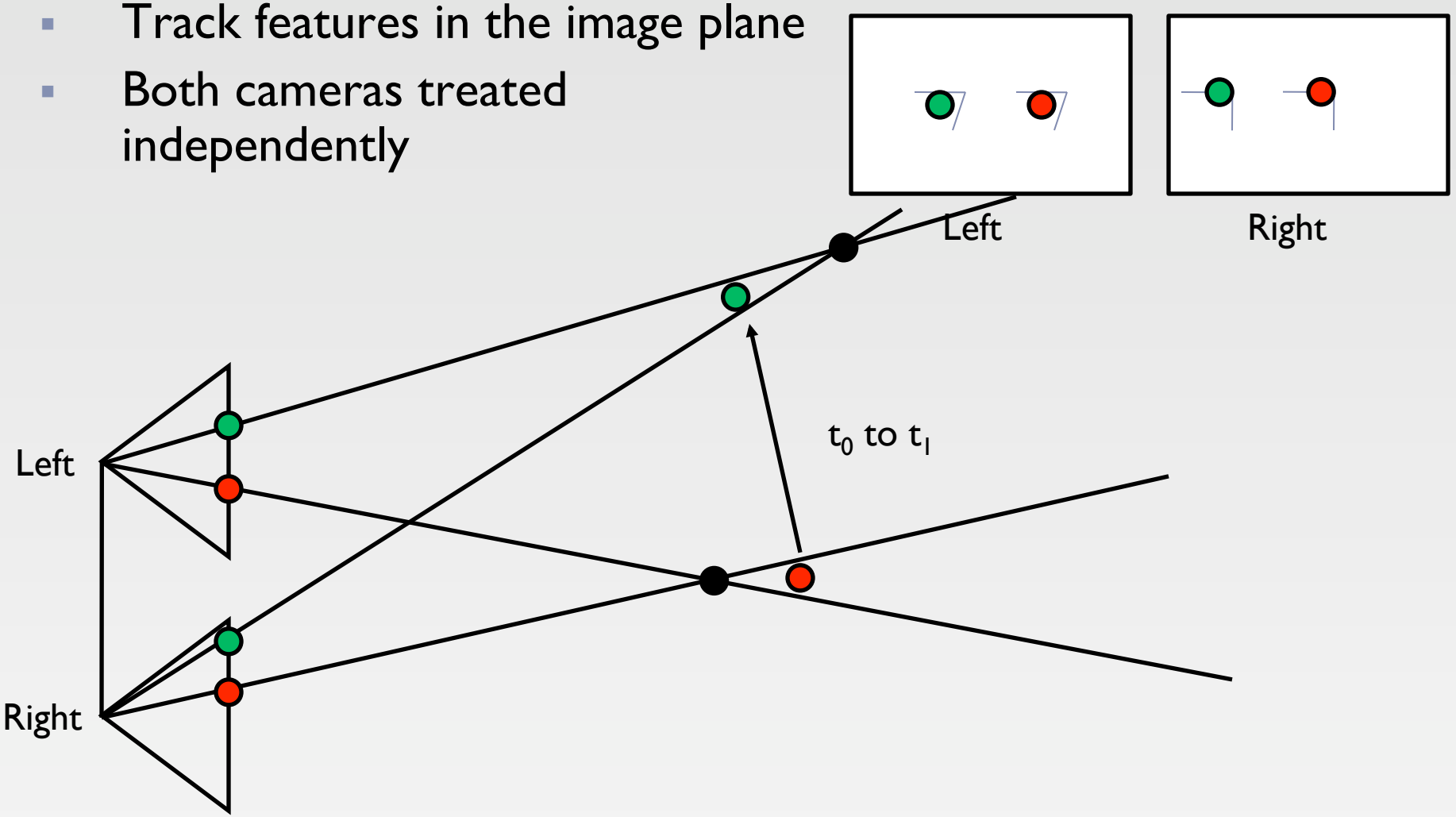
Making robots see

Stereo Camera Pair

with Honda Research

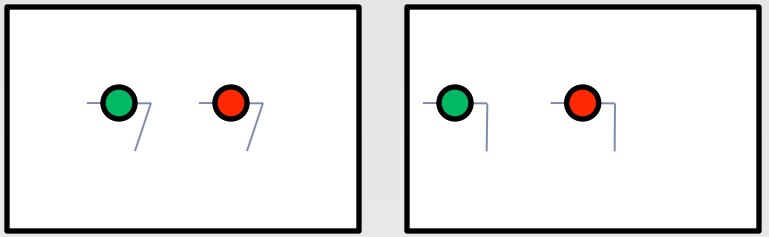
Single Camera KLT Formulation

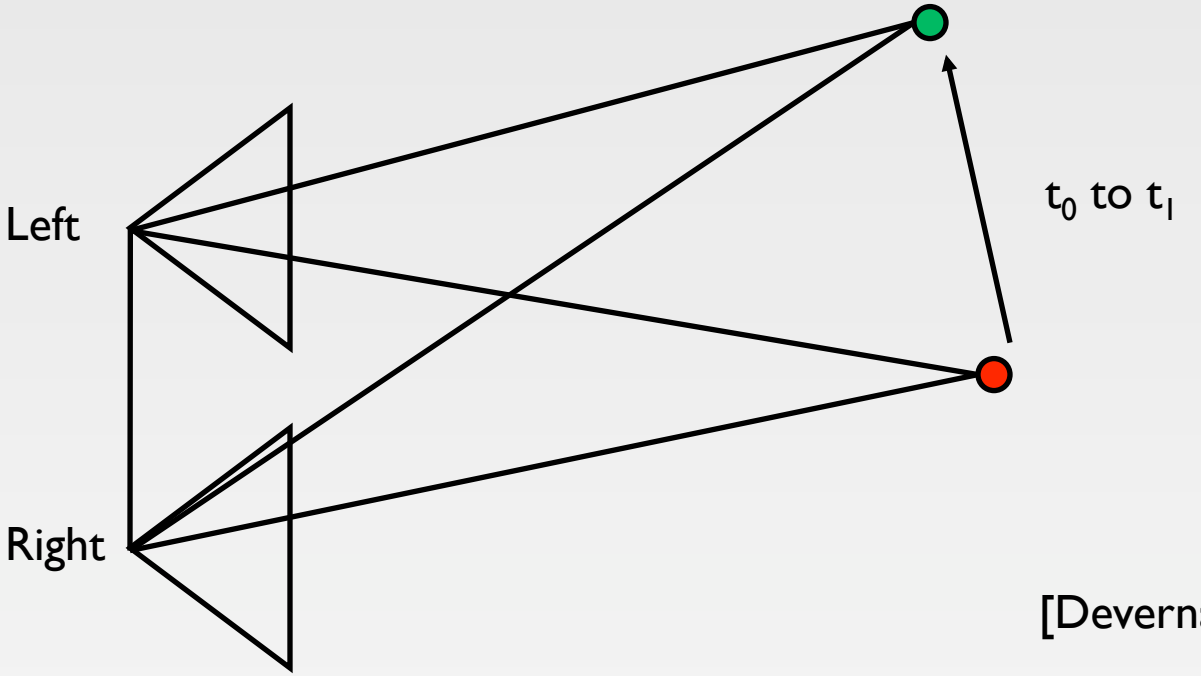
- Track features in the image plane
- Both cameras treated independently



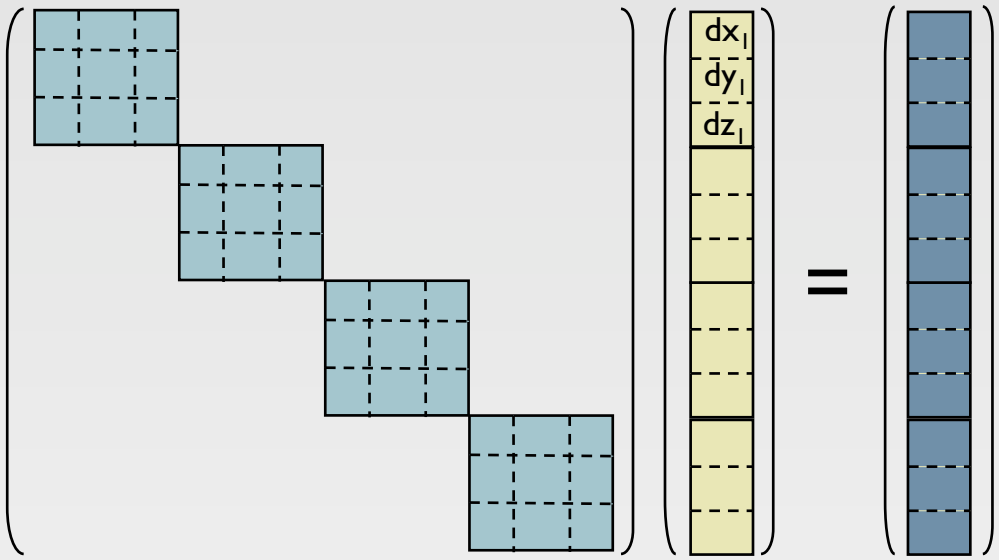
Stereo KLT Formulation

- Track features in 3D
- Both cameras treated jointly





[Devernay et al. CVPR 2006]

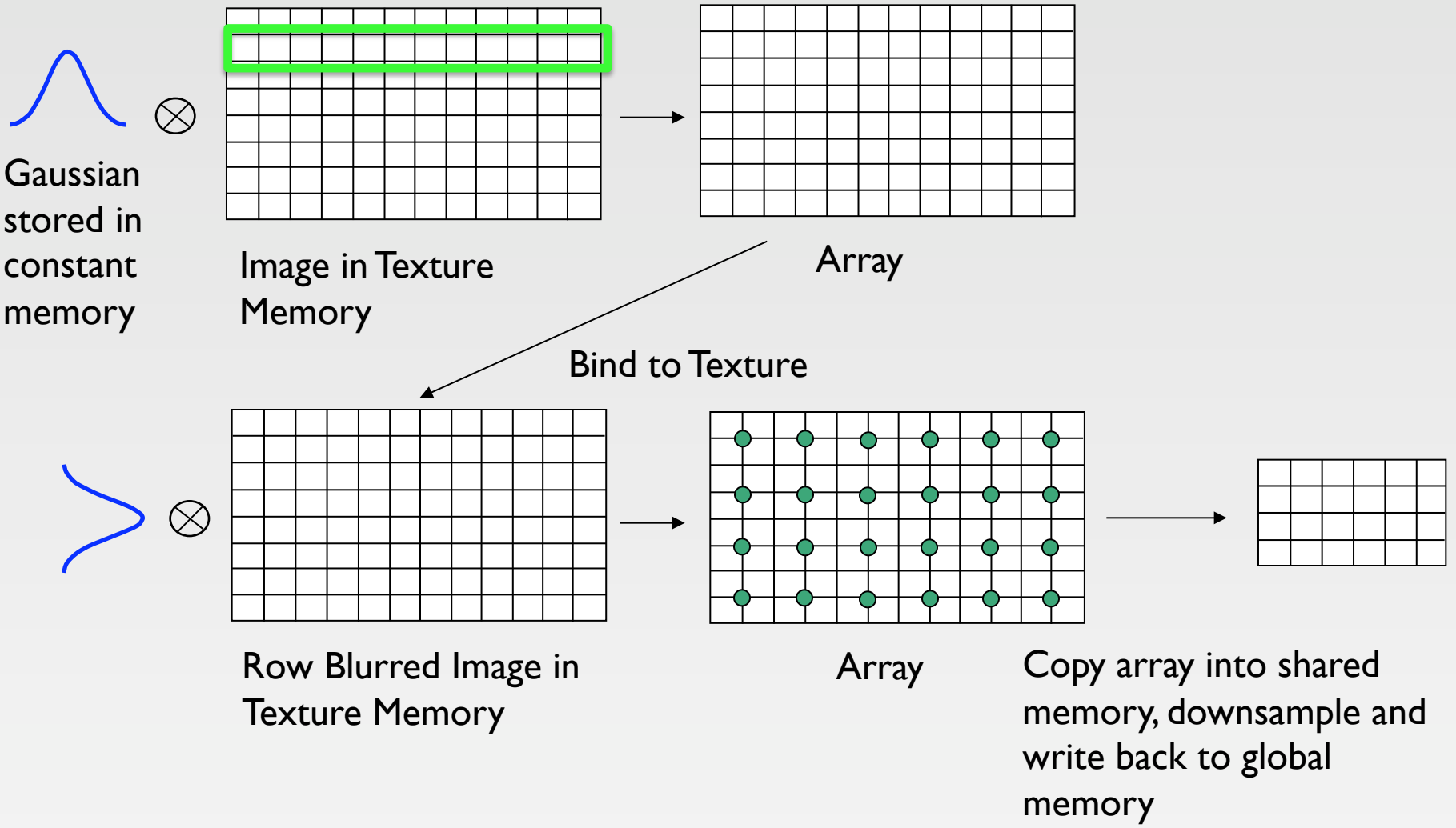


- 3D flow in [dx, dy, dz]
- independent 3x3 equation systems
- one flow vector per thread

CUDA Implementation

- Build Image Pyramid

Pyramid Building

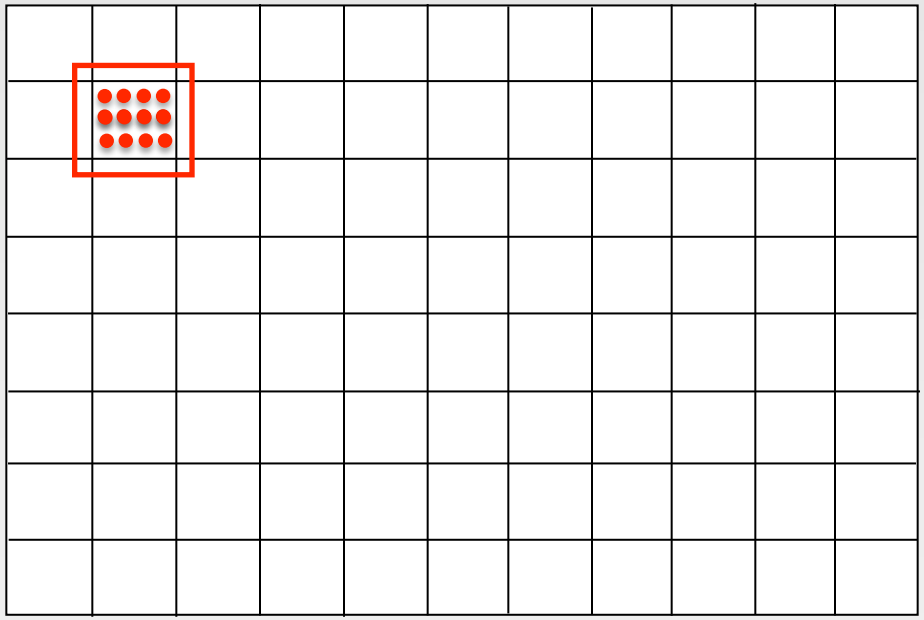


CUDA Implementation

- Build Image Pyramid
- Detection
  - if(sufficient motion || first frame)
    - Calculate cornerness images

Memory use in cornerness computation

- Break image up into regular sub-sections and assign each sub-section to a block



- Copy image block into shared memory
- Each thread then calculates the cornerness of its assigned pixel using data from the shared memory
- written to global memory

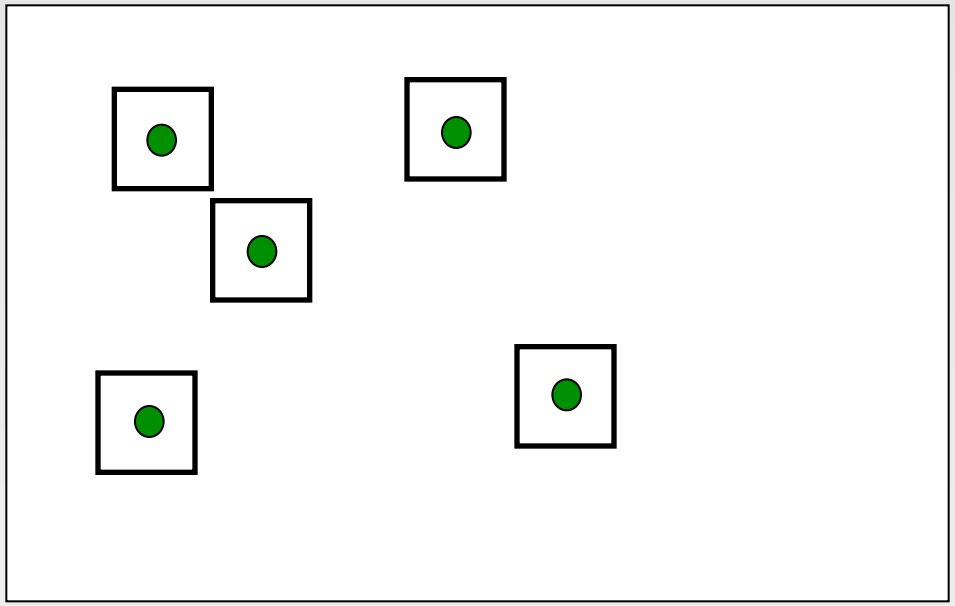
CUDA Implementation

- Build Image Pyramid
- Detection
  - if(sufficient motion || first frame)
    - Calculate cornerness images
    - Perform non-maxima supression on cornerness
    - Match corners that are not near existing features
    - Triangulate new features
    - Remaining features as standard KLT features
- Perform multi-scale tracking update on all features

Memory use for tracking update

- Feature positions distributed randomly across image

Texture memory

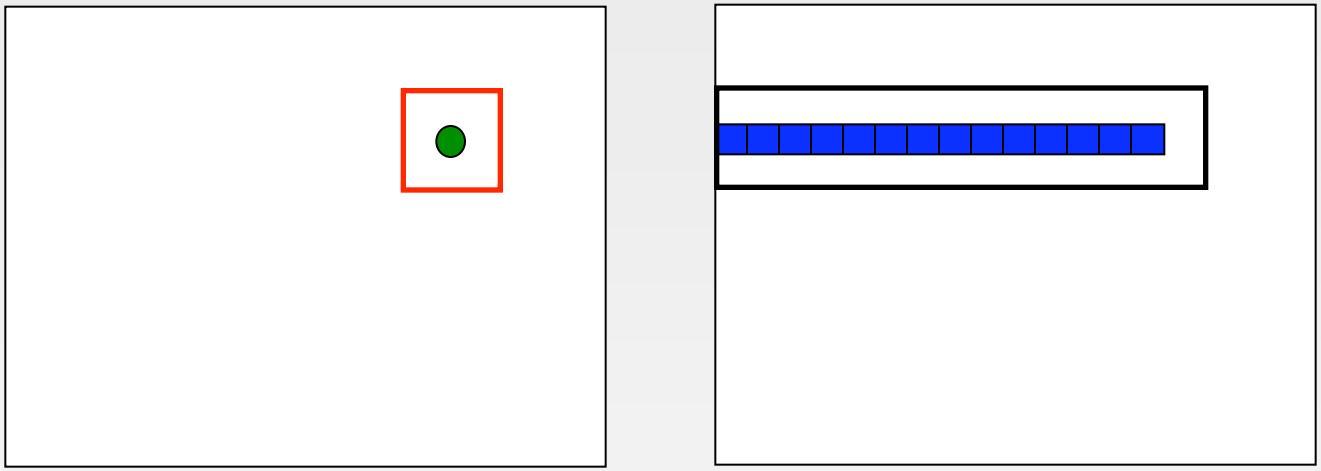


- No predictable overlap in local feature windows for use in memory management
- Use texture memory for access to image data to take advantage of the texture cache

Memory use for Stereo Match

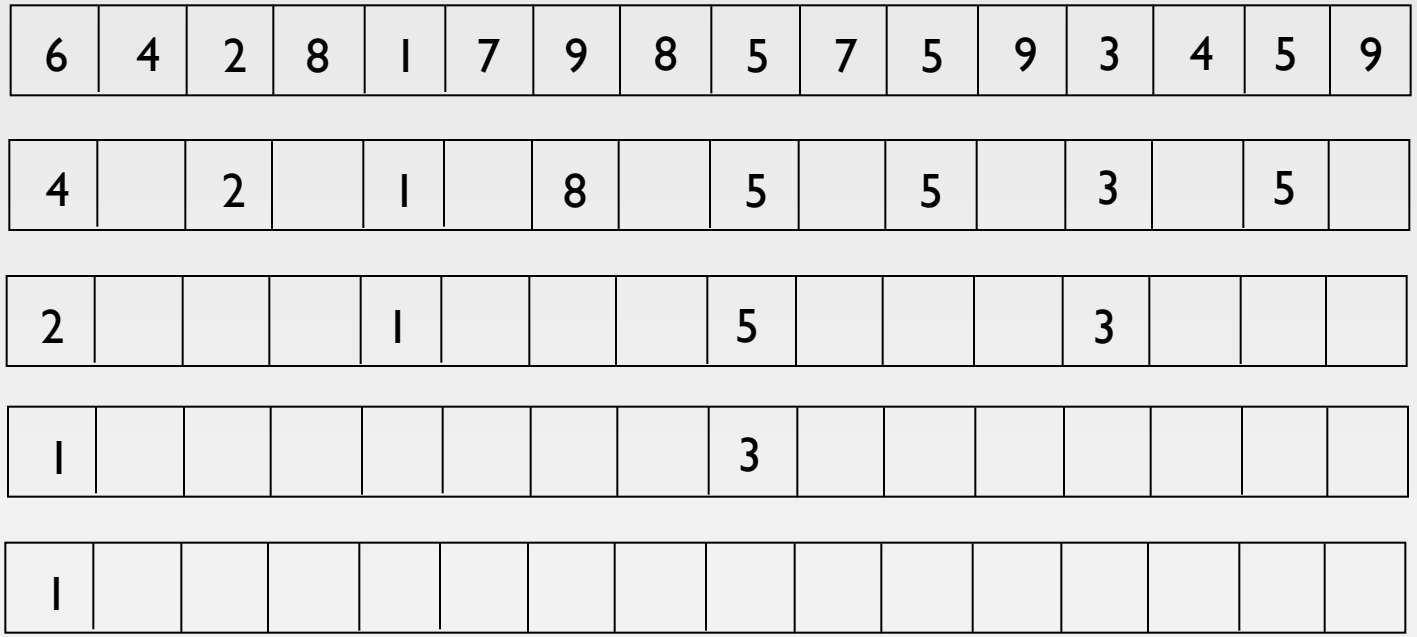
- Load feature neighborhood into shared memory
- Testing 256 disparities at once there is too much image data to fit into shared memory
- Use texture cache to retrieve texture for NCC calculation

Texture memory



Memory use for stereo match

- Store result in shared memory
- Use parallel min to find minimum NCC score
- Also match right to left. When L to R match agrees with R to L match we have a matched feature



CUDA Implementation

- Build Image Pyramid
- Detection
  - if(sufficient motion || first frame)
    - Calculate cornerness images
    - Perform non-maxima supression on cornerness
    - Match corners that are not near existing features
    - Triangulate new features
    - Remaining features as standard KLT features
- Perform multi-scale tracking update on all features
  - Standard KLT for features seen in one camera
  - Stereo KLT for features tracked in 3D
- NCC test for validation

Performance

- 50 fps on GTX 280 on 1024x768 video
- Extensive use of texture unit for access to hardware interpolation and the texture cache
- Performance is highly dependent on tuning block sizes to the particular GPU model being used
- Constant memory used to store Gaussian and DoG kernels as well as camera intrinsic and extrinsic calibration data used in calculating projection function Jaccobians

Real-Time Visual Odometry

Clipp, Zach, Frahm, Pollefeys ICCV'09

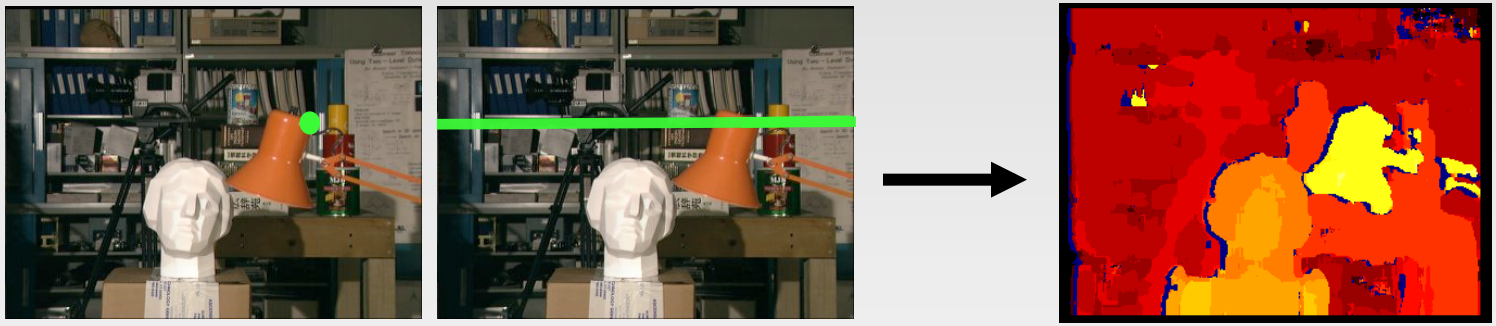
Two-view Stereo

David Gallup[1], Joe Stam[2] and Jan-Michael Frahm[1]

[1]Department of Computer Science, University of North Carolina at Chapel Hill
[2] Nvidia Corp

Stereo estimation

- Images are rectified
- Disparity search
  - along rows
  - with limited disparity range

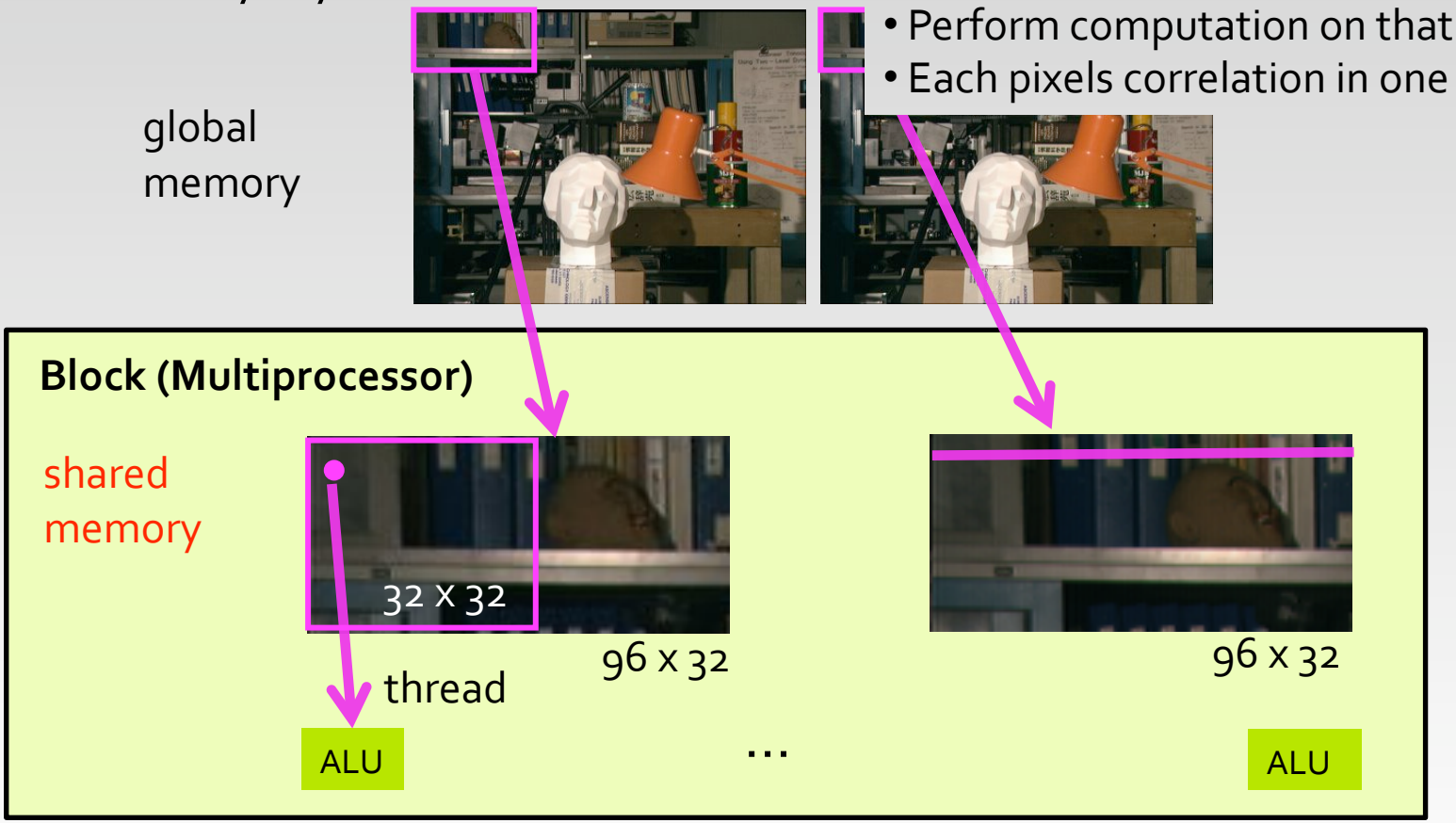


- Highly data parallel ⇒ fits well on GPU
- simple winner takes all strategy (SAD cost)
- faster than optimized shader implementations

CUDA Stereo

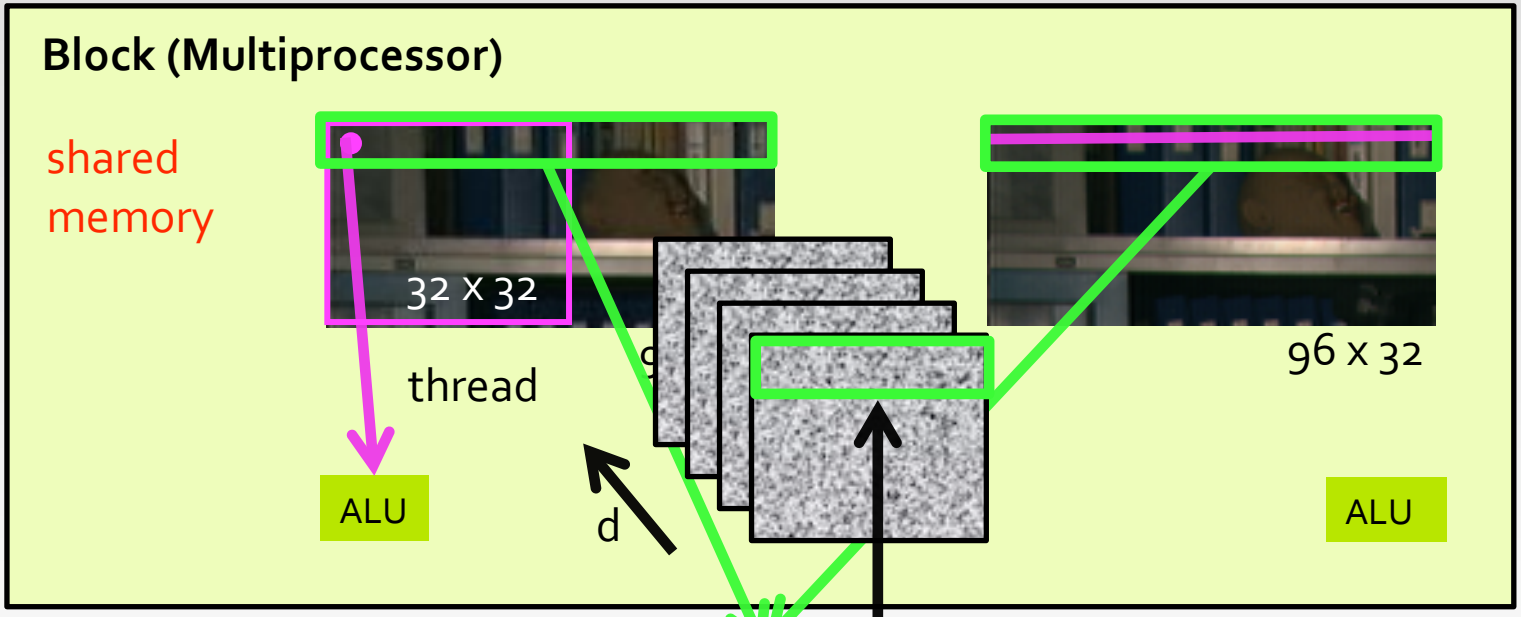
- Design decisions:
  - memory layout

Approach:
- Read block of global memory into shared memory.
- Perform computation on that block.
- Each pixels correlation in one thread



CUDA Stereo: Pixel Thread

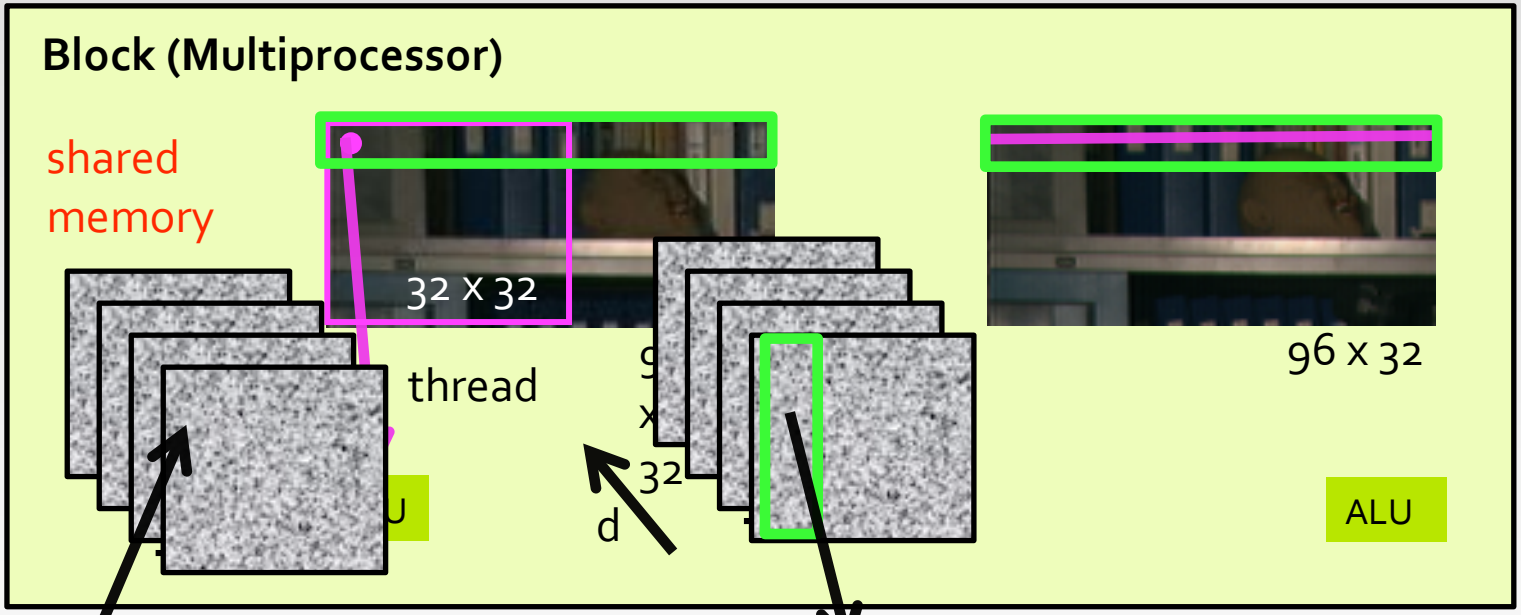
- Read image blocks from left and right images.
- For all disparities, compute SAD matching score, recording disparity with minimum score.



$$C_r(d,x,y) = \sum_{i \in W} \left| I(x+i,y) - I(x+i+d,y) \right|$$

CUDA Stereo

- Read image blocks from left and right images.
- For all disparities, compute SAD matching score, recording disparity with minimum score.

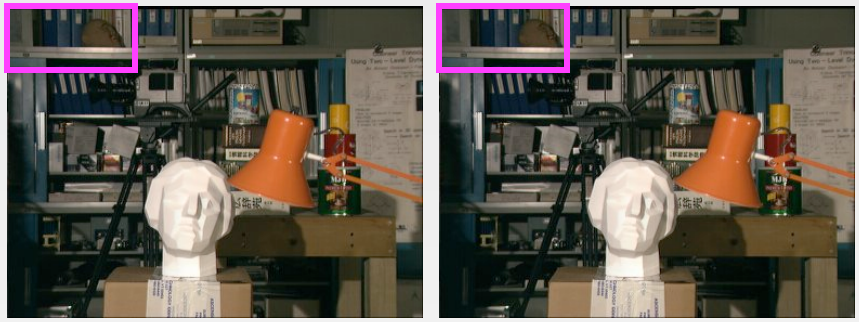


$$C(d,x,y) = \sum_{j \in W} \left| C_r(x, y+j) - C_r(x+d, y+j) \right|$$

CUDA Stereo

- Read image blocks from left and right images.
- For all disparities, compute SAD matching score, recording disparity with minimum score.
- Compute for left and right disparity maps

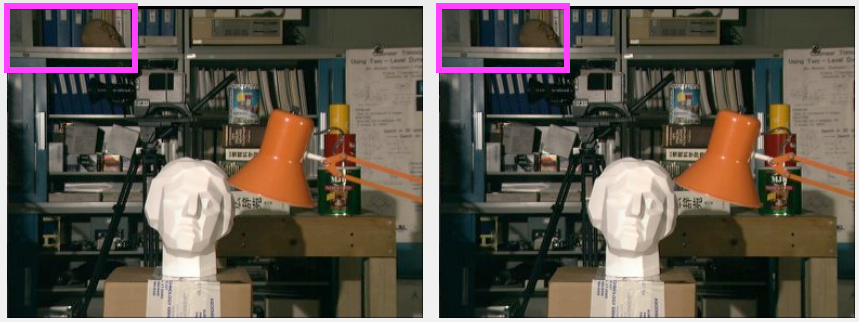
Global Memory



CUDA Stereo

- Read image blocks from left and right images.
- For all disparities, compute SAD matching score, recording disparity with minimum score.
- Compute for left and right disparity maps

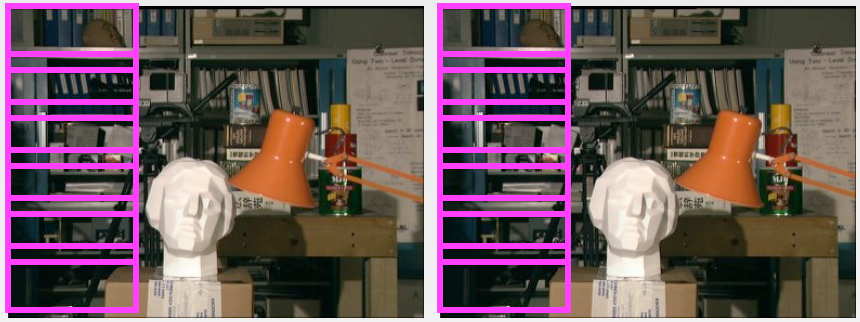
Global Memory



CUDA Stereo

- Read image blocks from left and right images.
- For all disparities, compute SAD matching score, recording disparity with minimum score.
- Compute for left and right disparity maps

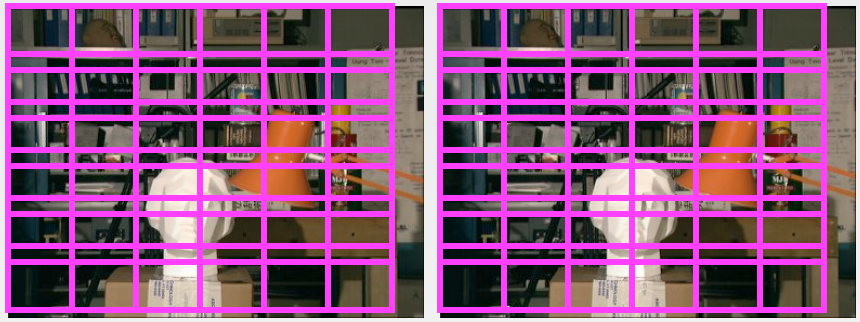
Global Memory



CUDA Stereo

- Read image blocks from left and right images.
- For all disparities, compute SAD matching score, recording disparity with minimum score.
- Compute for left and right disparity maps

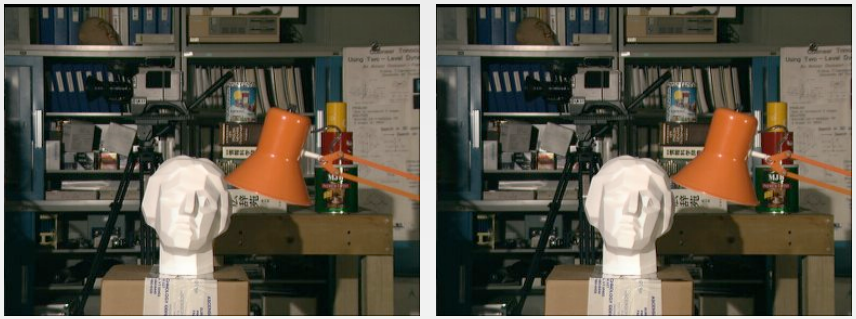
Global Memory

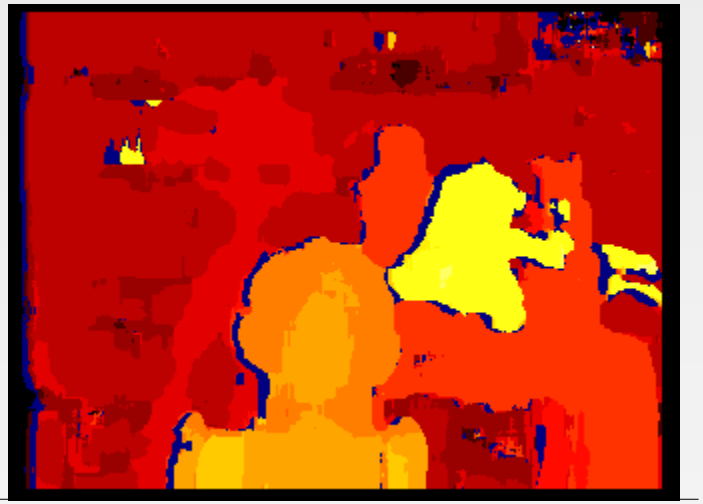


CUDA Stereo

- Detect occlusions and outliers by left/right depthmap consistency check.
  - If pixel x in left image corresponds to x' in right image, then x' should correspond to x, or one of its neighbors.

$$\left| D_L(x) - D_R(x + D_L(x)) \right| \leq C$$





CUDA Stereo

- Speed Results:
  - Binocular 640x480 grayscale images, 50 disparities, 11x11 matching window, left & right depthmaps with consistency check, upload and download
  - 46 ms (21.7 Hz)
  - Previous DirectX shader stereo: 50 ms (20.0 Hz)

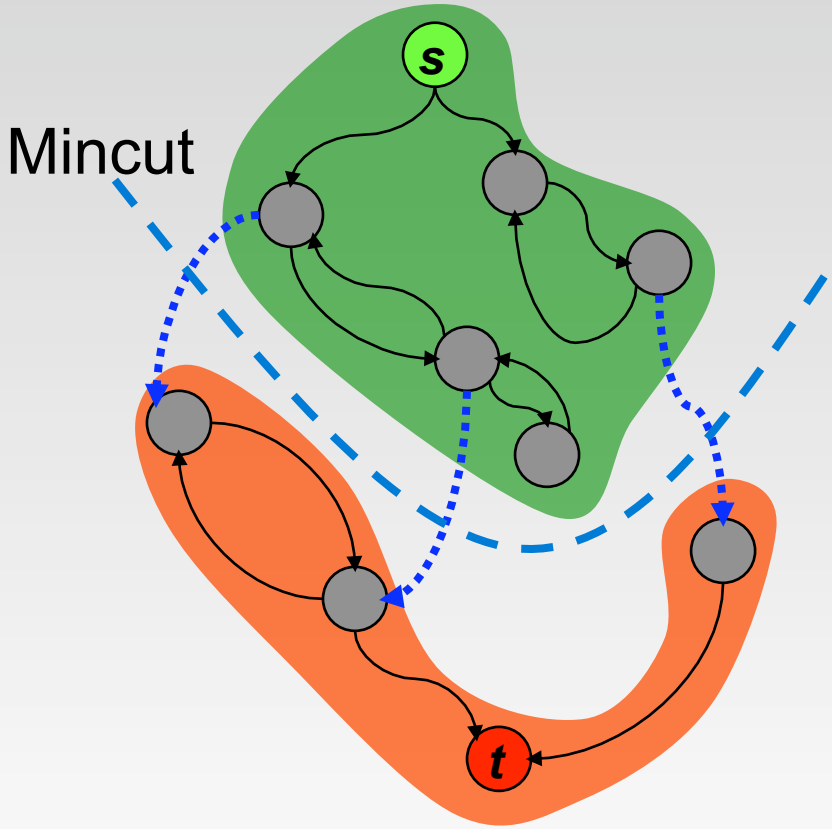
CUDA stereo

Vision Case Study: Graph Cuts

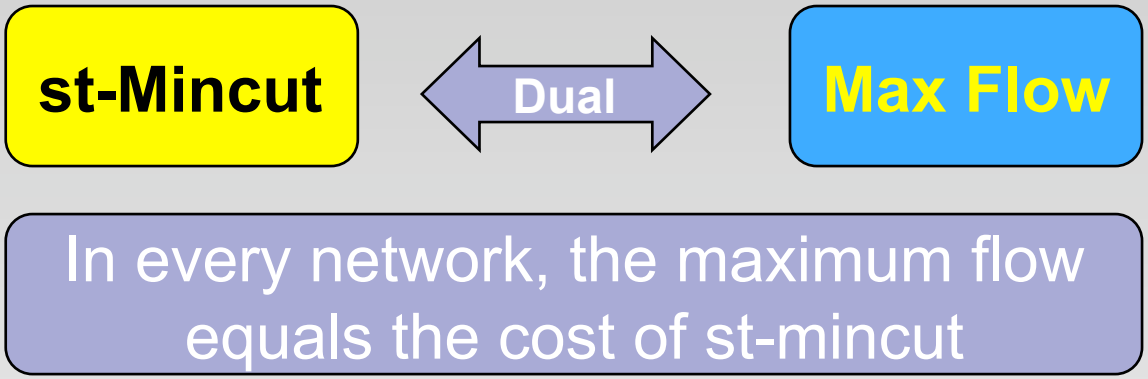
P.J. Narayanan
Centre for Visual Information Technology
IIIT, Hyderabad, India
(Work done with **Vibhav Vineet**)

The st-Mincut Problem

- Given a Graph ***G(V,E,W)*** and two vertices ***s*** and ***t***.
- Partition ***G*** into two disjoint components containing ***s*** and ***t*** respectively such that sum of edge weights from ***s*** to ***t*** is minimum

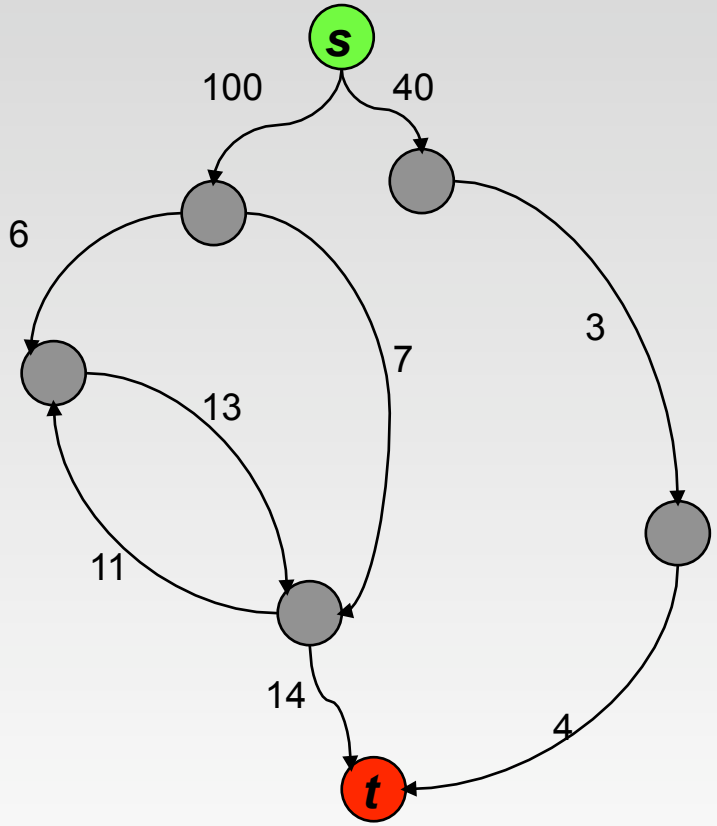


Computing the st-Mincut



- Solve the dual **Maximum Flow** problem
- Two approaches
  - Edmond Karp's Augmenting path method
  - Goldberg's Push-Relabel method

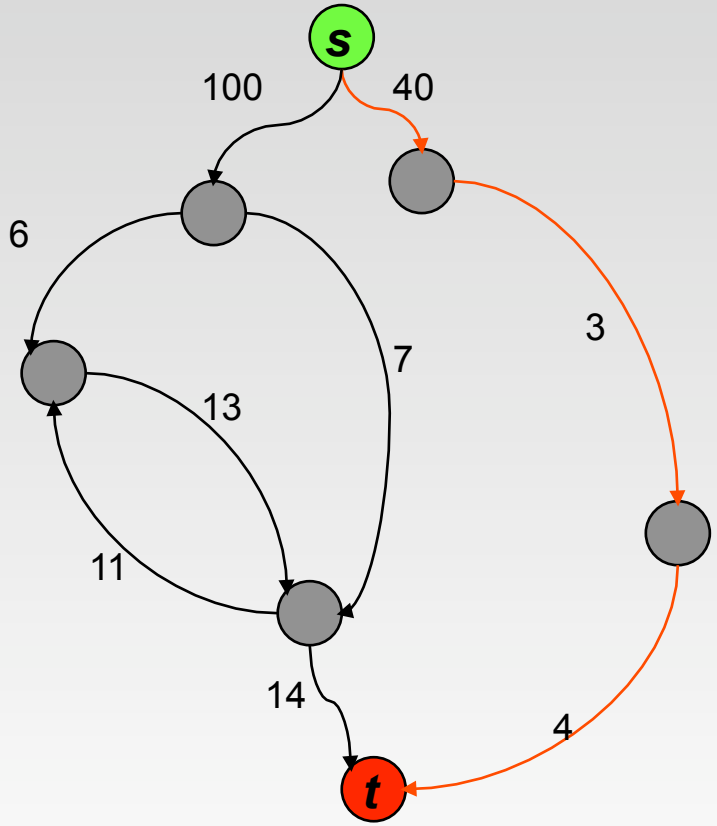
Edmond Karp Method



- Initialize flow in ***G*** to 0
- Find a shortest path from ***s*** to ***t***.
- Augment the path with minimum possible flow
- Repeat until there exists a path from ***s*** to ***t***

Current Flow: 0

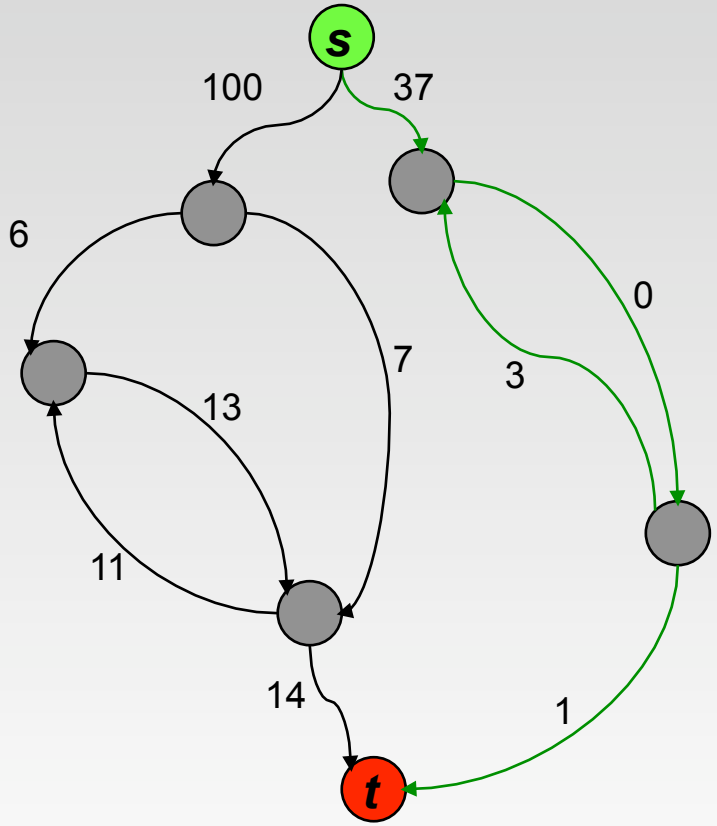
Edmond Karp Method



- Initialize flow in ***G*** to 0
- Find a shortest path from ***s*** to ***t***.
- Augment the path with minimum possible flow
- Repeat until there exists a path from ***s*** to ***t***

Current Flow: 0

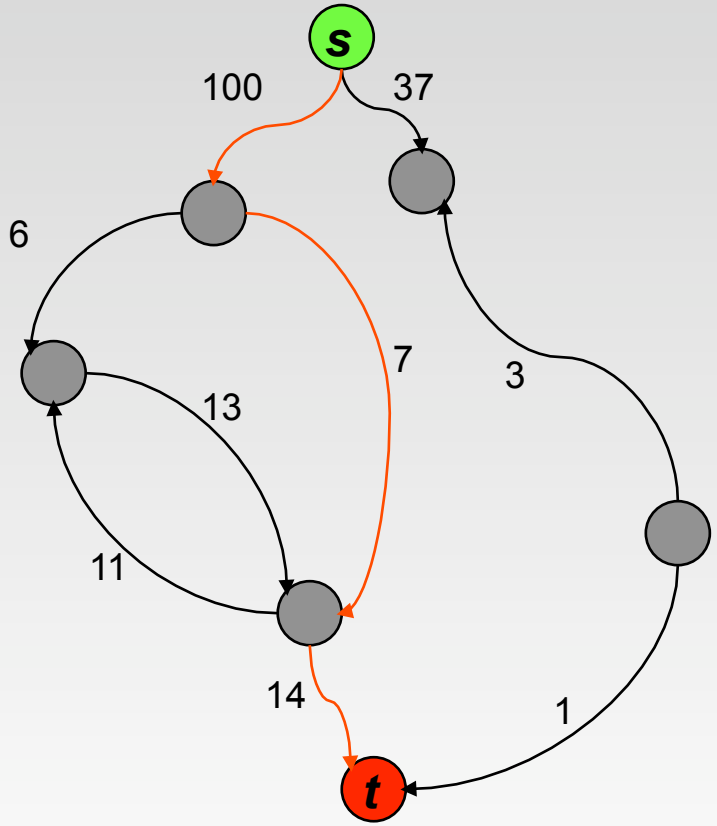
Edmond Karp Method



- Initialize flow in ***G*** to 0
- Find a shortest path from ***s*** to ***t***.
- Augment the path with minimum possible flow
- Repeat until there exists a path from ***s*** to ***t***

Current Flow: 3

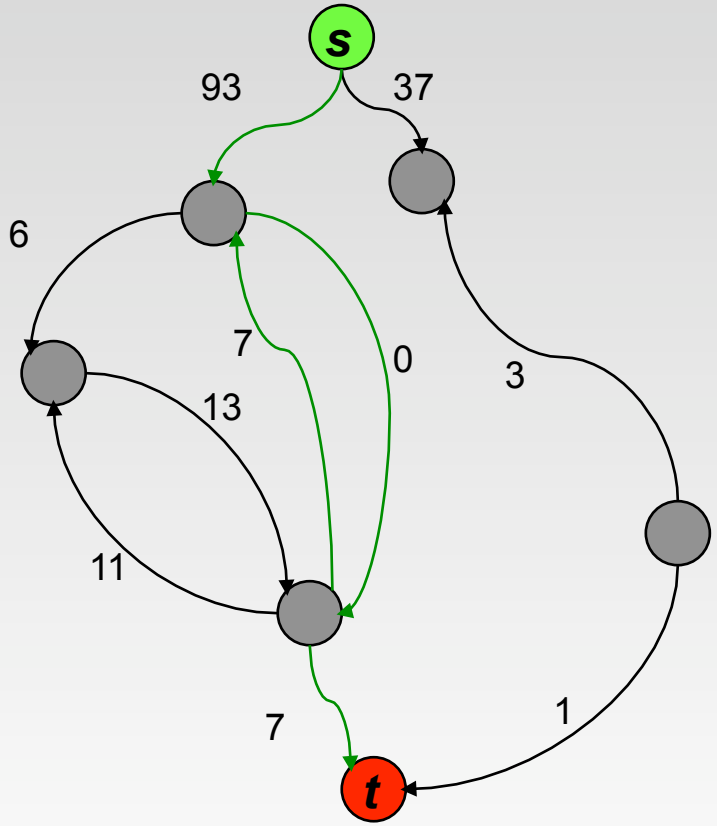
Edmond Karp Method



- Initialize flow in ***G*** to 0
- Find a shortest path from ***s*** to ***t***.
- Augment the path with minimum possible flow
- Repeat until there exists a path from ***s*** to ***t***

Current Flow: 3

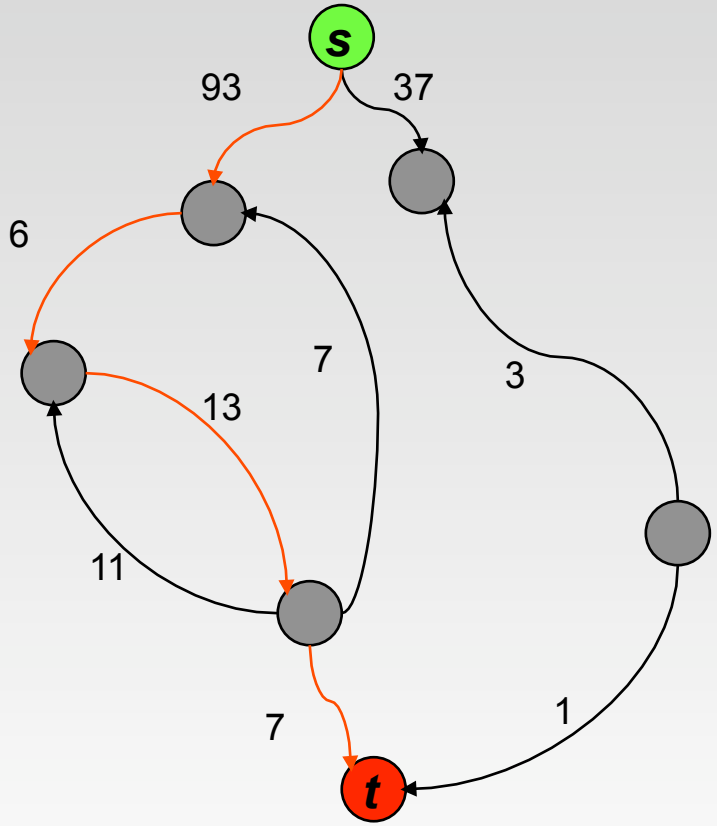
Edmond Karp Method



- Initialize flow in ***G*** to 0
- Find a shortest path from ***s*** to ***t***.
- Augment the path with minimum possible flow
- Repeat until there exists a path from ***s*** to ***t***

Current Flow: 10

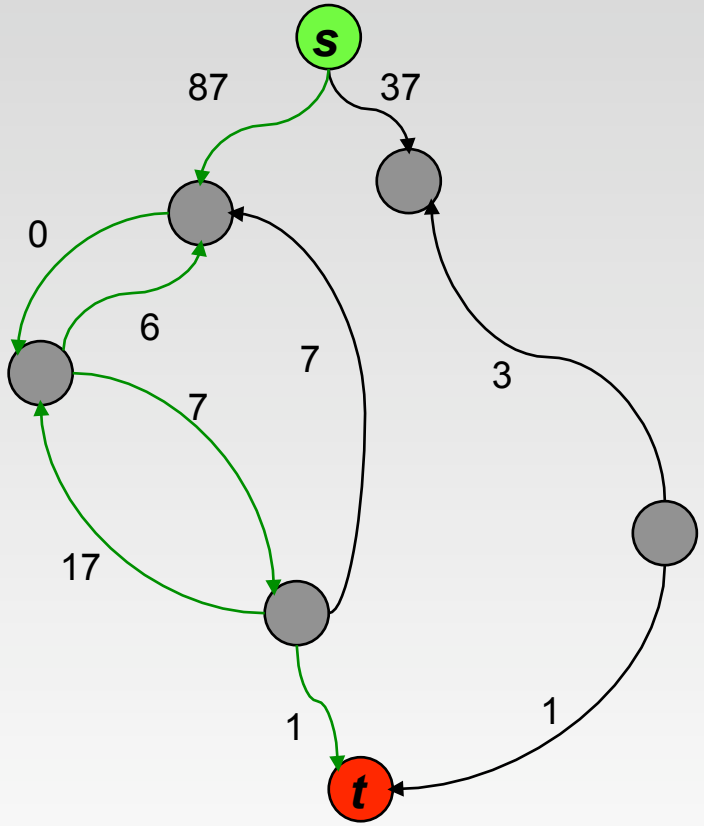
Edmond Karp Method



- Initialize flow in ***G*** to 0
- Find a shortest path from ***s*** to ***t***.
- Augment the path with minimum possible flow
- Repeat until there exists a path from ***s*** to ***t***

Current Flow: 10

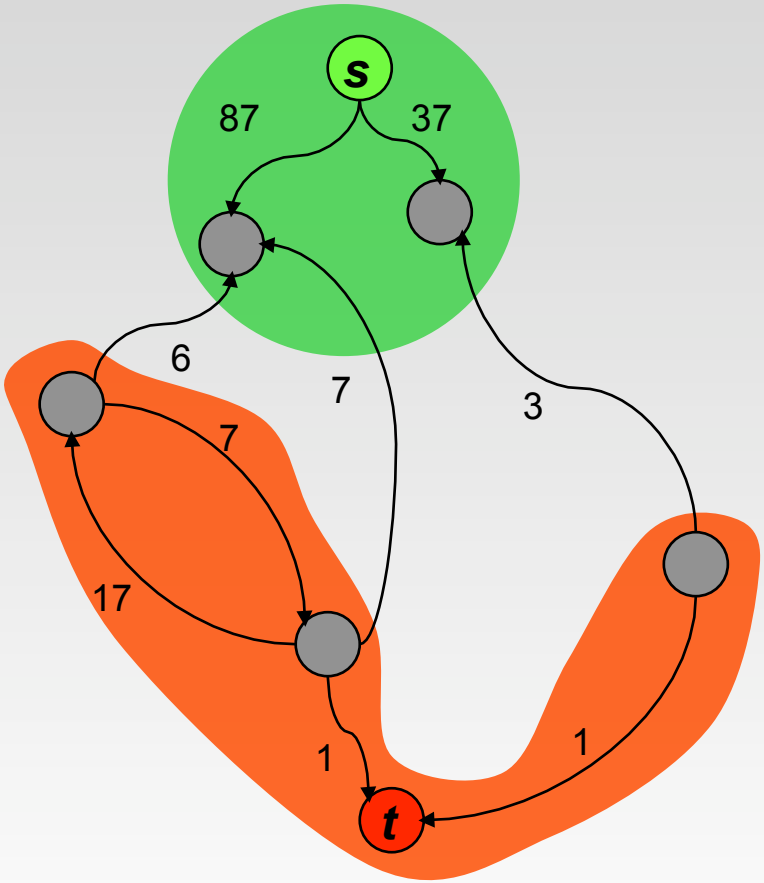
Edmond Karp Method



- Initialize flow in ***G*** to 0
- Find a shortest path from ***s*** to ***t***.
- Augment the path with minimum possible flow
- Repeat until there exists a path from ***s*** to ***t***

Current Flow: 16

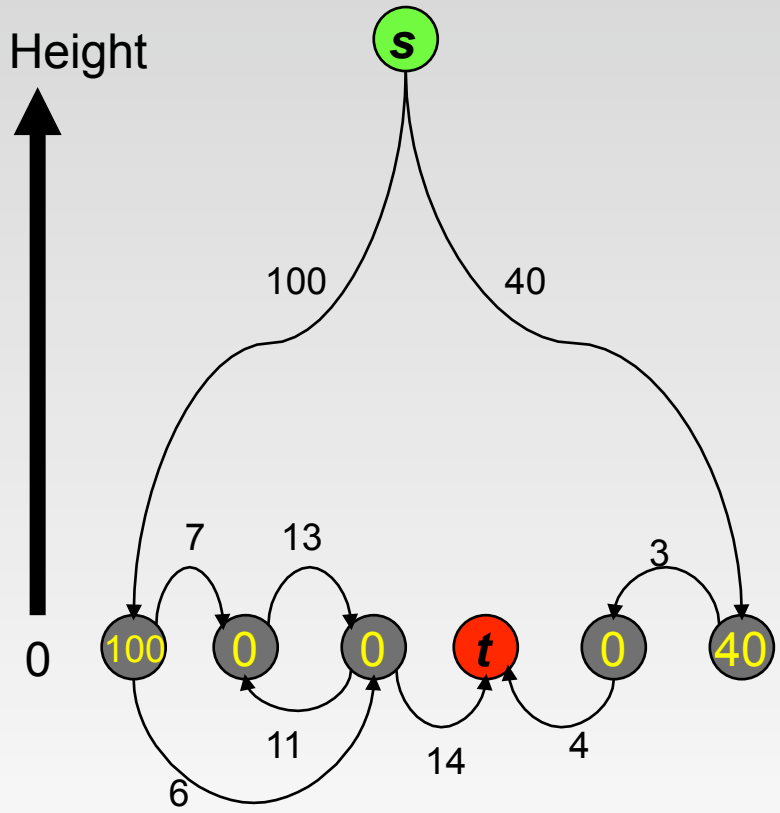
Edmond Karp Method



- Initialize flow in ***G*** to 0
- Find a shortest path from ***s*** to ***t***.
- Augment the path with minimum possible flow
- Repeat until there exists a path from ***s*** to ***t***

Current Flow: 16

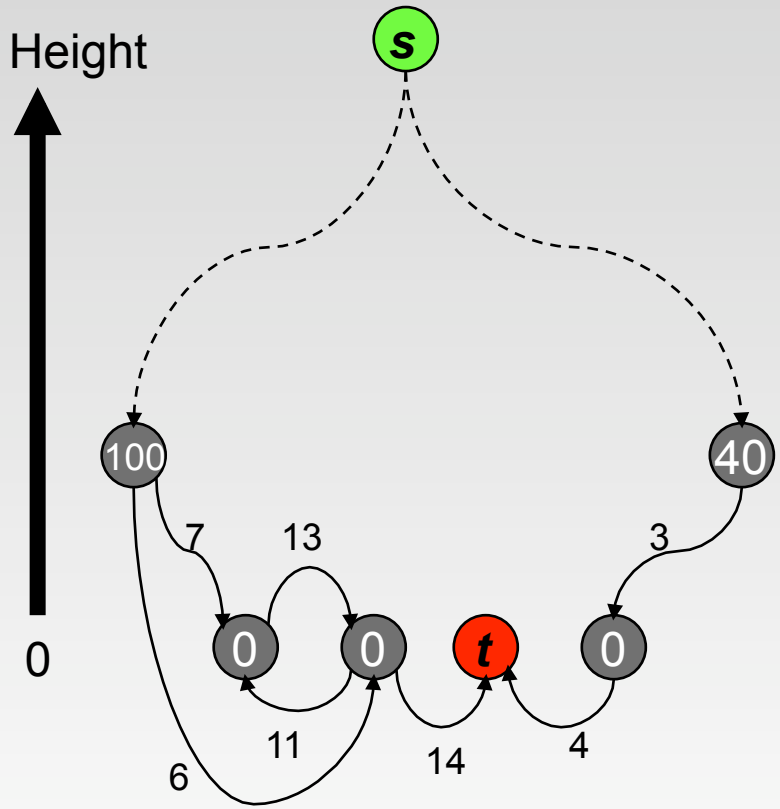
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable Push or Relabel operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 0

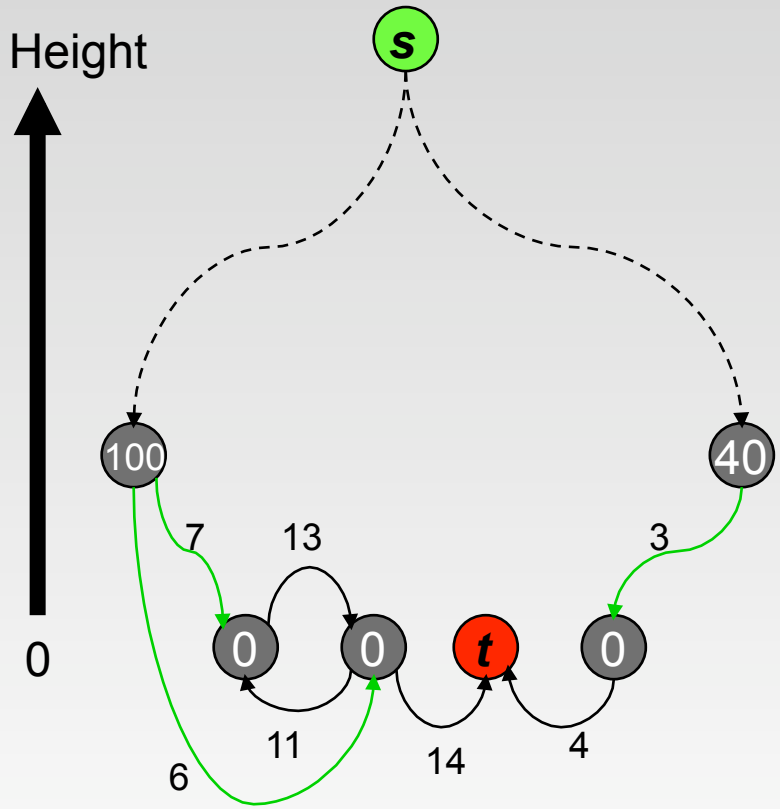
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable Push or **Relabel** operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 0

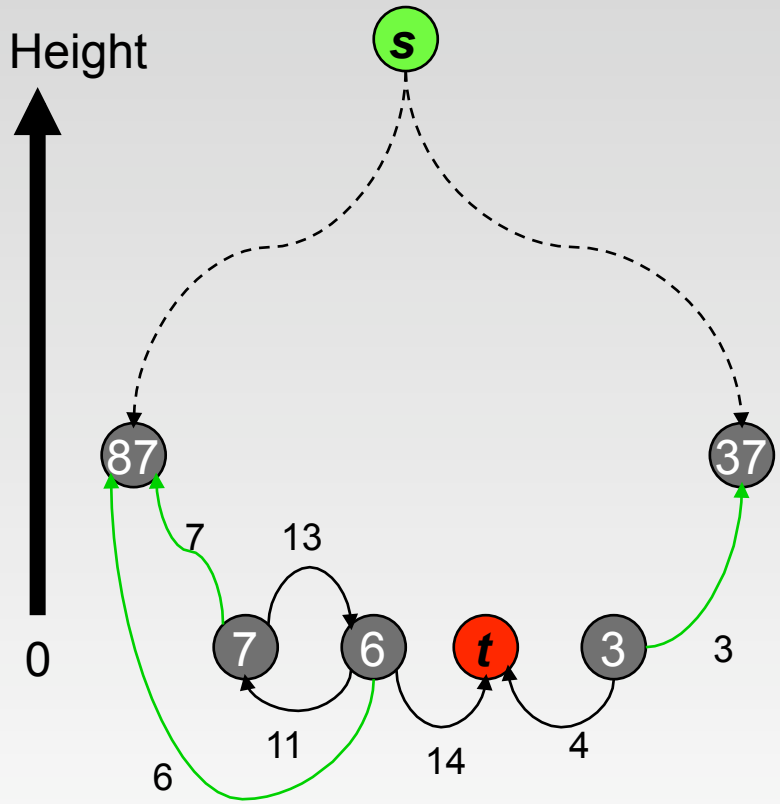
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable **Push** or Relabel operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 0

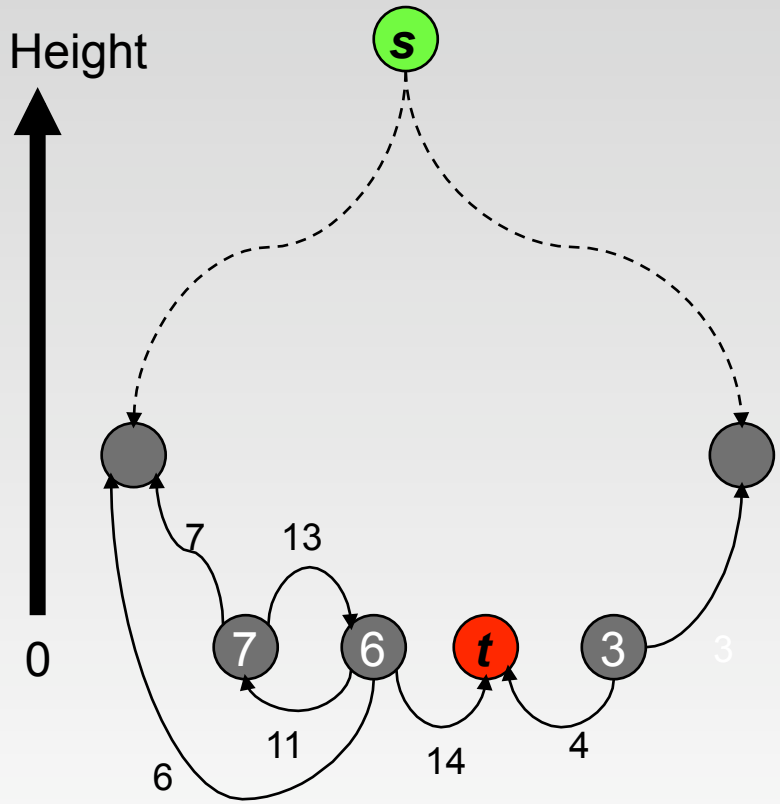
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable **Push** or Relabel operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 0

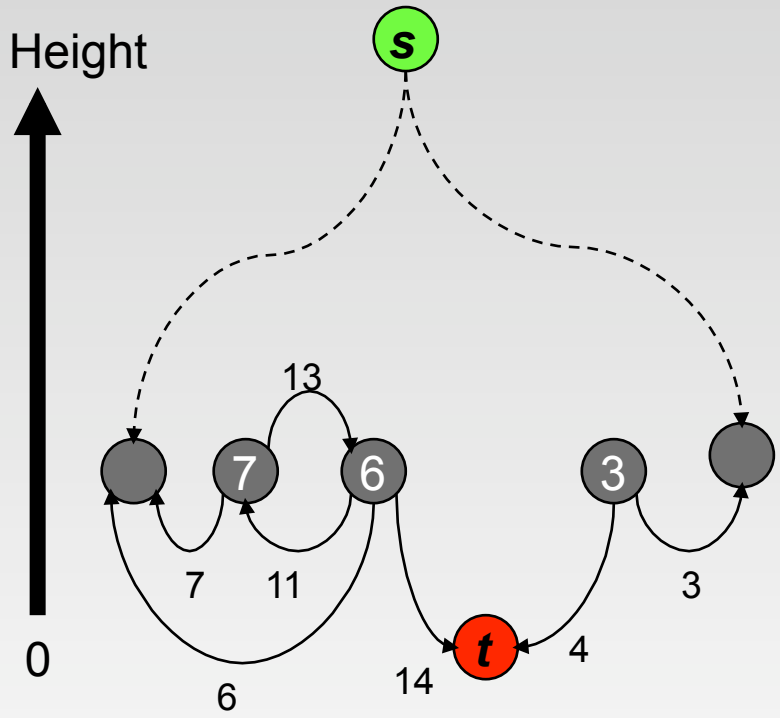
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable Push or Relabel operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 0

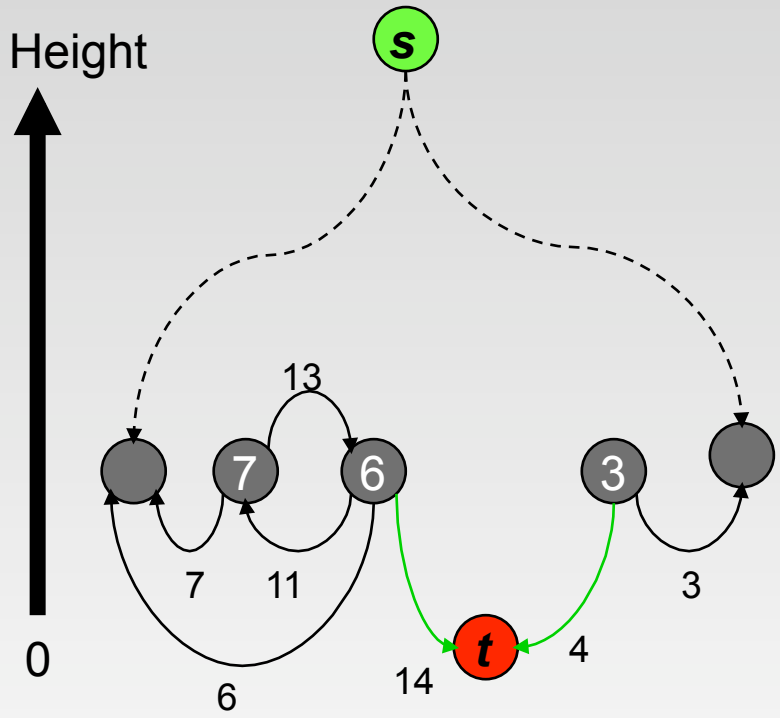
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable Push or **Relabel** operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 0

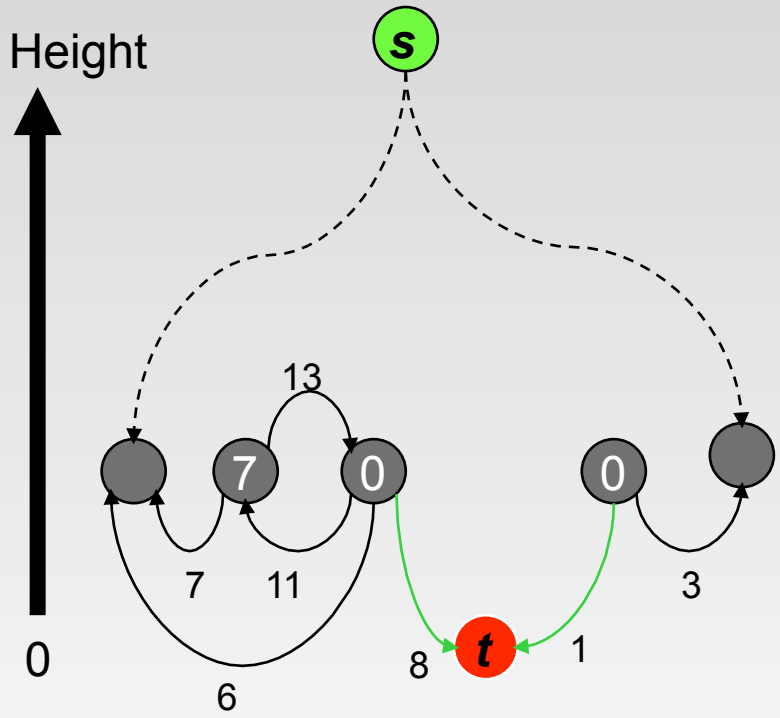
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable **Push** or Relabel operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 0

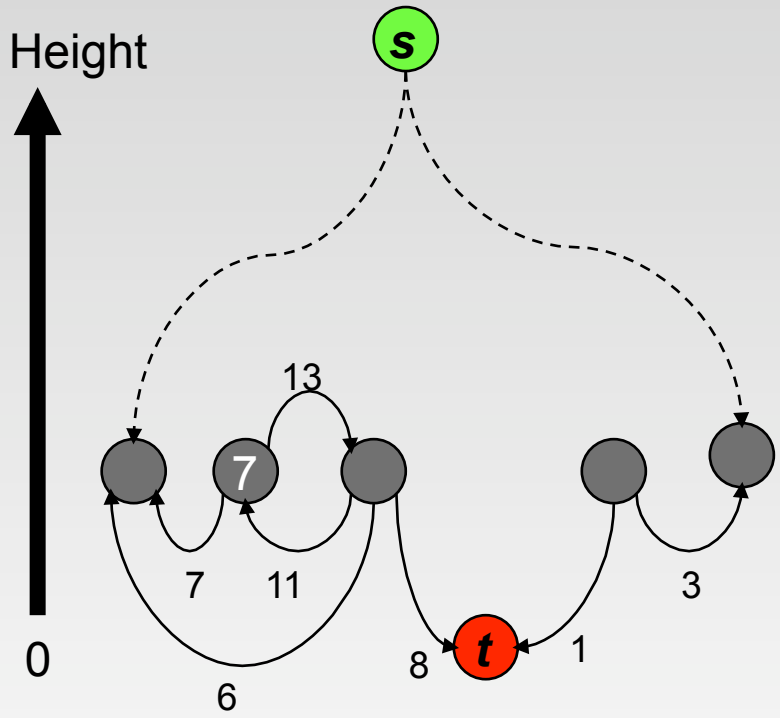
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable **Push** or Relabel operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 9

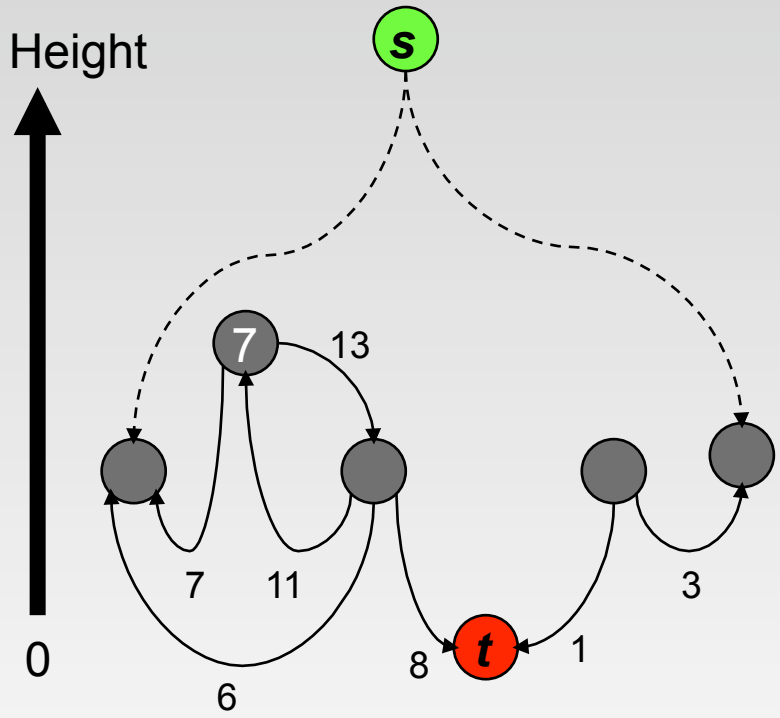
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable Push or Relabel operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 9

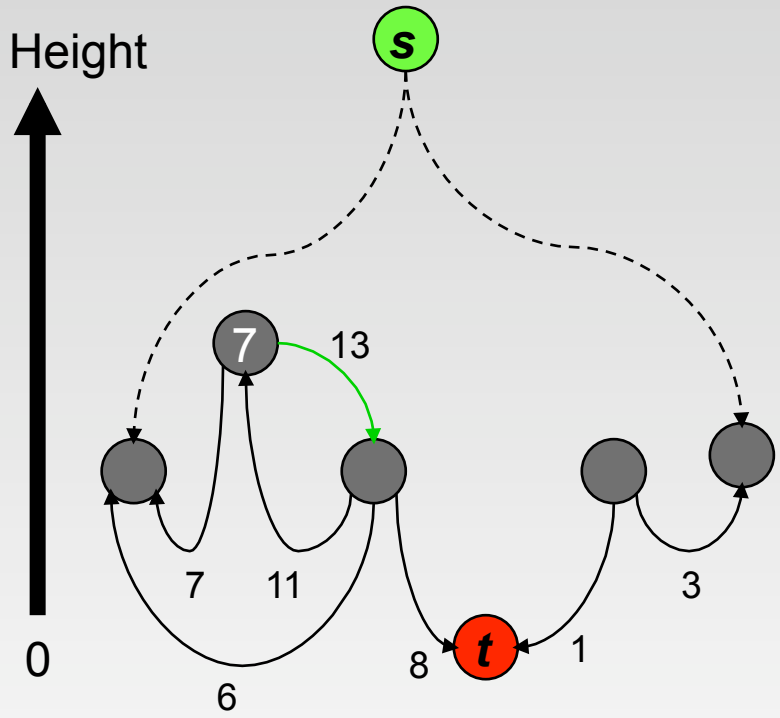
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable Push or **Relabel** operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 9

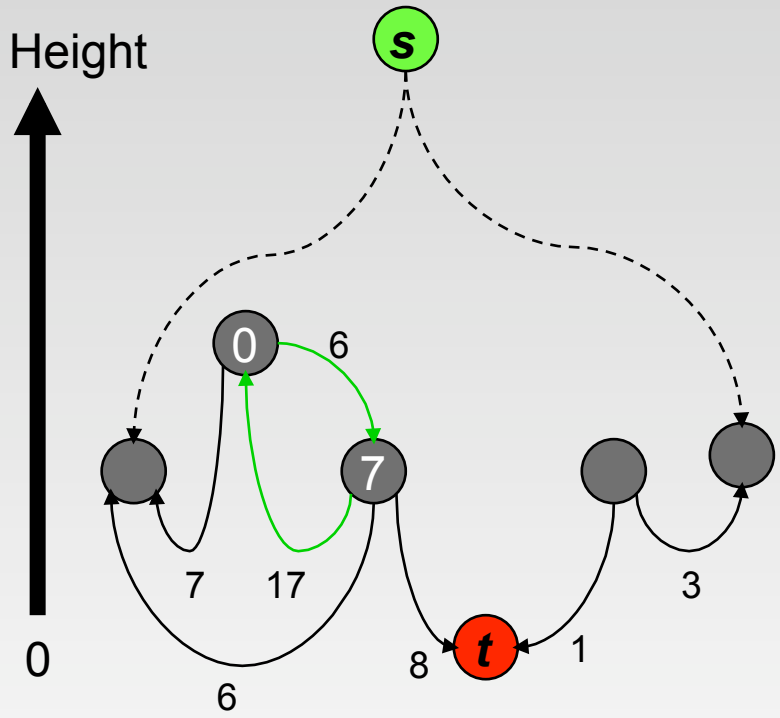
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable **Push** or Relabel operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 9

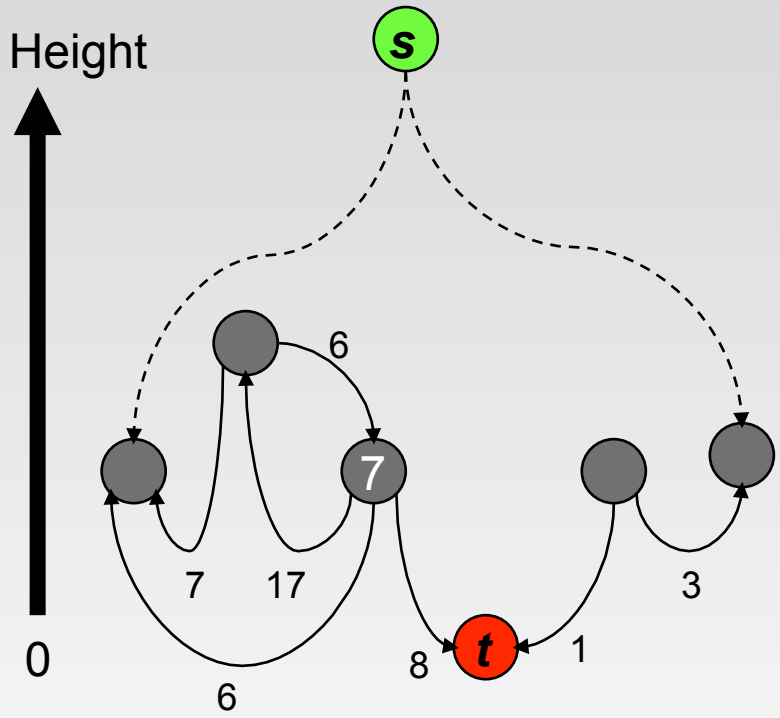
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable **Push** or Relabel operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 9

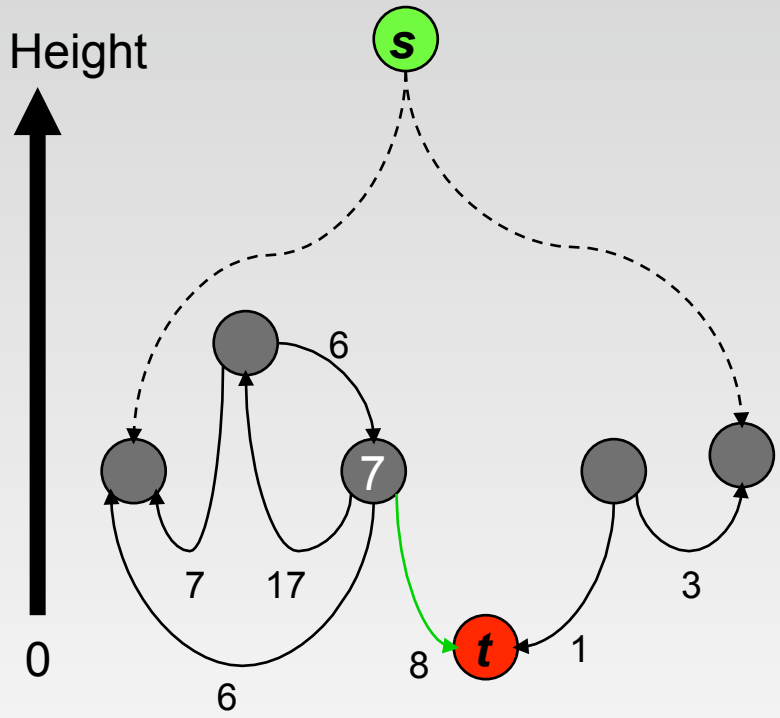
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable Push or Relabel operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 9

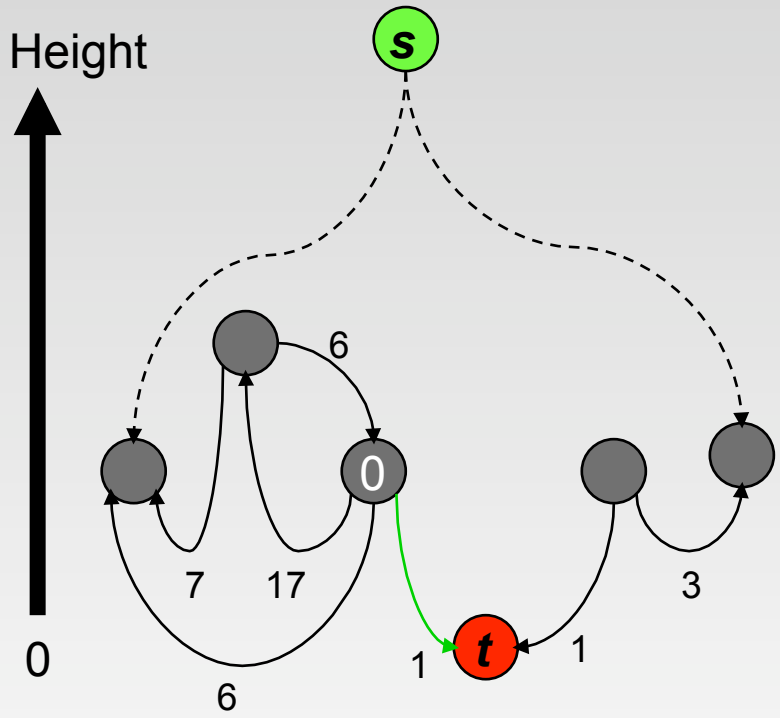
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable **Push** or Relabel operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 9

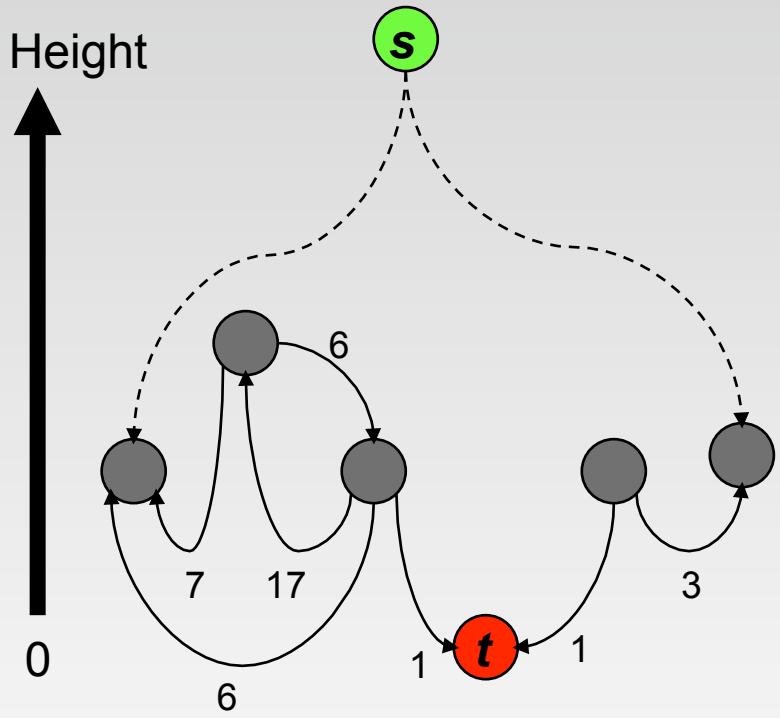
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable **Push** or Relabel operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 16

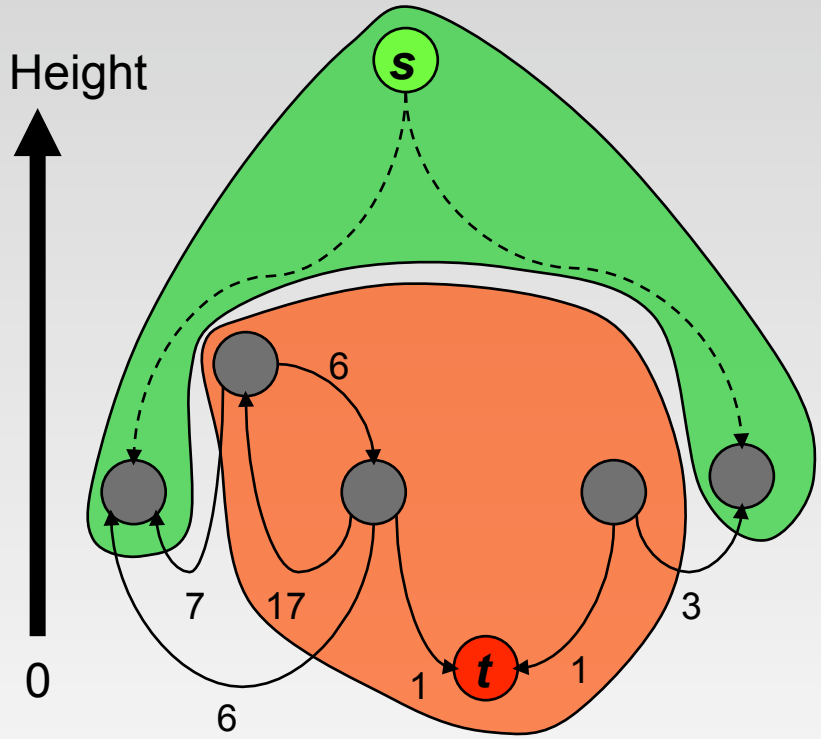
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable Push or Relabel operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 16

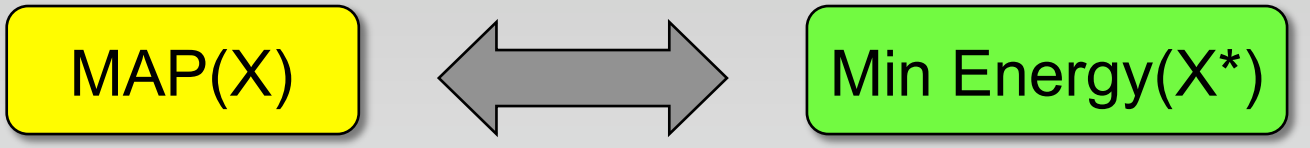
Goldberg's Push-Relabel Algorithm



- Initialize excess flow and heights in ***G***
- Perform an applicable Push or Relabel operation
- Repeat until there exists an applicable push or relabel operation

Current Flow: 16

MAP-MRF Formulation

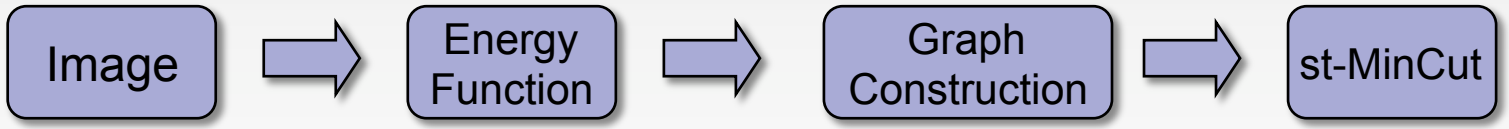


MAP estimation of a configuration X is equivalent to the minimum energy defined over the configuration

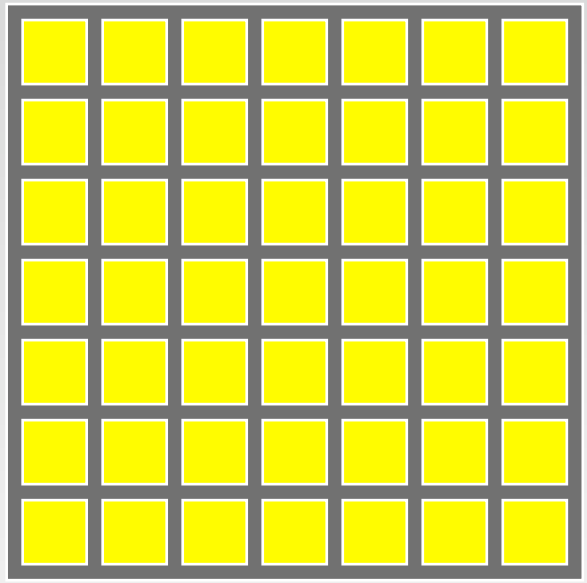
$$E_p(x) = D_p(x_p) + V_{\{p,q\}}(x_p, x_q)$$

Energy = Data Term + Smoothness Term

Graph Cuts in Computer Vision



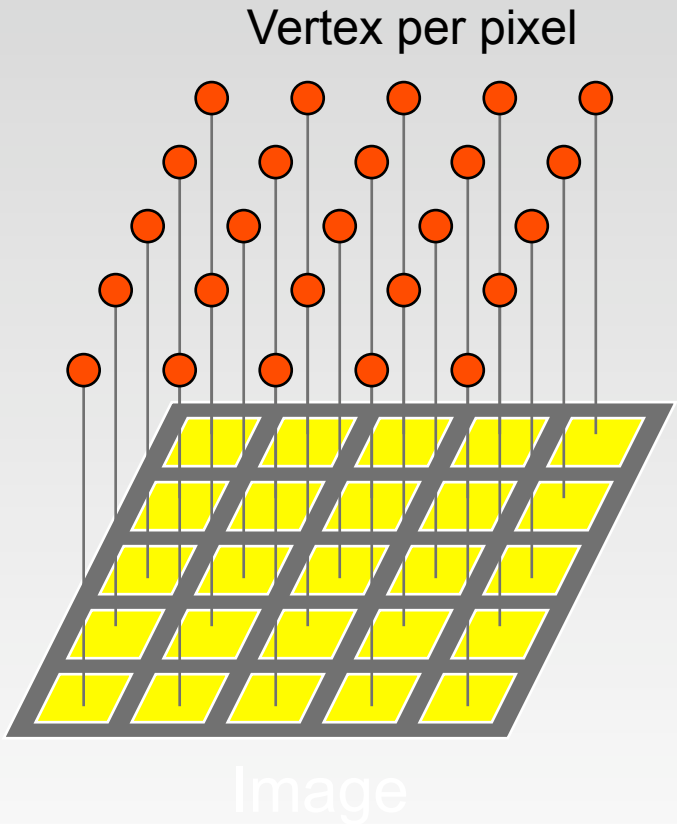
Graph Construction



Image

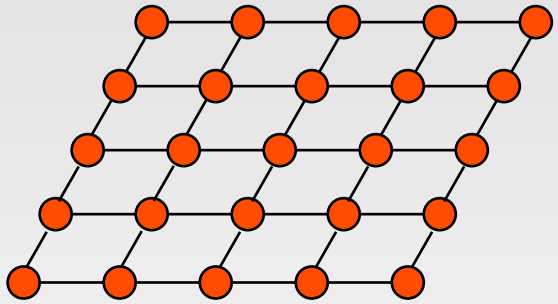
- Graphs constructed for vision problems
  - Grid graphs
  - Low connectivity
  - Connectivity is limited to 4, 8 or 27

Graph Construction



- Graphs constructed for vision problems
  - Grid graphs
  - Low connectivity
  - Connectivity is limited to 4, 8 or 27

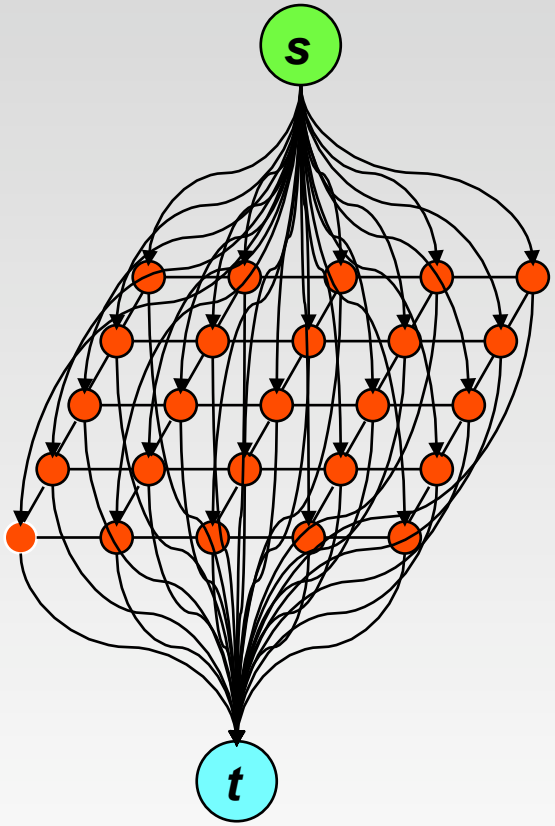
Graph Construction



- Graphs constructed for vision problems
  - Grid graphs
  - Low connectivity
  - Connectivity is limited to 4, 8 or 27

Add n-edges / Lateral Edges

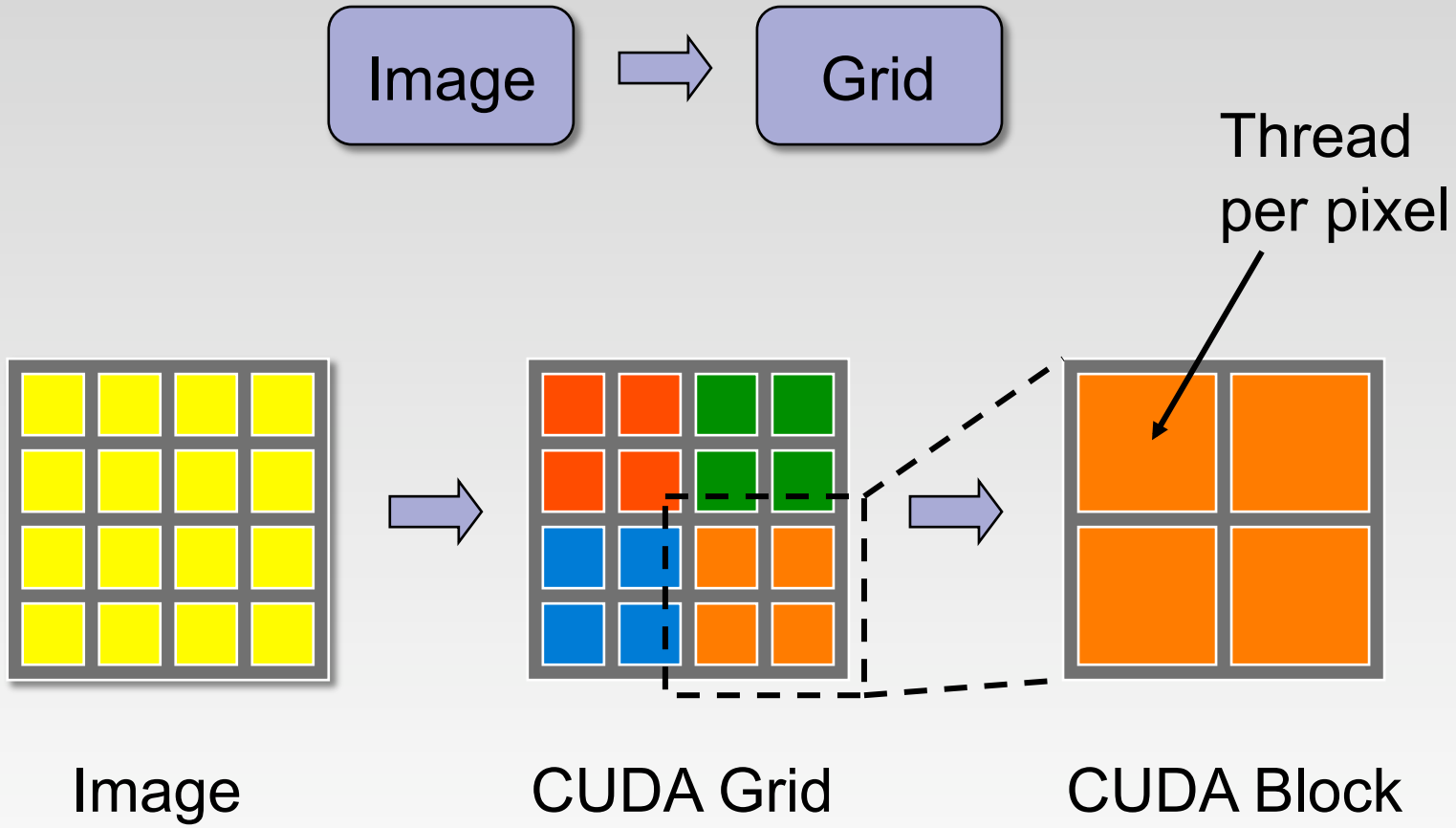
Graph Construction



- Graphs constructed for vision problems
  - Grid graphs
  - Low connectivity
  - Connectivity is limited to 4, 8 or 27

Add t-edges / Vertical Edges

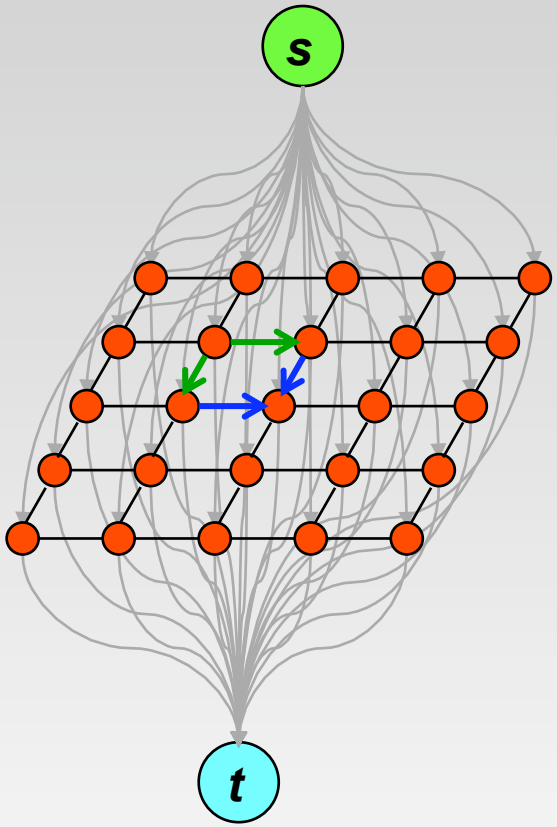
Mapping to CUDA



Push-Relabel Algorithm on CUDA

- Push is a local, independent operation
  - Each node sends flows to its neighbors without violating capacity constraints
  - Flows coming into each node are accumulated
- Relabel is also a local operation
  - Each vertex updates its own height based on its neighbours
- Problems:
  - Possible Read-After-Write inconsistency
  - Synchronization of push, relabel operations

Handling problems without Atomics

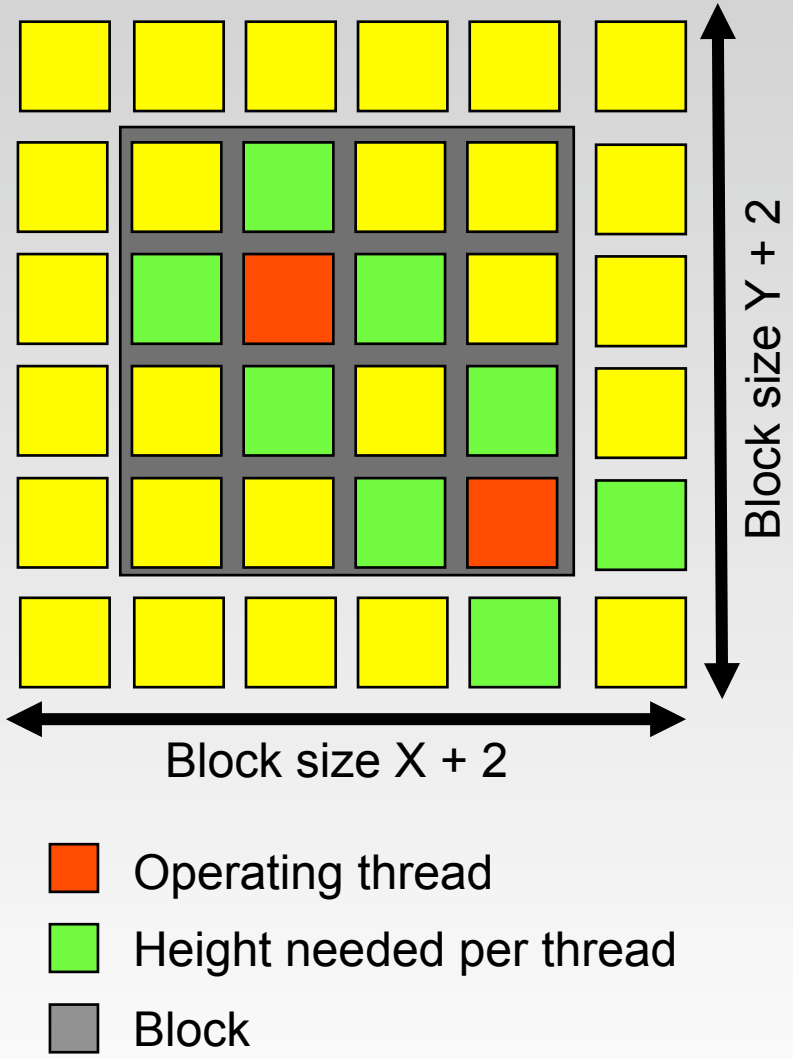


- Push is divided into two phases: **Push** and **Pull**
- Relabel is applied later
- Naïve Solution:
  - Push Kernel. All nodes push flows along all possible edges. Store them in different variables at the destination without conflict.
  - Pull Kernel collects flows and updates excess flow values
  - Relabel Kernel updates heights

Pull and Relabel Kernels can be combined into one kernel

Data in Shared Memory

- Each CUDA block handles *m x n* nodes. And apron values
- Each node checks heights of its neighbours before pushing
  - Push allowed only to a node 1 less in height
  - Each value needed by 4 neighbours
- Store heights in the shared memory to optimize global memory reads
- Edge capacities can be loaded when needed



The Push Kernel

Do for each pixel-node in parallel

- Load heights from the global memory to shared memory for the block. Each thread reads its own. Each border thread reads a neighbour's also.
- Synchronize threads of a block to ensure completion of loading heights
- Push flow to all eligible nodes. Update capacities of edges in the residual graph.
- Store the flow pushed to each vertex in a location specific to the direction, in the global memory

The Pull Kernel

- Read and add the flows pushed from all neighbors
- Compute the final excess flow by aggregating all incoming flows
- Store it as excess flow in the global memory

The Relabel Kernel

- Load height from the global memory to the shared memory.
- Synchronize threads to ensure the completion of load operation.
- Compute the minimum height of all neighbors and set own height to plus one of this.
- Write new height to global memory.

Push-Relabel using Atomics

- Push and Pull operations can be performed without any read after write inconsistencies using atomic add operations
  - Push + Pull Kernel: Add to the excess flow of each neighbour atomically when pushing to it.
  - Relabel Kernel: As before, a per vertex operation
- Lowers global memory accesses.
- Faster convergence is observed.

The Push + Pull Kernel

- Load heights from the global memory to the shared memory.
- Synchronize threads ensuring the completion of load operation.
- Push flows to eligible neighbors and update excess flows **atomically** in the residual graph.
- Update the edge-weights **atomically** in the residual graph.

Push and Relabel

- Multiple push operations can be performed before a relabel
  - For most grid graphs, 3 push operations before a relabel gives good performance
- Global relabel heuristic is known to improve performance
- BFS-based global relabelling is very expensive on grid graphcs
- For general graphs, global relabelling helps. Typically done once every 7 or so rounds of local relabelling

Stochastic Cuts

- MRF consists of *simple* and *difficult* pixels.
  - *Simple* pixels get their correct labels in few initial iterations
  - *Difficult* pixels exchange flows with their neighbors in later iterations
- Stochastic Cuts: process pixels based on their activity
- Activity is defined based on the change in flows from previous to current iteration. Low activity is observed for simple pixels
- We measure the overall activity in a block of pixels
  - Active blocks are processed every iteration
  - Others are processed every *k* iterations
  - Improves utilization of the GPU hardware and increases speed

Experimental Results

Experimental Results

Image	Size	Time CPU (ms)	Time Non Atomic (ms)	Time Atomic (ms)	Time Stochastic (ms)
Sponge	640x480	142	28	16	11
Flower	608x456	188	33	26	16
Person	608x456	140	31	27	20
Synthetic	1Kx1K	655	19	10	7

Times on an Nvidia GTX280

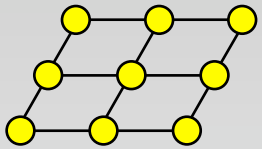
Dynamic Cuts

- Exploiting temporal coherence across frames in a video.
- Few pixels change values between consecutive frames.
- Better initialization for graph cut for the next frame
- Faster convergence
- Steps involved
  - Updation
  - Re-parameterization

Kohli, Torr. ICCV 2005

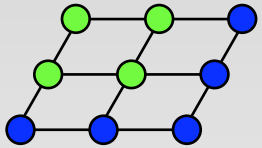
Dynamic Cuts Steps Involved

c_i



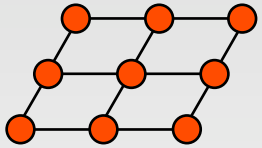
Previous Frame

f_i
r_i



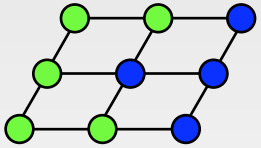
Previous Frame after st-MinCut

c_i'

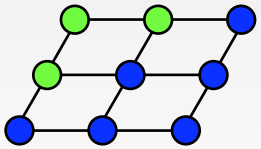


Current Frame

r_i'
f_i'



Approximate cut using previous frame and its st-MinCut



Final st-MinCut of current frame

- Edge capacities are updated and reparameterized using
  - Previous frame edge capacities
  - Previous frame residual flow
  - Current frame edge capacities

Updation Step:

$$r_i' = r_i + c_i' - c_i$$

Reparameterization Step:

$$r_{si}' = 0$$

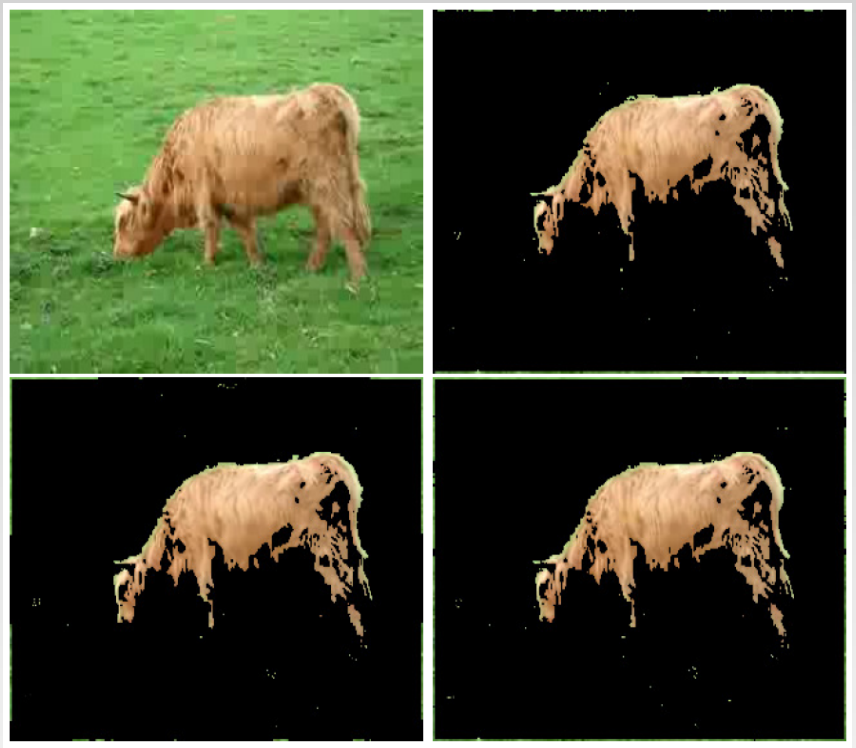
$$r_{it}' = c_{it} - f_{it} + f_{si} - c_{si}'$$

Dynamic Cuts are parallizable

- Updation and Reparameterization are independent and parallizable operations, work locally at every vertex.

- st-Mincut is performed using a parallel implementation of Push Relabel algorithm.

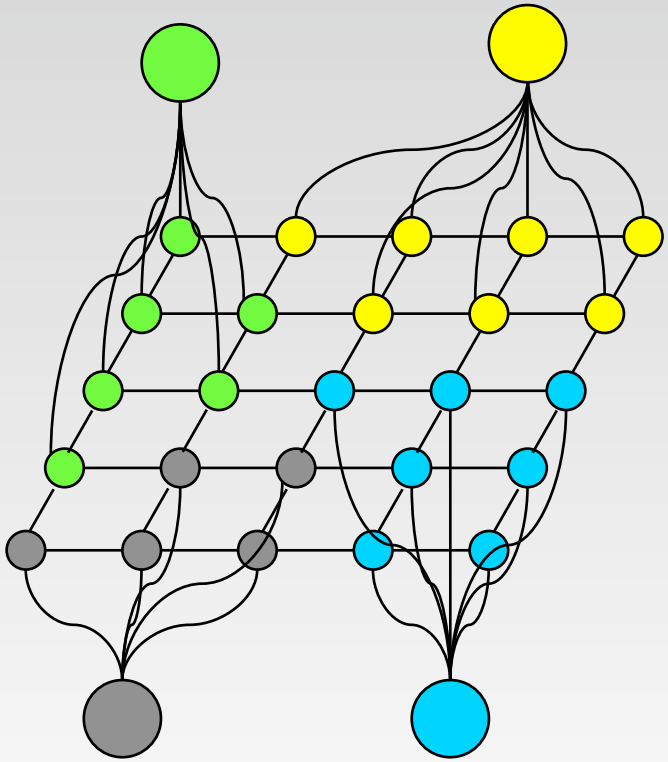
Dynamic Cuts Empirically



Consecutive frames of a video segmented using dynamic cuts

- Running time depends on the percentage of weights that changed
- On a low resolution video, the dynamic cuts takes about 2 ms compared to 7 ms on the same image for the st-MinCut

The Multilabeling problem



- Multi-way cut on any graph is an NP-Hard problem for $L > 2$
- Approximate solutions based on graph cuts
  - α-Expansion
  - αβ-Swap

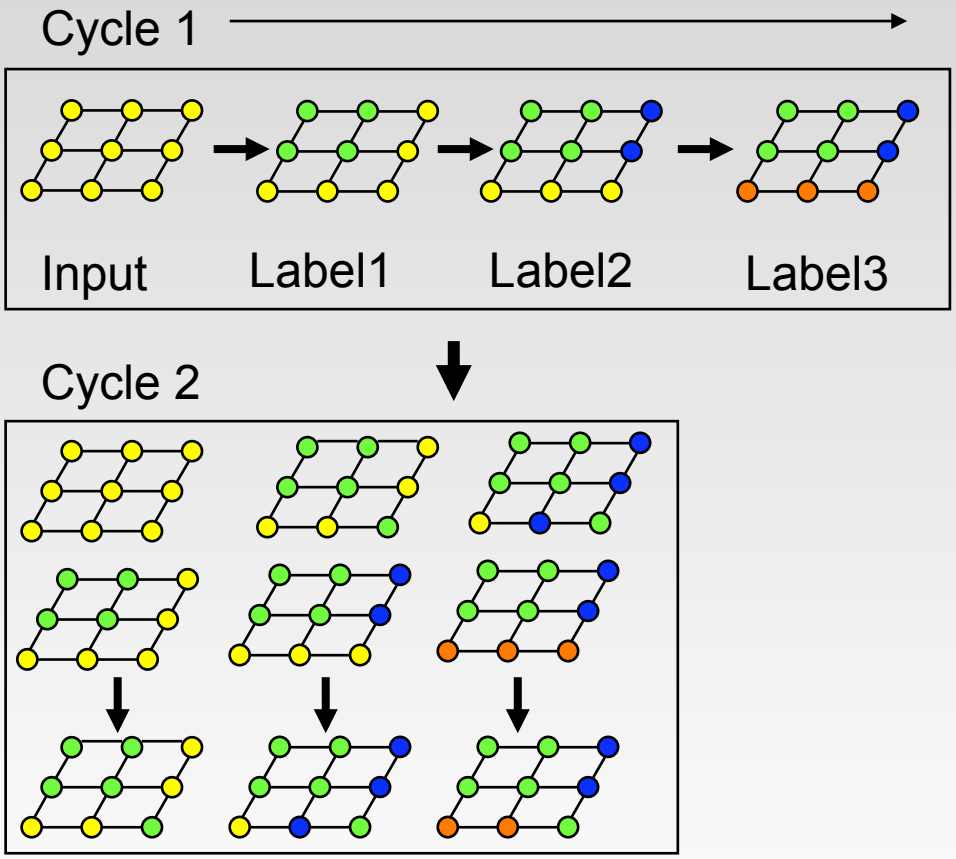
The α-Expansion

- Construct the graph based on the energy function
- Start with an arbitrary configuration of MRF.
- Select a label α
  - Perform one α-Expansion step (st-cut)
  - Update the configuration if the energy decreases
- Stop when there is no decrease in energy

α-Expansion Graph Construction

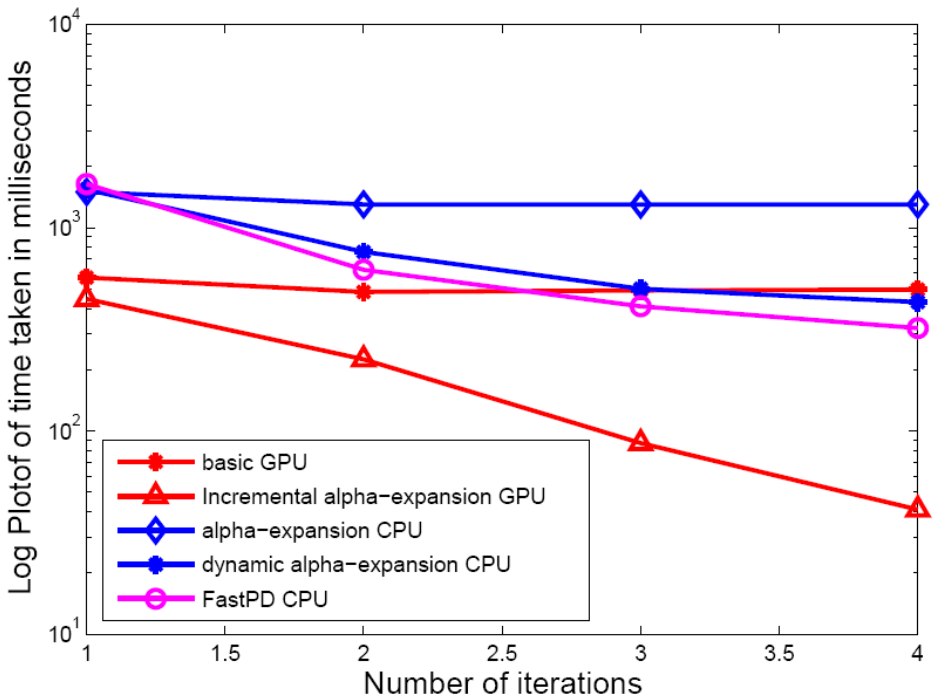
- Construct the graph based on the energy function
  - Boykov's graph construction
    - Introduces new nodes in the graph.
    - Grid structure of the graph is not maintained
  - Kolmogorov's graph construction
    - Does not introduce any new node.
    - Retains the grid structure of the graph.
    - Suitable for the GPU architecture.
- We construct flagged graphs using it
  - Nodes participate based on labels only

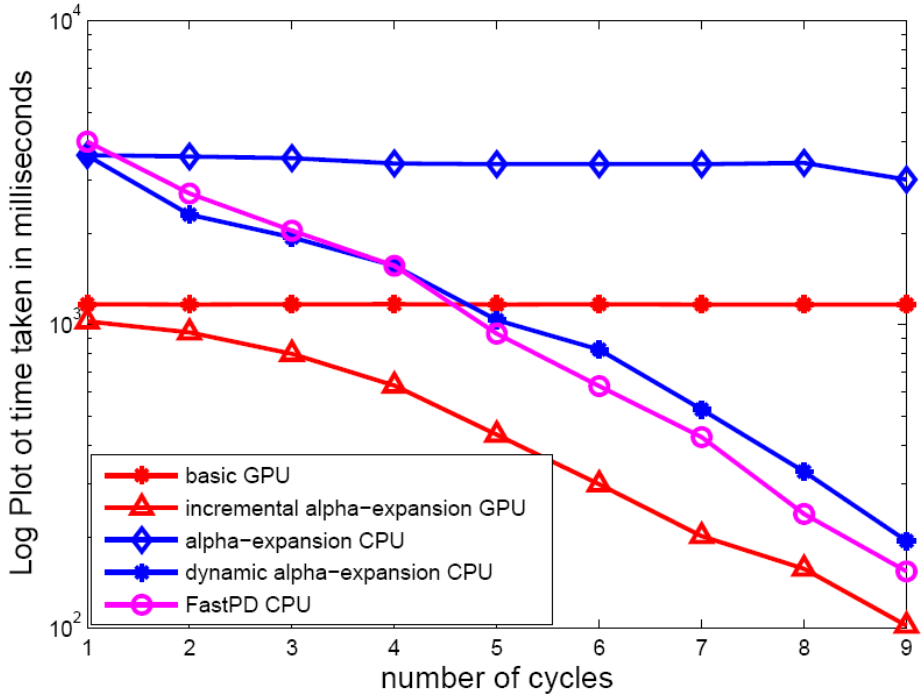
Incremental α-Expansion



- Reusability of flows, as in dynamic MRF
  - Better initializations for next graph cut
- Incremental/Dynamic
  - Reuse the flows from label to label and and recycle flows from cycle to cycle.

Incremental α-Expansion Results

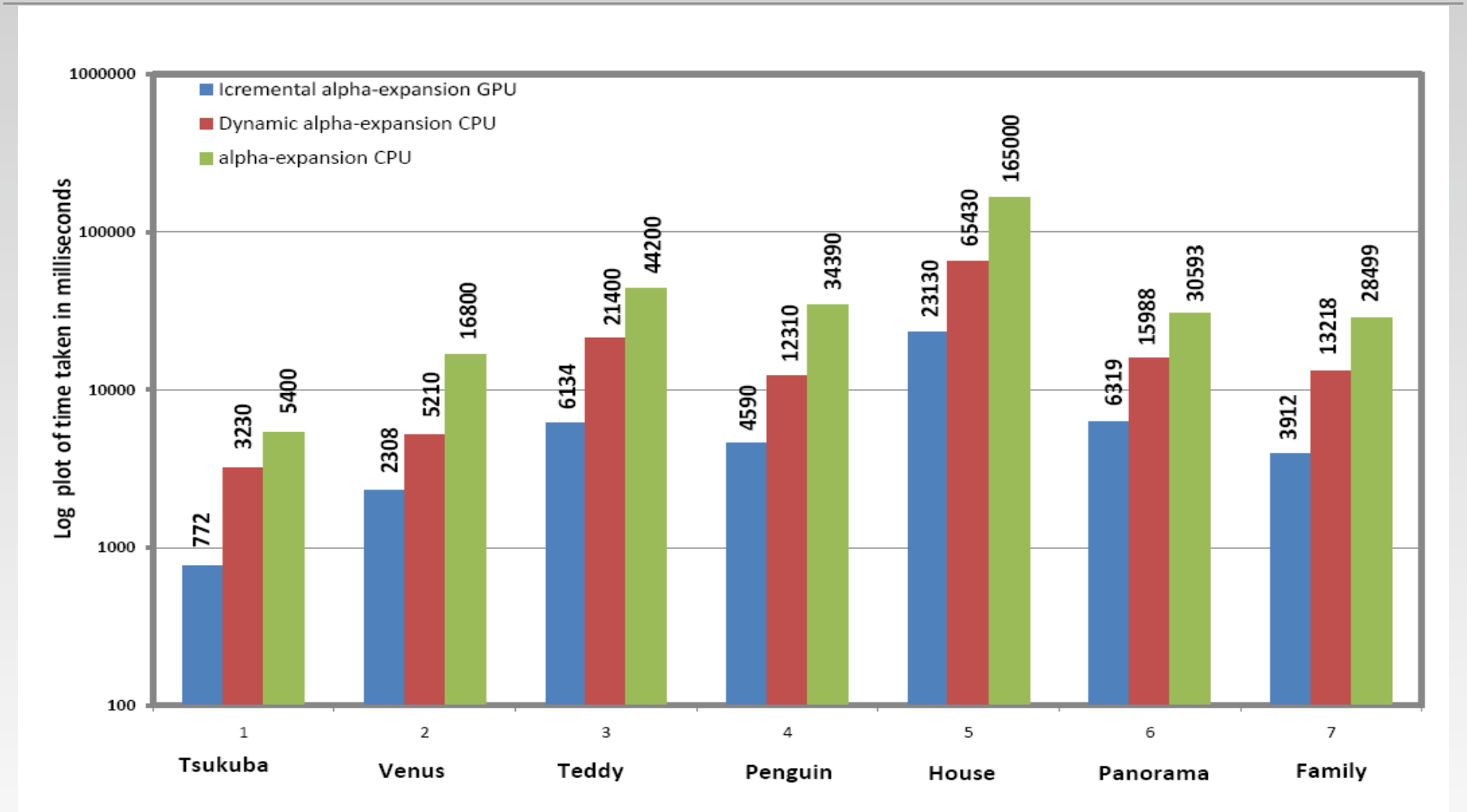




Stereo: Tsukuba Image with 16 labels

Restoration: Penguin Image with 256 labels

Incremental α-Expansion Results



Total Timings on Different Datasets

Summary

- Graph cuts can be performed at real-time rates
- Multilabel MRF optimizations built on the basic 2-level maxflow algorithm
- Code is available from http://cvit.iiit.ac.in/resources

Related Work

- On Implementing Graph Cuts on CUDA, M. Hussein, A. Varshney, and L. Davis. In GPGPU, October 2007.
- A Scalable graph-cut algorithm for N-D grids. Delong and Boykov, In CVPR-08
- Hardware-Efficient Belief Propagation. Liang, Cheng, Lai, Chen. In CVPR-09

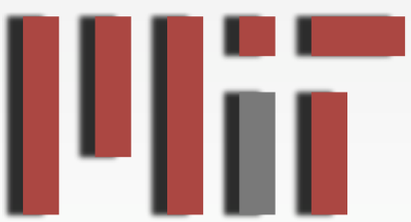
Thank you!

Thanks to Nvidia Corporation, Naval Research Board of India

Unlocking Biologically-Inspired Computer Vision: a High-Throughput Approach

Nicolas Pinto, James DiCarlo and David Cox

CVPR 2009 | June 21, 2009



The Rowland Institute at Harvard
HARVARD UNIVERSITY

Quote to remember...

Friend: **So, what are you studying for your PhD?**

Me: **I study biological and artificial vision.**

Friend: **What?!? But vision is super easy!**

The Problem:
Visual Object Recognition

- *Fast*
- *Accurate*
- *Tolerant to variation*
- *Effortless*
- *Critical to survival*

(for primates)

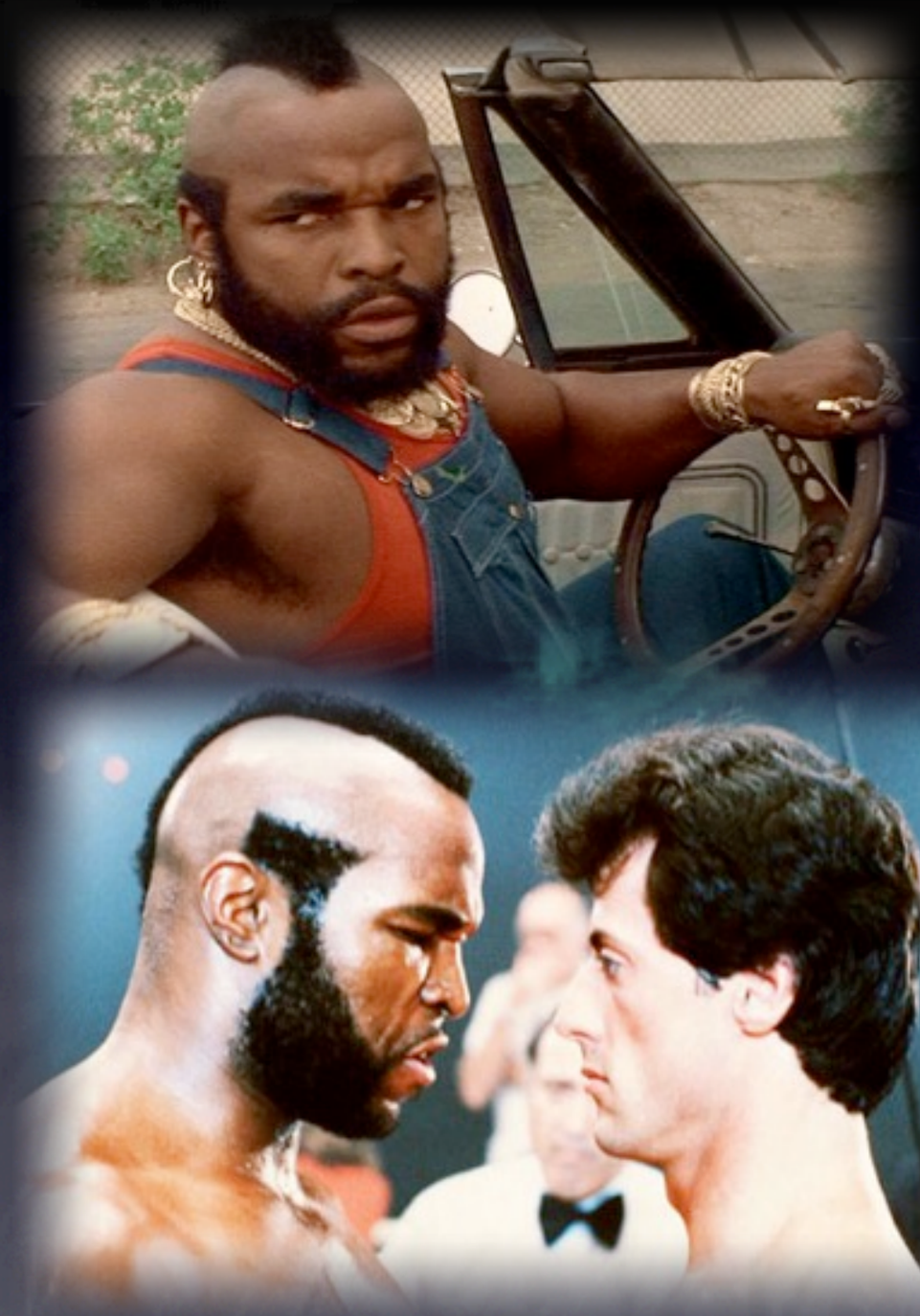
hard?

// the world is 3D but the retina is 2D

// the curse of dimensionality

// considerable image variation

image variation!



do you recognize me ?

the brain!

~50% of that is for vision!

you learned it...

Need for speed

Hardware

Software

Results

The Approach:
Reverse Engineering the Brain

REVERSE

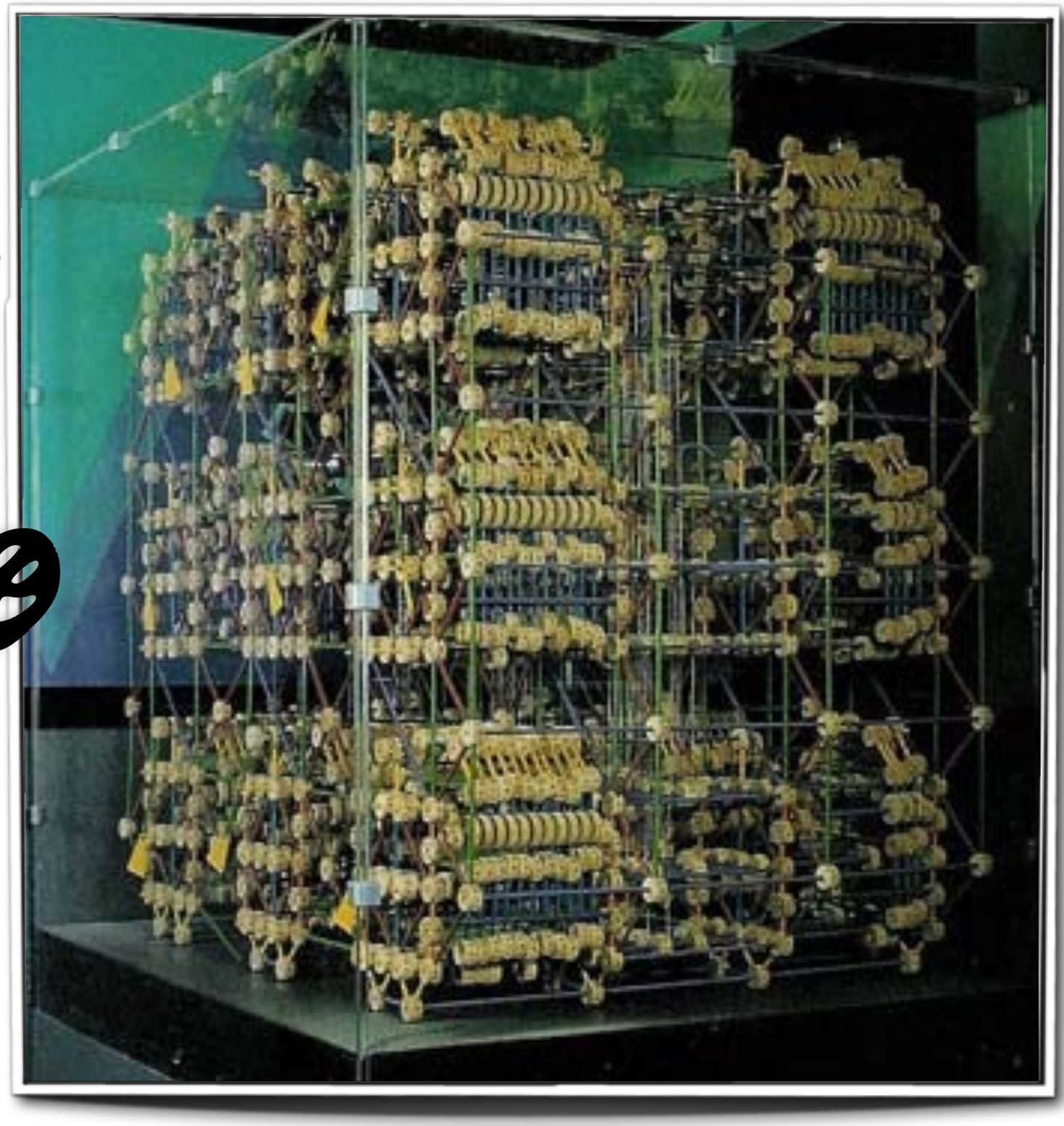
Study
Natural System

FORWARD

Build
Artificial System

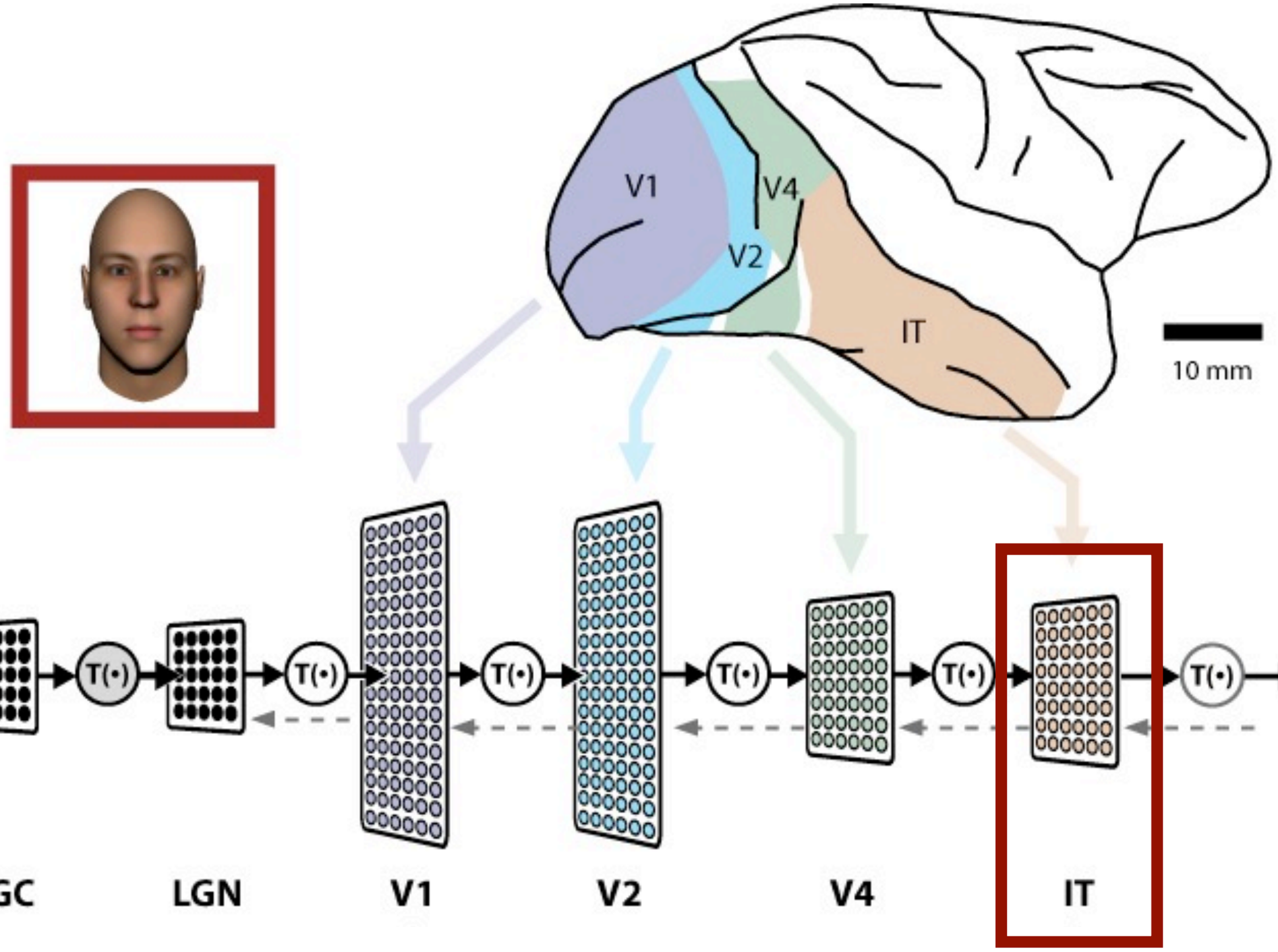
Reverse Engineering ...

What is this doing ?

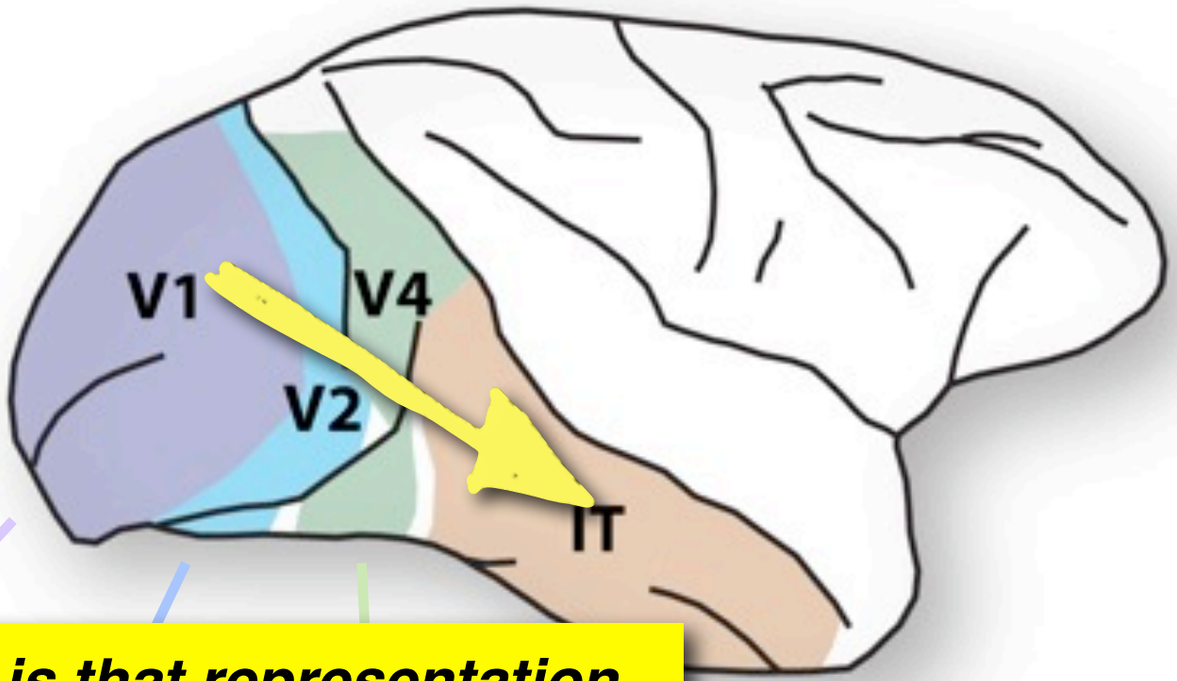


Reverse Engineering **the Brain!**

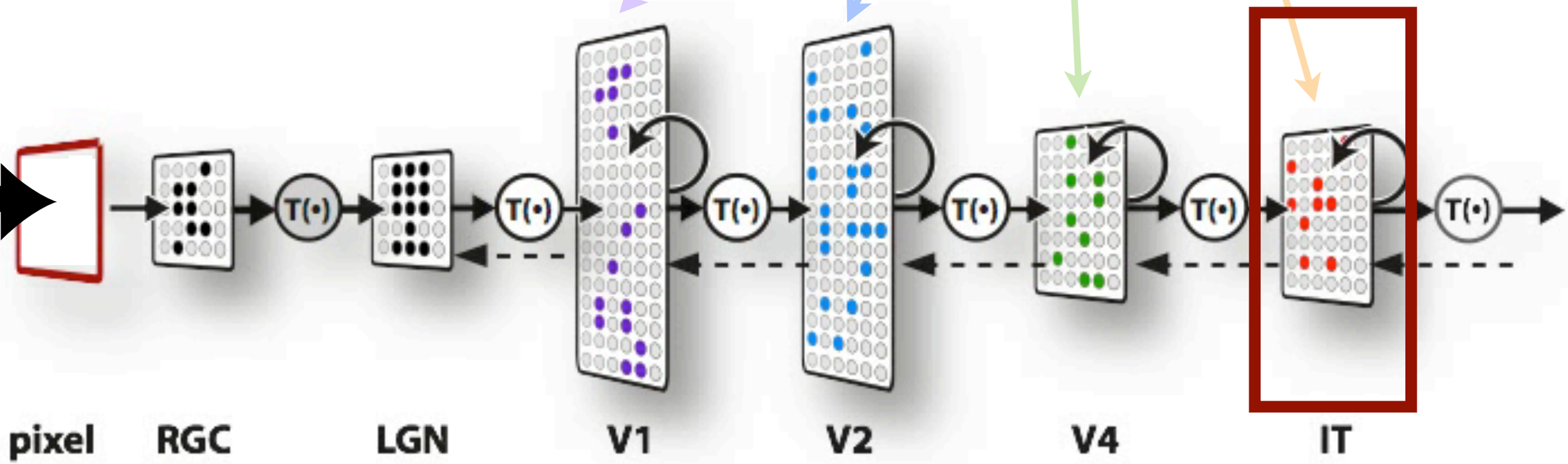
Visual Cortex



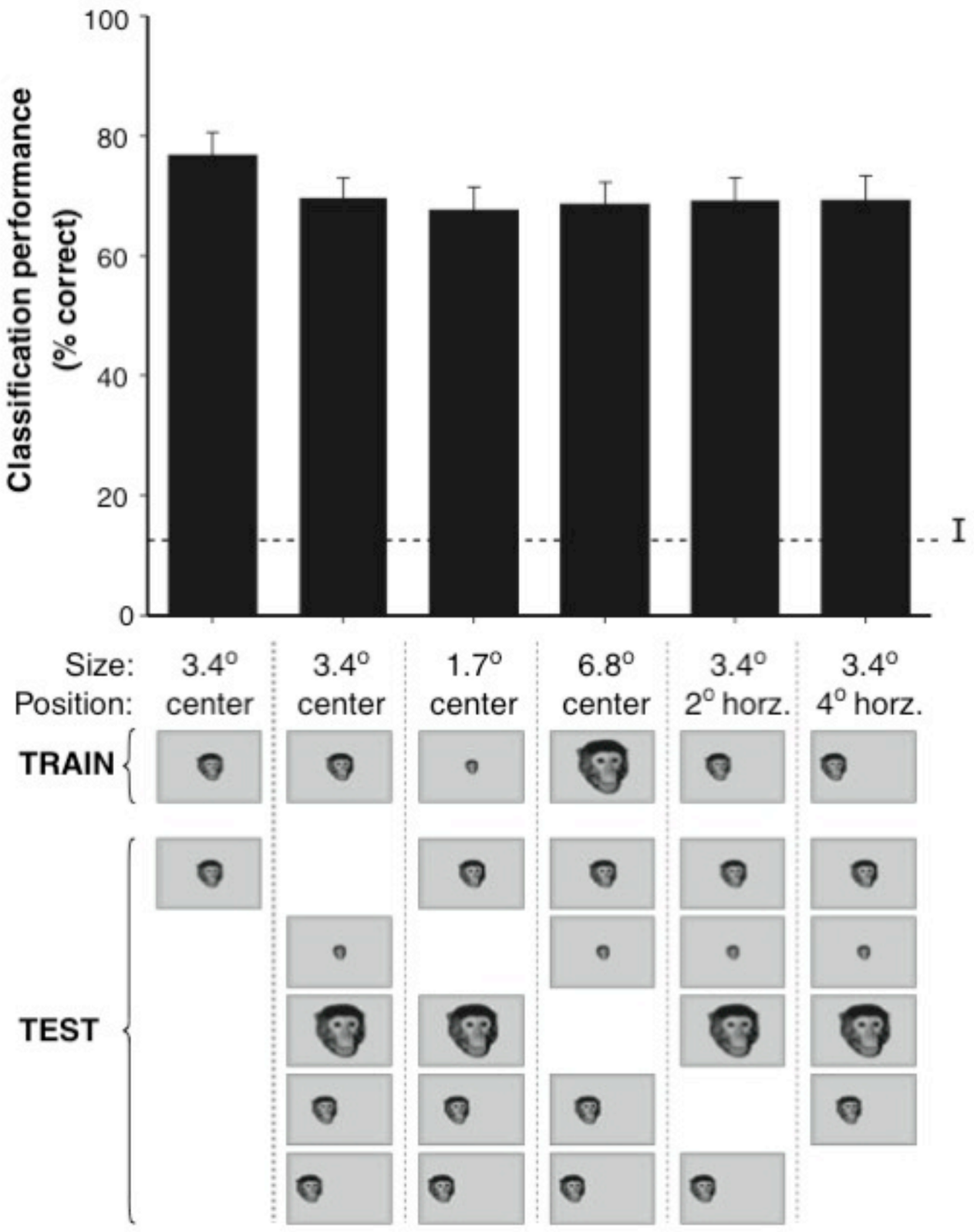
The Ventral Visual Stream

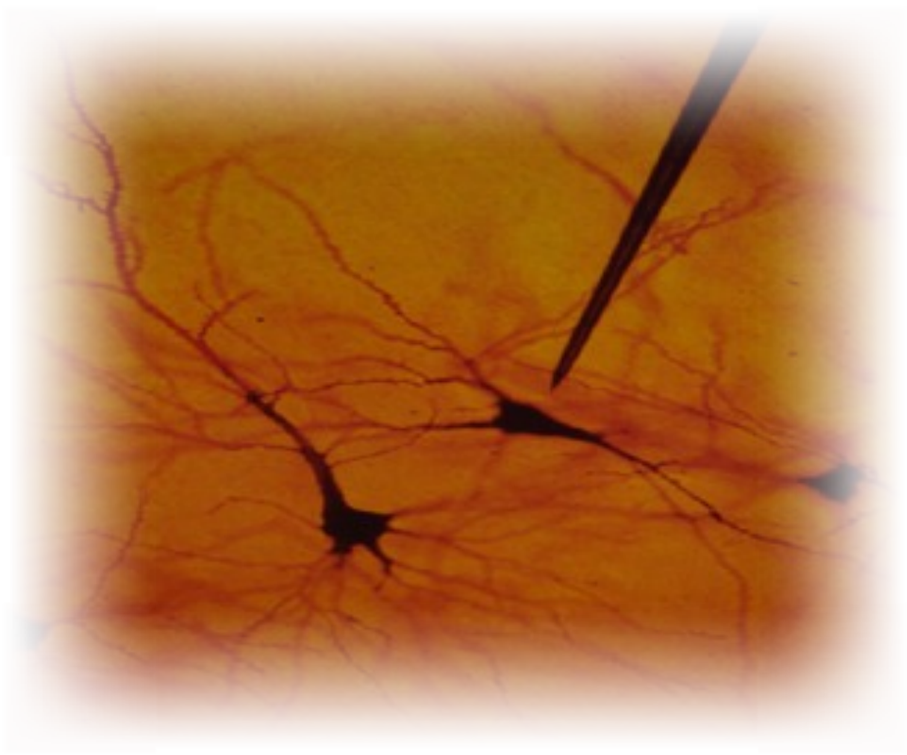


How is that representation constructed ???



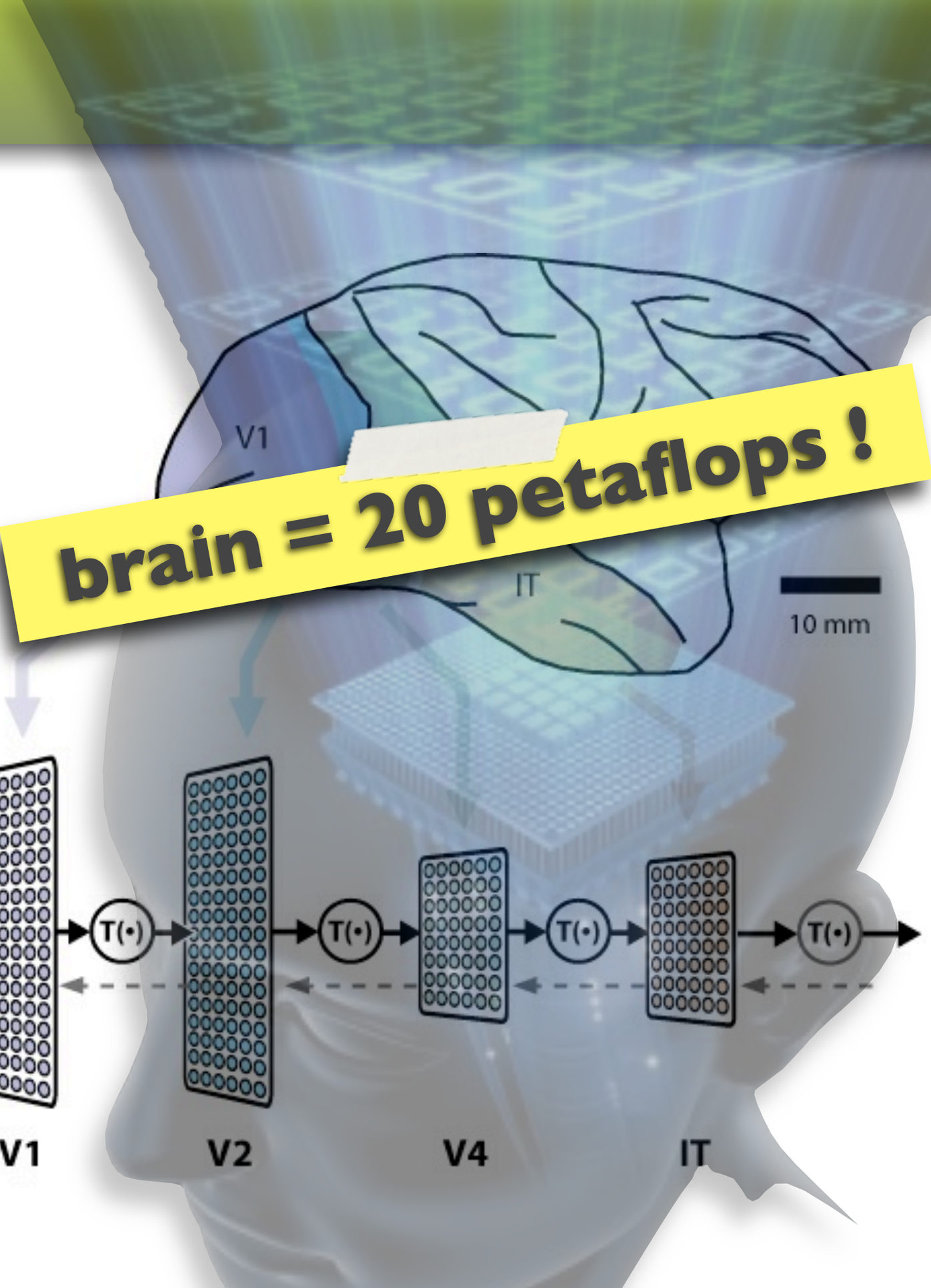
IT Cortex can do object recognition





Hung, Kreiman*, Poggio and DiCarlo,* ***Science*** *(2005)*

Visual Cortex



brain = 20 petaflops !

The need for **speed**

- **billions** of neurons and synapses

- **large-scale** natural evolution ("high-throughput screening" of neural architectures)

- **hours** of unsupervised learning experience

- faithful reproduction of other models (i.e. blend **many highly tuned** techniques)

Our strategy

Capitalizing on non-scientific high-tech markets and their $billions of R&D...

- **Gaming:** GPUs, PlayStation 3 (CellBE)
- **Web 2.0:** Cloud Computing (Amazon, Google)

Need for speed

Hardware

Software

Results

GPUs (since 2006)

7800 GTX
(2006)

OpenGL/Cg

C++/Python

Monster16GPU
(2008)

CUDA

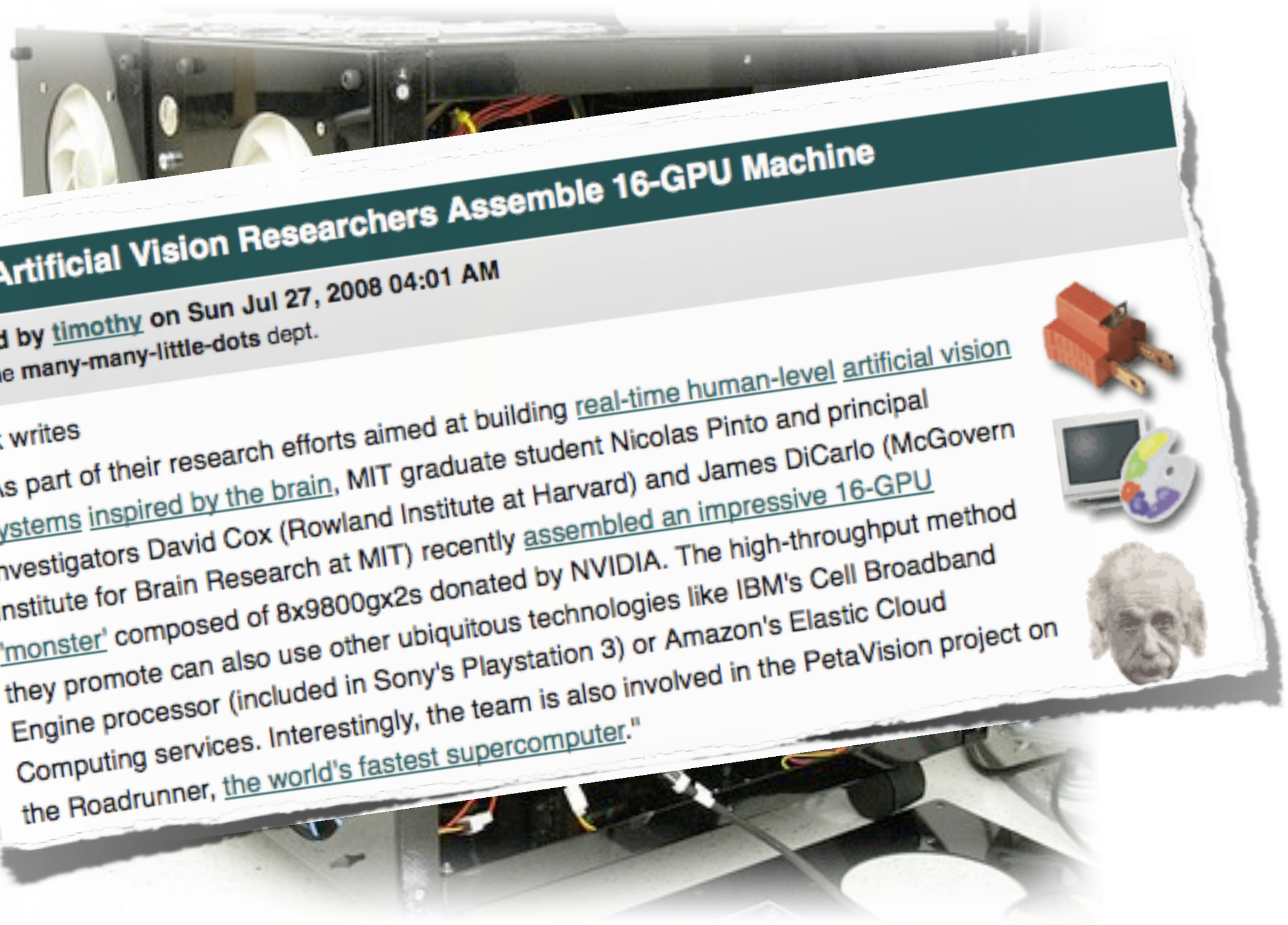
Python

Tesla Cluster
(2009)

CUDA/OpenCL

Python

GPU pr0n



MIT Artificial Vision Researchers Assemble 16-GPU Machine

Posted by timothy on Sun Jul 27, 2008 04:01 AM
from the **many-many-little-dots** dept.

lindik writes

> "As part of their research efforts aimed at building real-time human-level artificial vision systems inspired by the brain, MIT graduate student Nicolas Pinto and principal investigators David Cox (Rowland Institute at Harvard) and James DiCarlo (McGovern Institute for Brain Research at MIT) recently assembled an impressive 16-GPU 'monster' composed of 8x9800gx2s donated by NVIDIA. The high-throughput method they promote can also use other ubiquitous technologies like IBM's Cell Broadband Engine processor (included in Sony's Playstation 3) or Amazon's Elastic Cloud Computing services. Interestingly, the team is also involved in the PetaVision project on the Roadrunner, the world's fastest supercomputer."

Build your own!

Our 16-GPU Monster-Class Supercomputer

the world's most compact (18"x18"x18") and inexpensive ($3000) supercomputer

Cell Broadband Engine (since 2007)

Teraflop Playstation3 clusters:

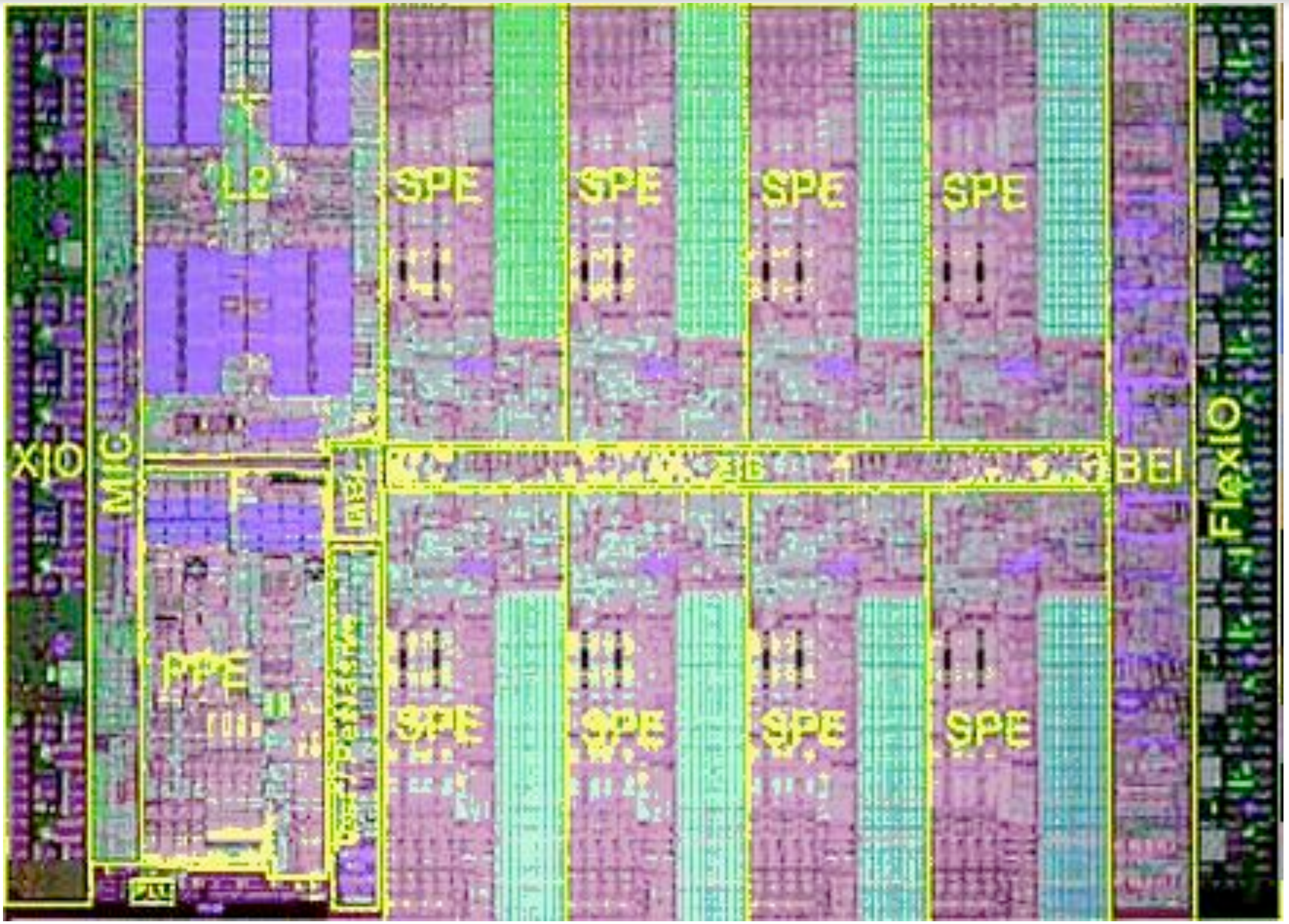
DiCarlo Lab / MIT

Cox Lab / Harvard

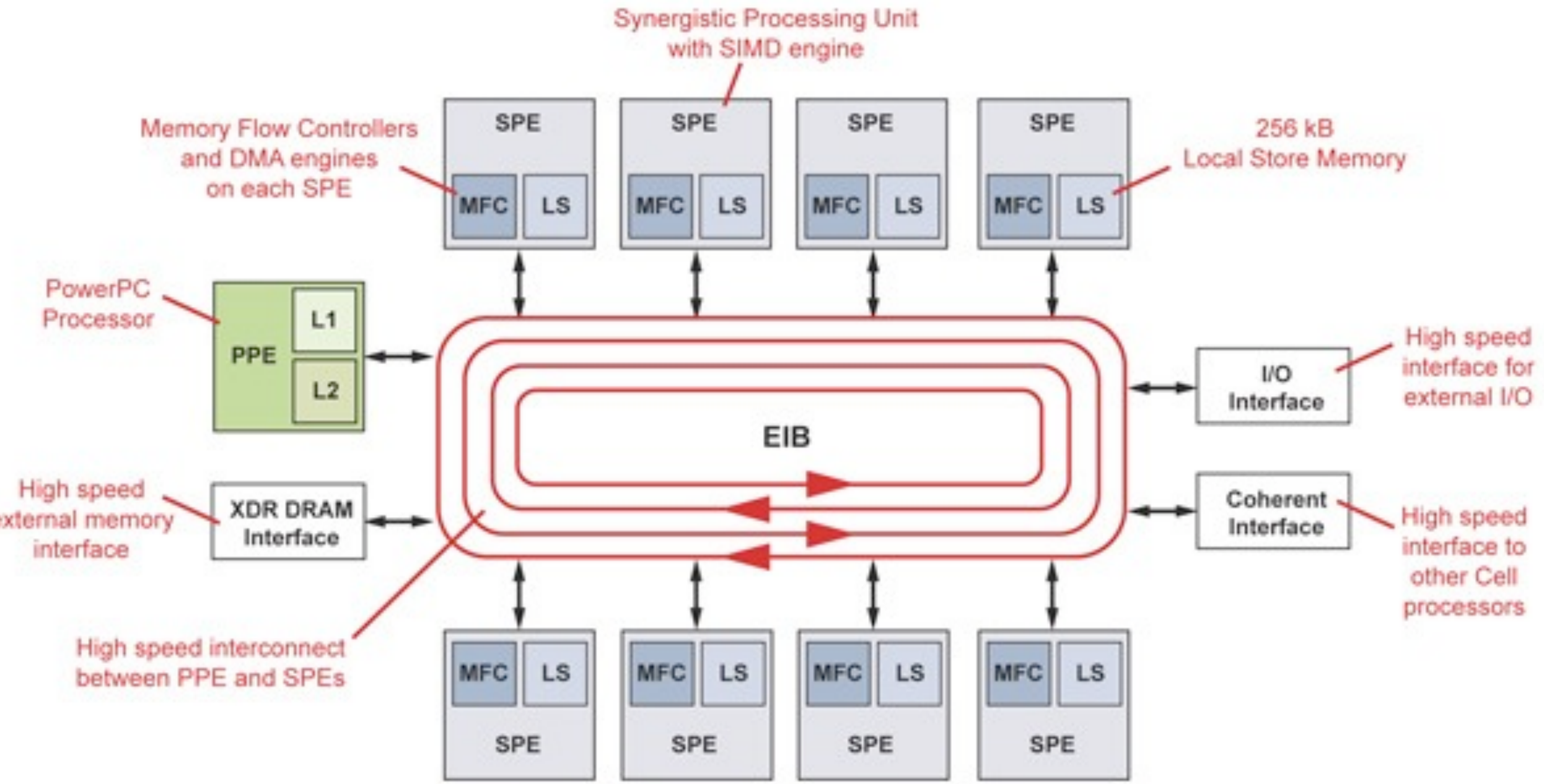
Cell Broadband Engine: history

- Designed in 2001 by Sony, Toshiba and IBM
- First appearance in Sony's Playstation 3
- IBM's Petaflop Roadrunner uses 12,240 Cell processors:

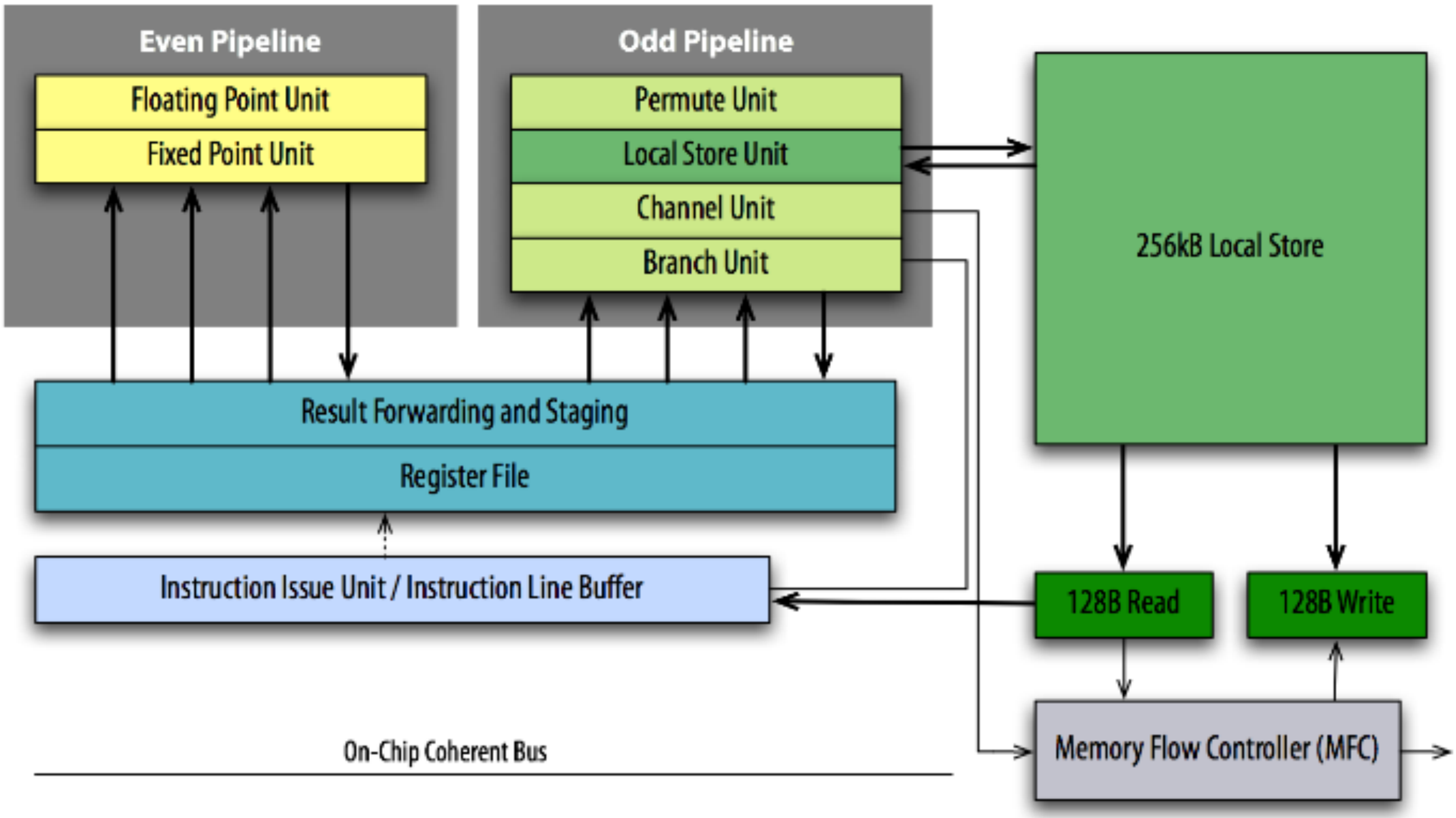
Cell Broadband Engine: architecture



Cell Broadband Engine: architecture



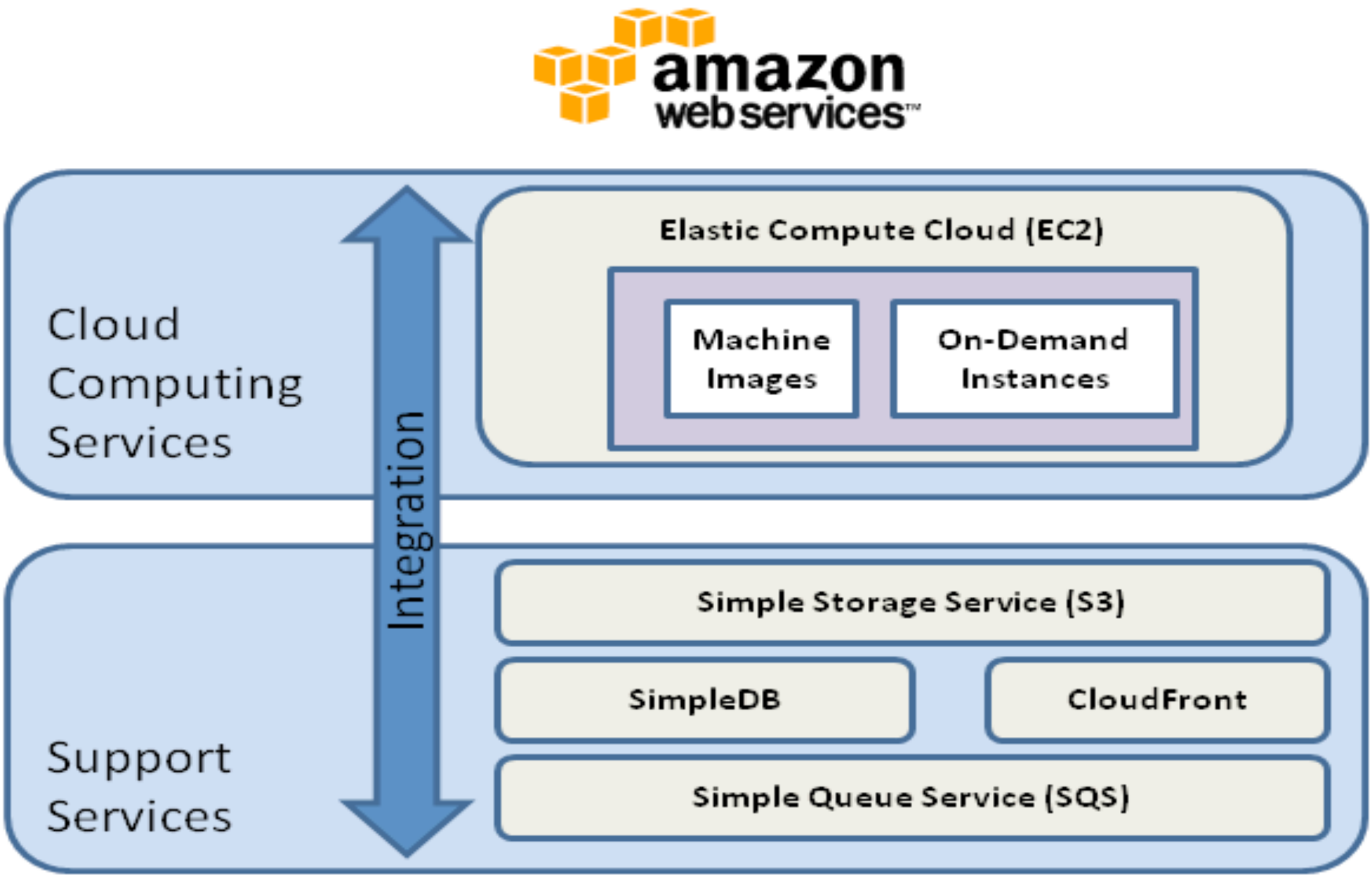
Cell Broadband Engine: pipelines



Cell Broadband Engine: pipelines

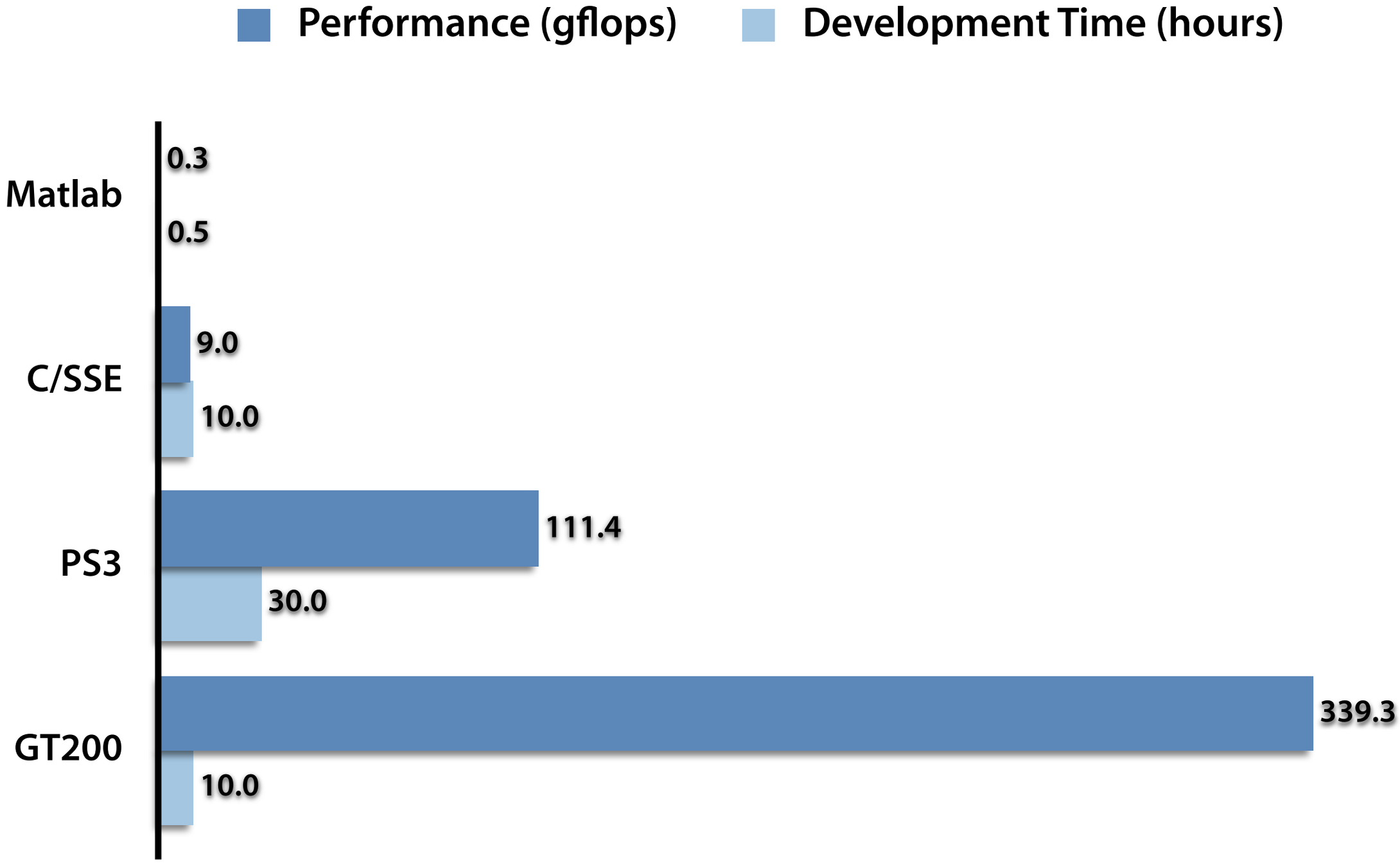
```
0D                                  78      a       $49,$8,$10
1D  012                             789     lqx     $51,$6,$9
0D                                   89     ila     $47,66051
1D  0123                             89     lqx     $52,$6,$11
0   0                                 9     ai      $7,$7,-1
0   ----456789                            fma     $50,$51,$12,$52
1       -----012345                       stqx    $50,$6,$11
1            123456                       lqx     $48,$8,$10
0D            23                          ai      $8,$8,4
1D            234567                      lqa     $44,ctx+16
1              345678                     lqx     $43,$6,$9
1                ---7890                  rotqby  $46,$48,$49
1                    ---1234              shufb   $45,$46,$46,$47
0                        ---567890        fm      $42,$12,$45
```

Amazon Cloud Computing (since 2008)



Some numbers...

3D Filterbank Convolution



Some numbers...

3D Filterbank Convolution

Performance (gflops)

Need for speed

Hardware

Software

Results

What do we all want?

- Ease of use
- **Maximum raw speed**
- Ease of extension
- Hardware “agnostic”

A little story

You just finished your code...

1. You run it on one image: it works!

2. You adjust your parameters: it’s slow!

3. Your optimize your code: it’s fast now!

4. You run it on another image: it’s slow now!

5. You repeat or you stop...

Here are the keys to Easy-High-Performance !

How ?!

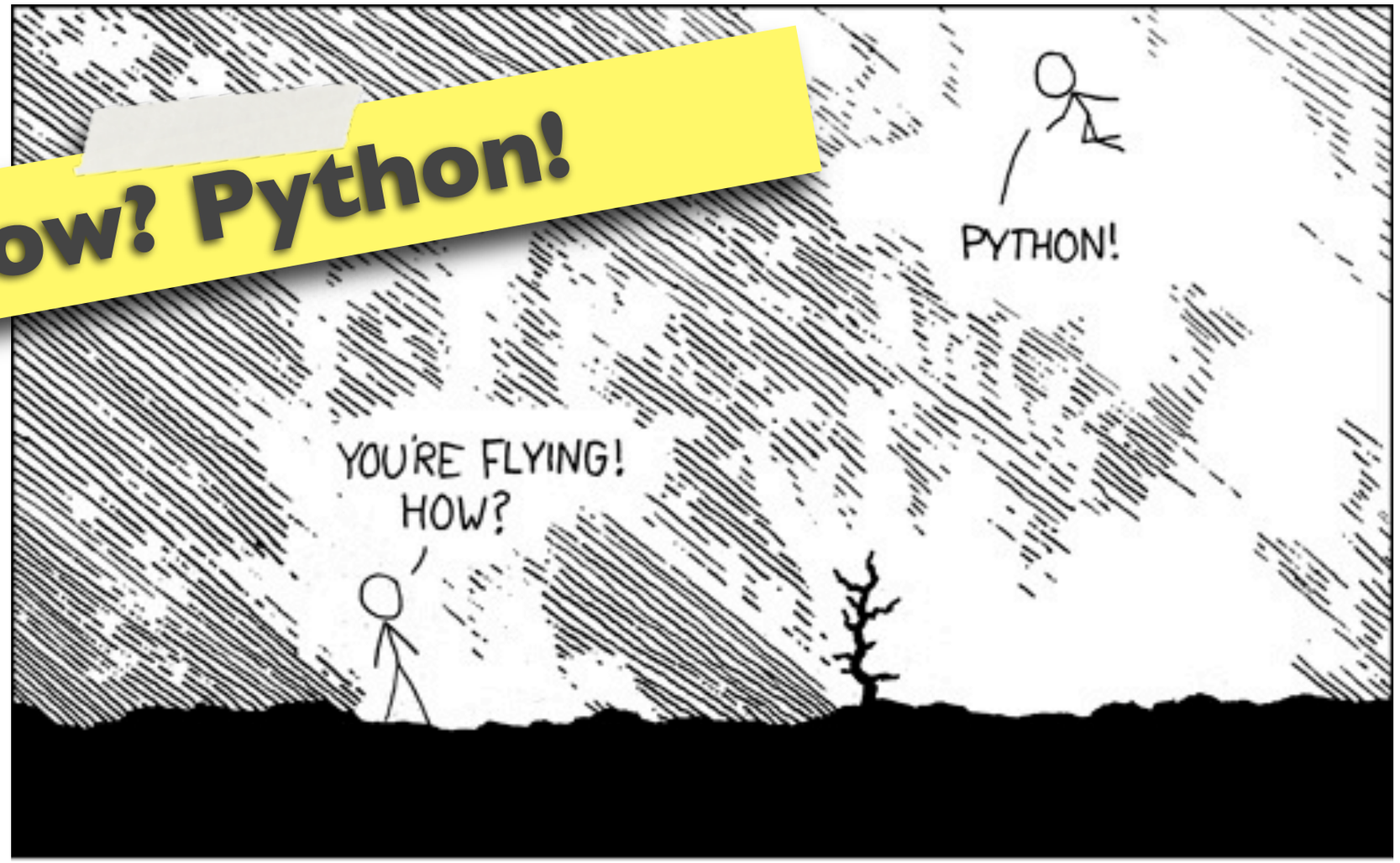
Our mantra: always use the right tool !

Our mantra: always use the right tool !

Observation:
Sadly, you can't go very far with MATLAB (R)

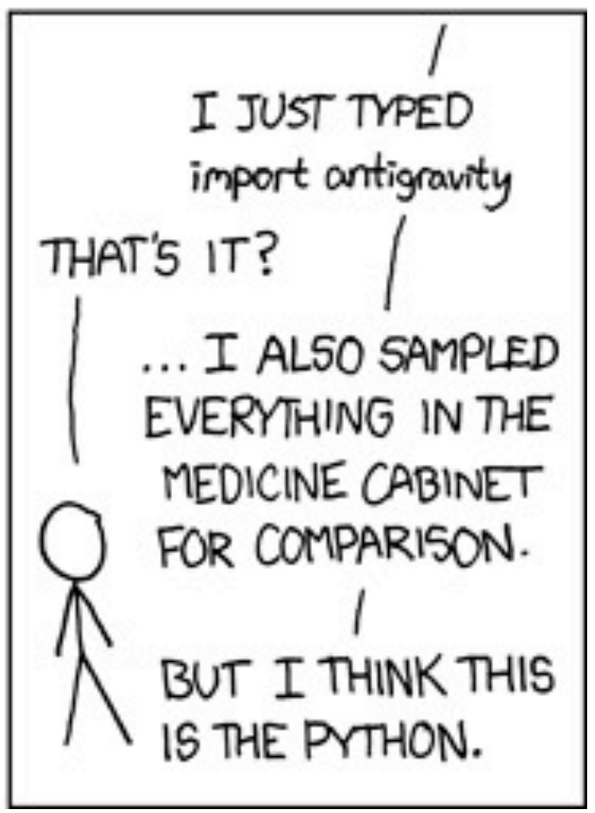
- Proprietary: difficult in the cloud...
- Only one programming paradigm
- Hard to engineer good code (poor OOP)
- Hard to profile / optimize
- MEX is a joke (more like a bad taste joke)

How? Python!


PYTHON!
YOU'RE FLYING! HOW?


I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"


I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?


I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

Python is our friend

- **GPU**: python-cg, PyCuda, python-cuda, PyOpenCL
- **PS3/Cell**: C/ctypes, C++/Boost:Python
- **Multicore**: ipython, PyMPI, multiprocessing
- **Amazon EC2**: StarCluster/SunGridEngine

- Compatible with any language:
  - Fortan, C, C++, R, etc.
  - MATLAB: mlabwrap
  - Mathematica-like: sage

Meta-programming?

Meta-programming !

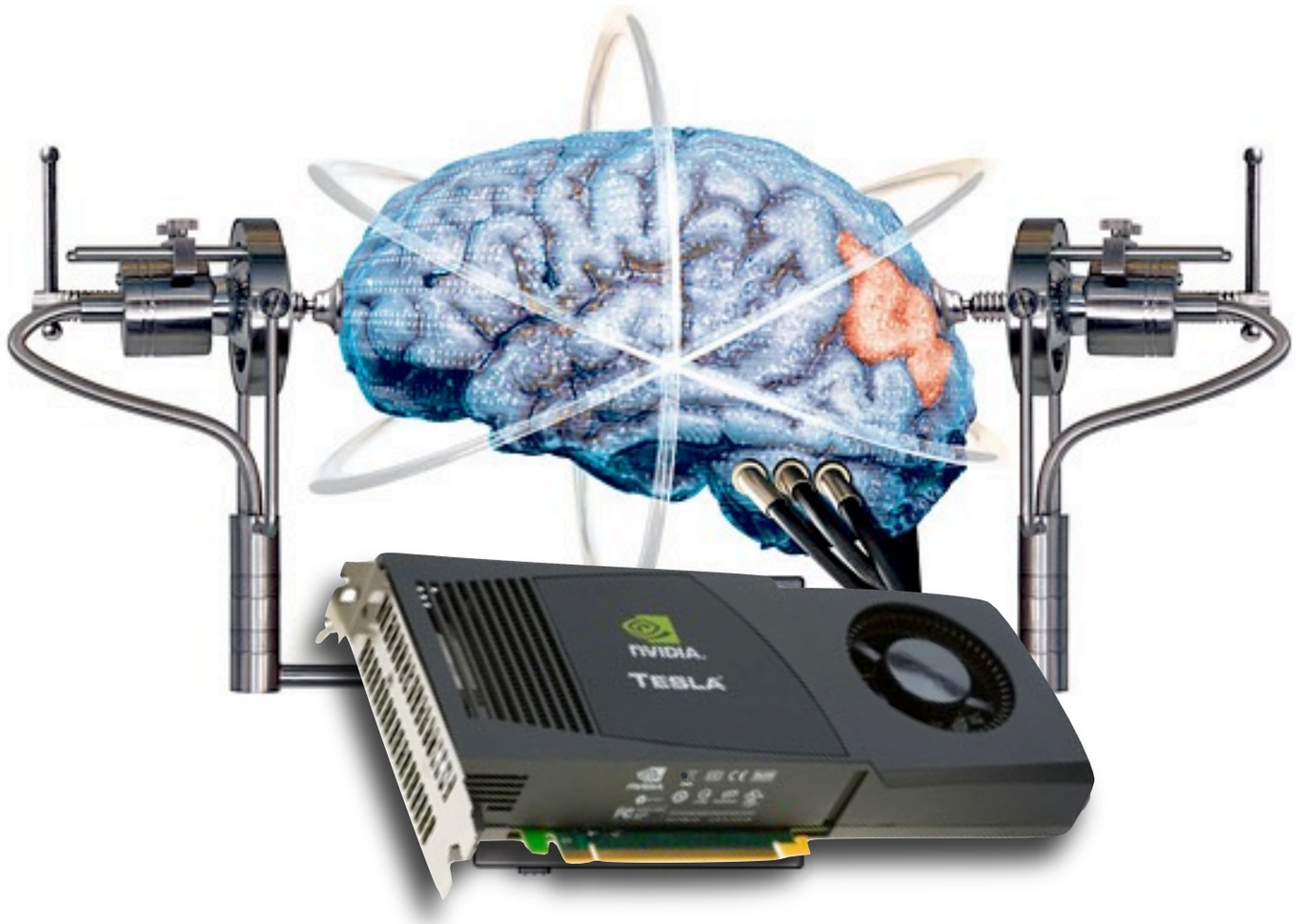
Leave the **grunt-programming** to the computer (i.e. auto-tuning like ATLAS or FFTW)

- Dynamically compile **specialized versions** of the same kernel for different conditions (~Just-in-Time Compilation (JIT))
- **Smooth** syntactic ugliness: unroll loops, index un-indexable registers
- **Dynamic**, empirical run-time **tuning**

Meta-programming!

"Instrumentalize" your solutions:

- Block size
- Work size
- Loop unrolling
- Pre-fetching
- Spilling
- **etc.**



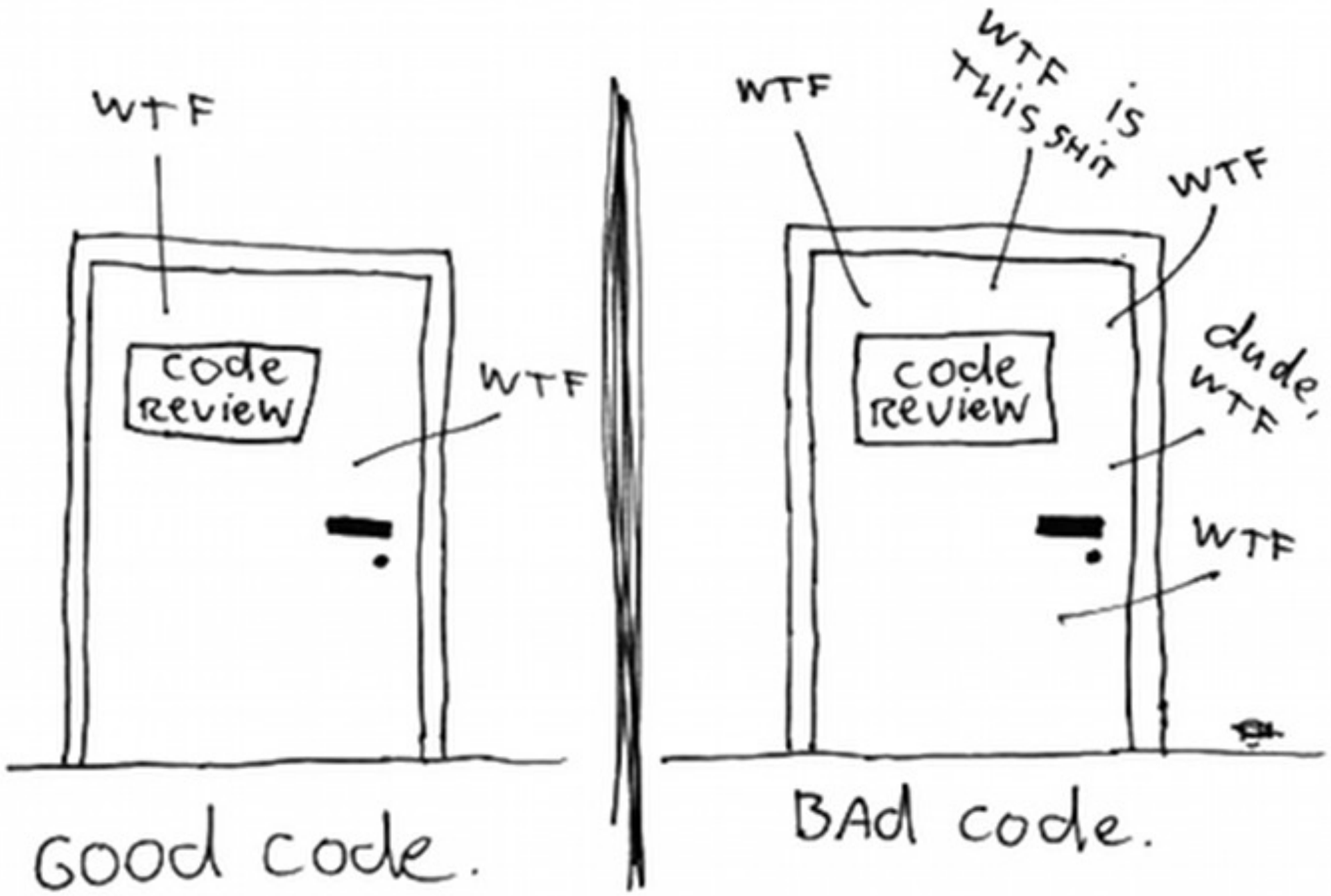
Meta-programming!

Let the computer find the **optimal code**:

- brute-force search with a **global objective**
- machine-learning approach with **local objectives** and **hidden variables** (advanced)
  - eg. PyCuda makes this easy:
    - Access properties of compiled code: `func.{registers,lmem,smem}`
    - Exact GPU timing via events
    - Can calculate hardware-dependent MP occupancy

Meta-programming requires **careful engineering**

Meta-programming requires **careful engineering**



Need for speed

Hardware

Software

Results

The Approach:
Forward Engineering the Brain

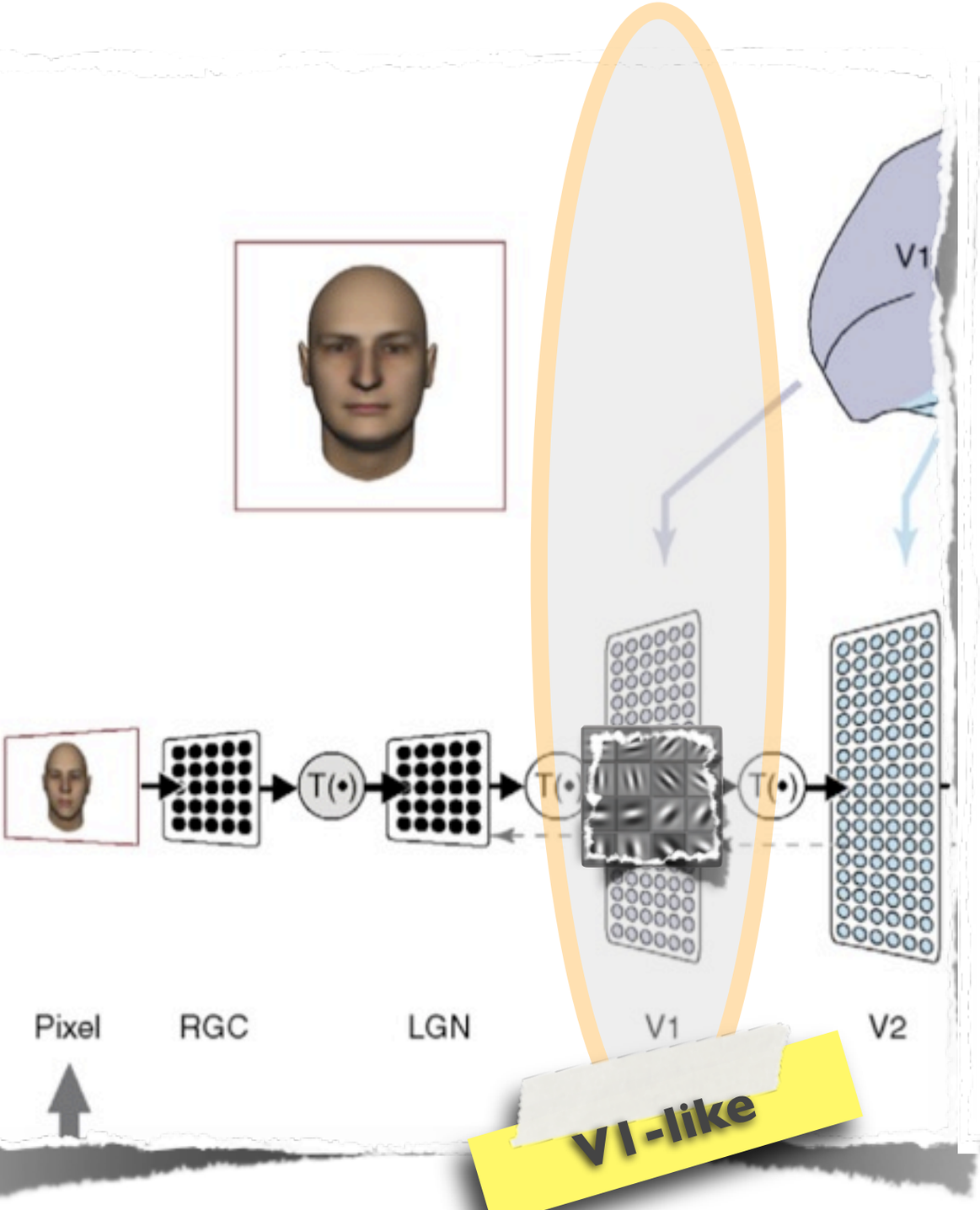
REVERSE

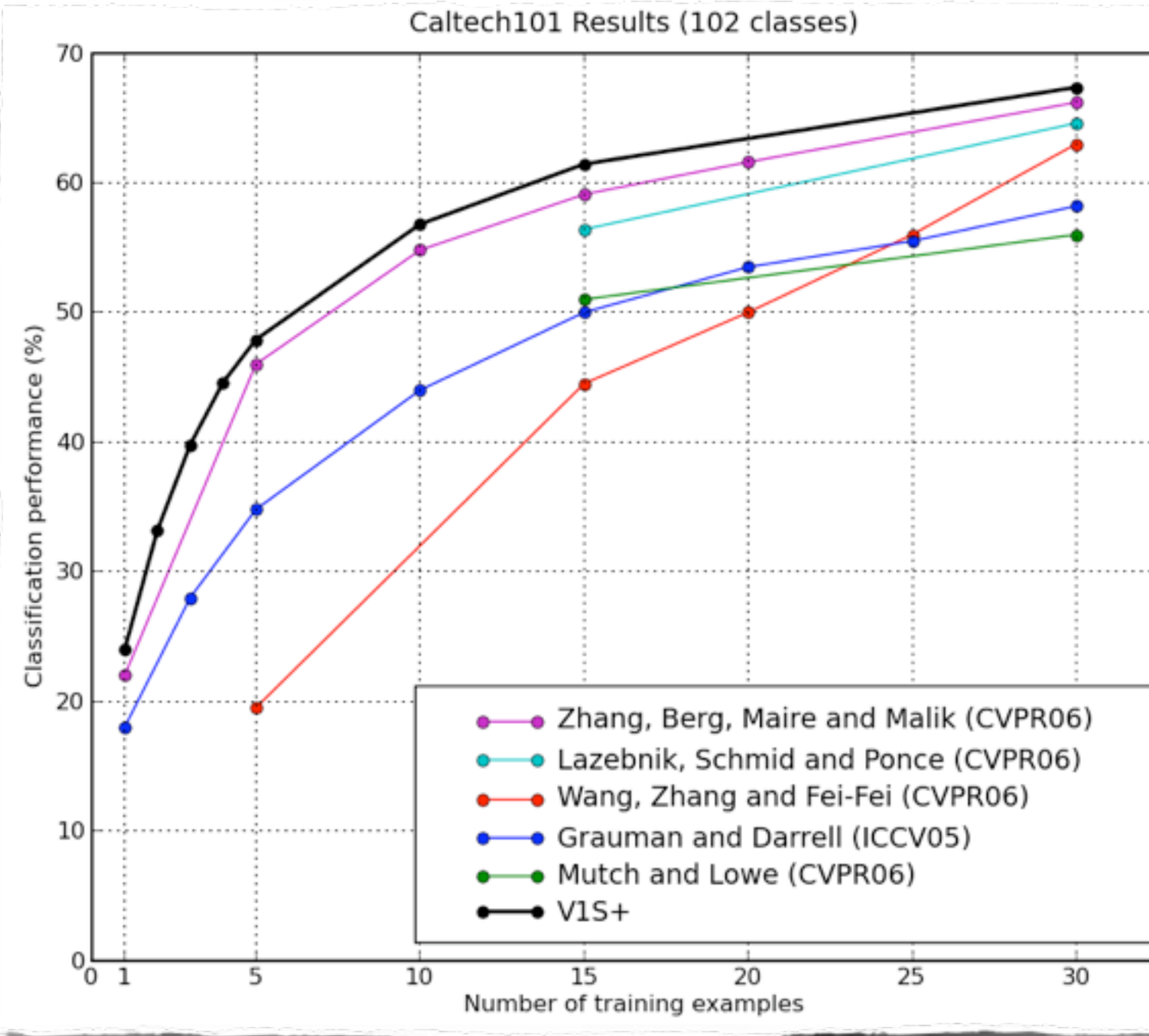
FORWARD

Study
Natural System

Build
Artificial System

State-of-the-art performance

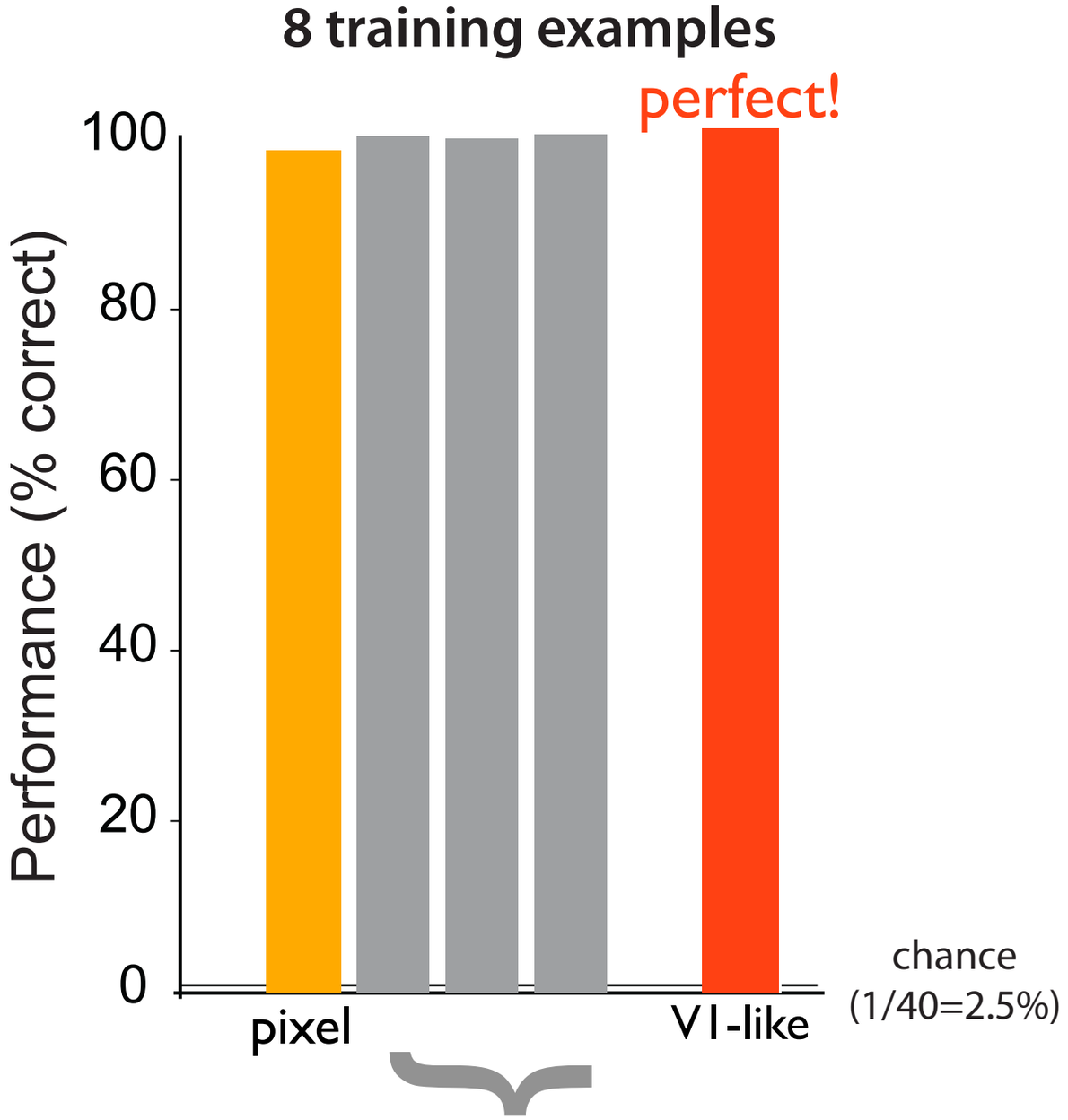




Pinto, Cox, DiCarlo PLoS08

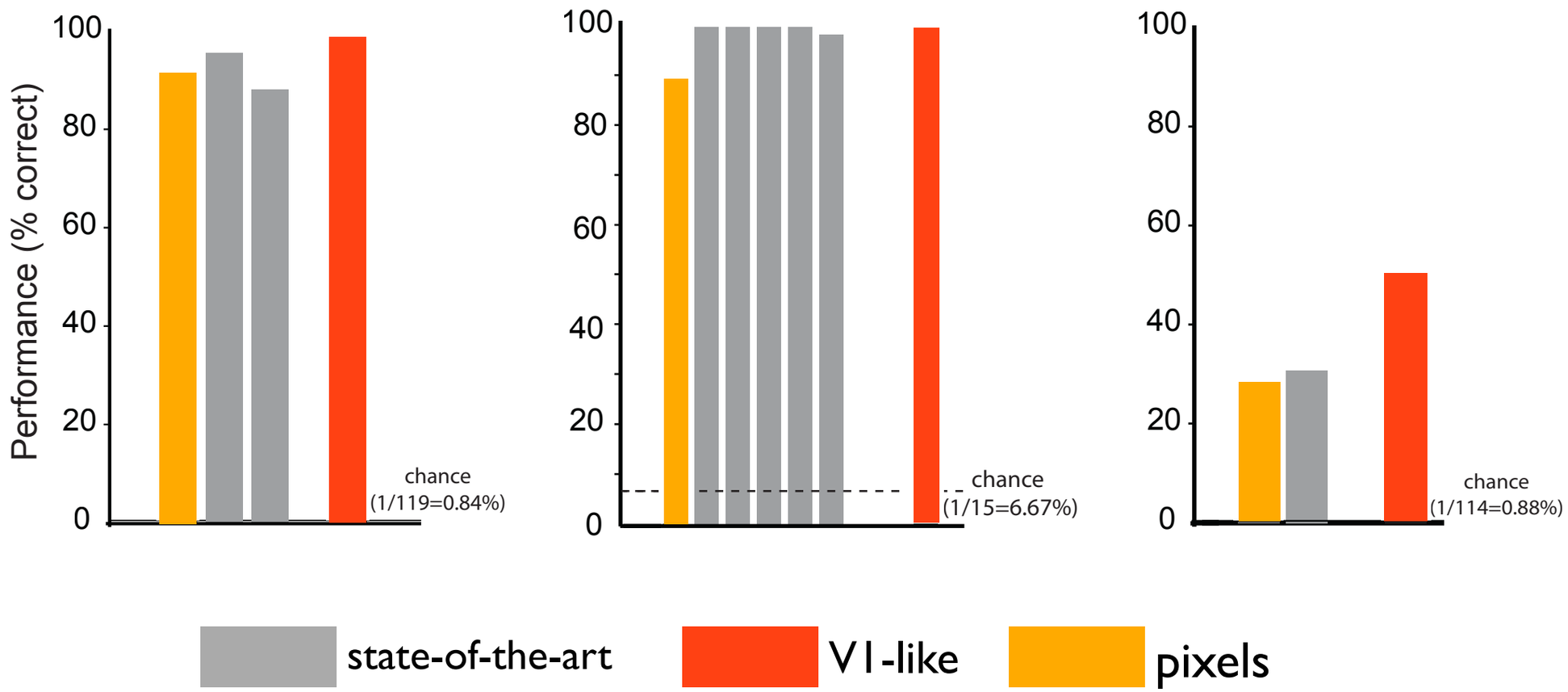
State-of-the-art performance

ORL Face Set



Pinto, DiCarlo, Cox ECCV08

State-of-the-art performance



Pinto, DiCarlo, Cox ECCV08

State-of-the-art performance

LFW Face Set

Reference	Methods	Performance
Huang08 [6]	Nowak [8]	73.93%±0.49
	MERL	70.52%±0.60
	Nowak+MERL	76.18%±0.58
Wolf08 [17]	descriptor-based	70.62%±0.57
	one-shot-learning*	76.53%±0.54
	hybrid*	78.47%±0.51
This paper	Pixels/MKL	68.22%±0.41
	V1-like/MKL	**79.35%±0.55**

Table 3. Average performance comparison with the current state-of-the-art on LFW. *note that the "one-shot-learning" and "hybrid" methods from [17] can't directly be compared to ours as they exploit the fact that individuals in the training and testing sets are mutually exclusive (i.e. using this property, you can build a powerful one-shot-learning classifier knowing that each test example is *different* from all the training examples, see [17] for more details. Our decision not to use such techniques effectively handicaps our results relative to reports that use them).

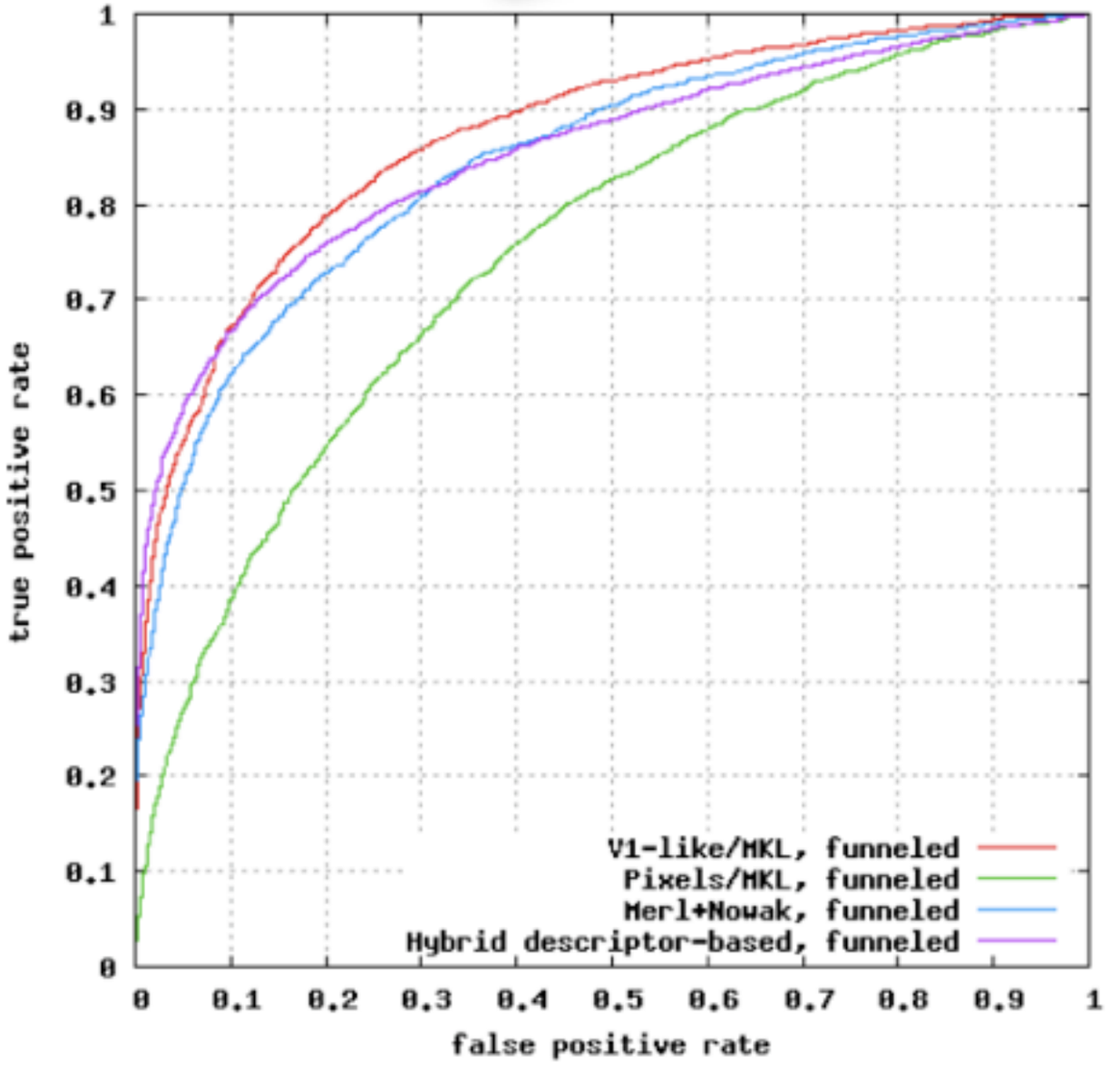
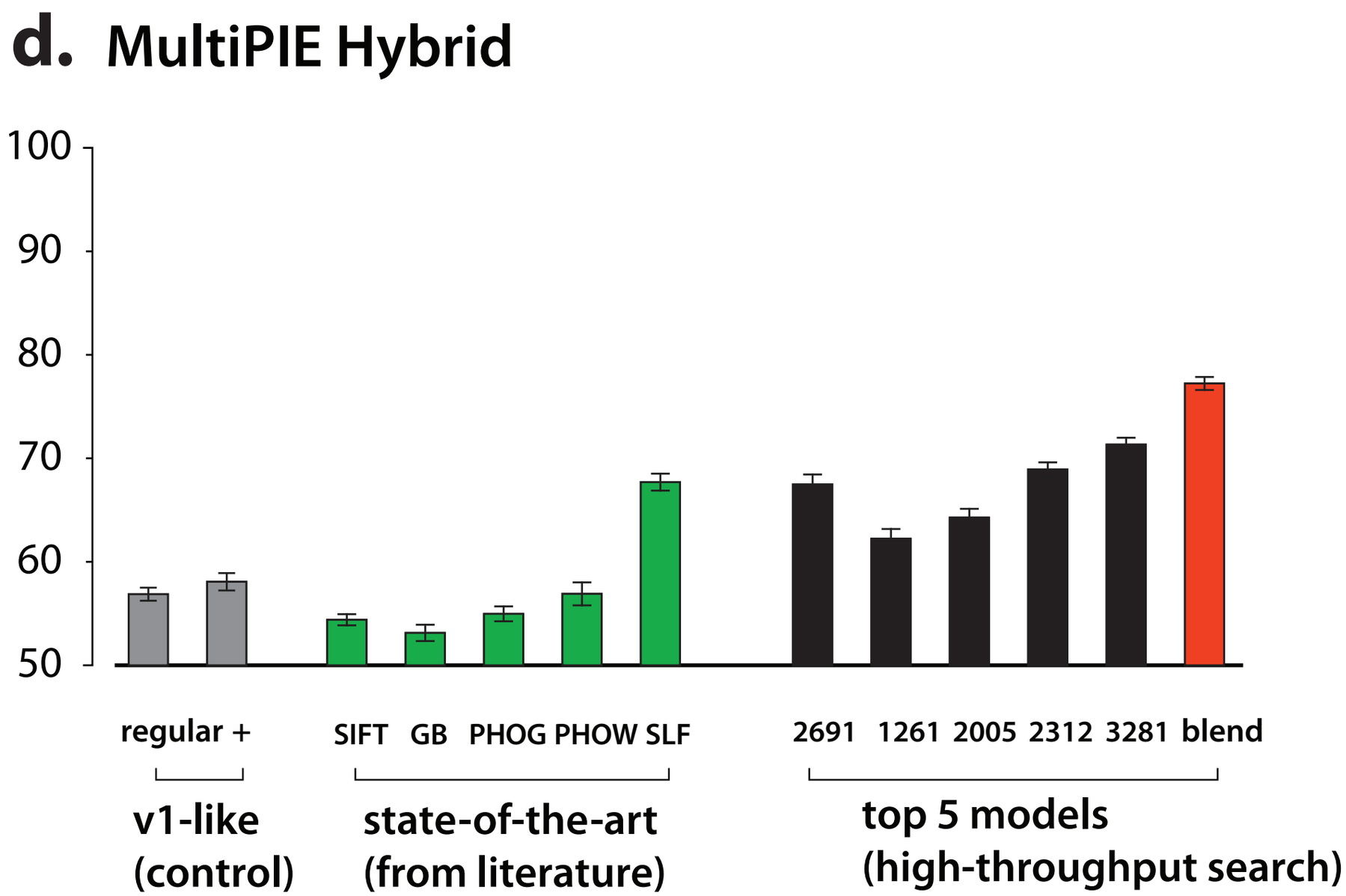


Figure 1. ROC curve comparison with the current state-of-the-art on LFW. These curves were generated using the standard procedure described in [24].

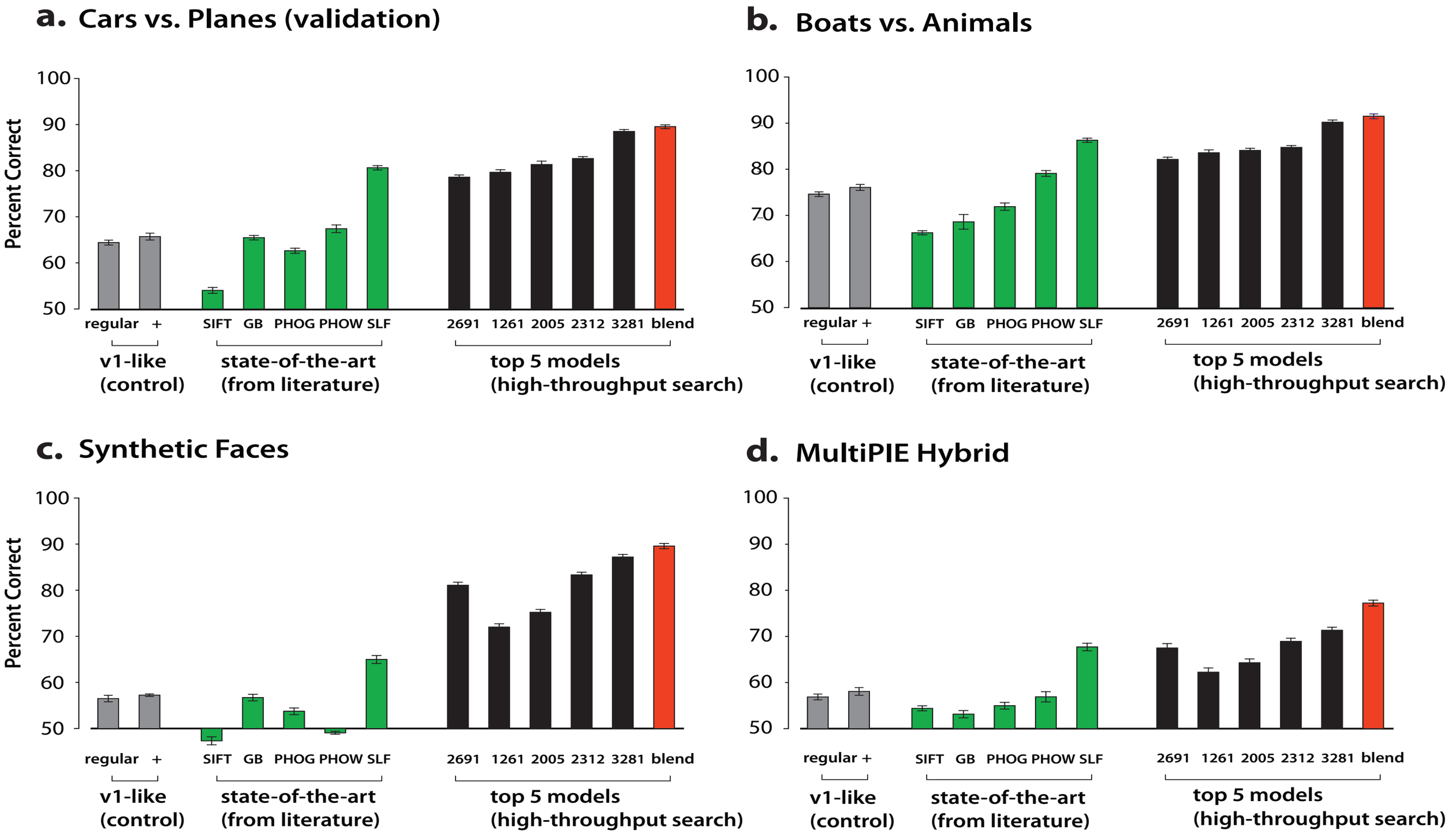
Pinto, DiCarlo, Cox CVPR09

State-of-the-art performance



Pinto, DiCarlo, Cox (in review)

State-of-the-art performance



Pinto, DiCarlo, Cox (in review)

Acknowledgements

Jim DiCarlo

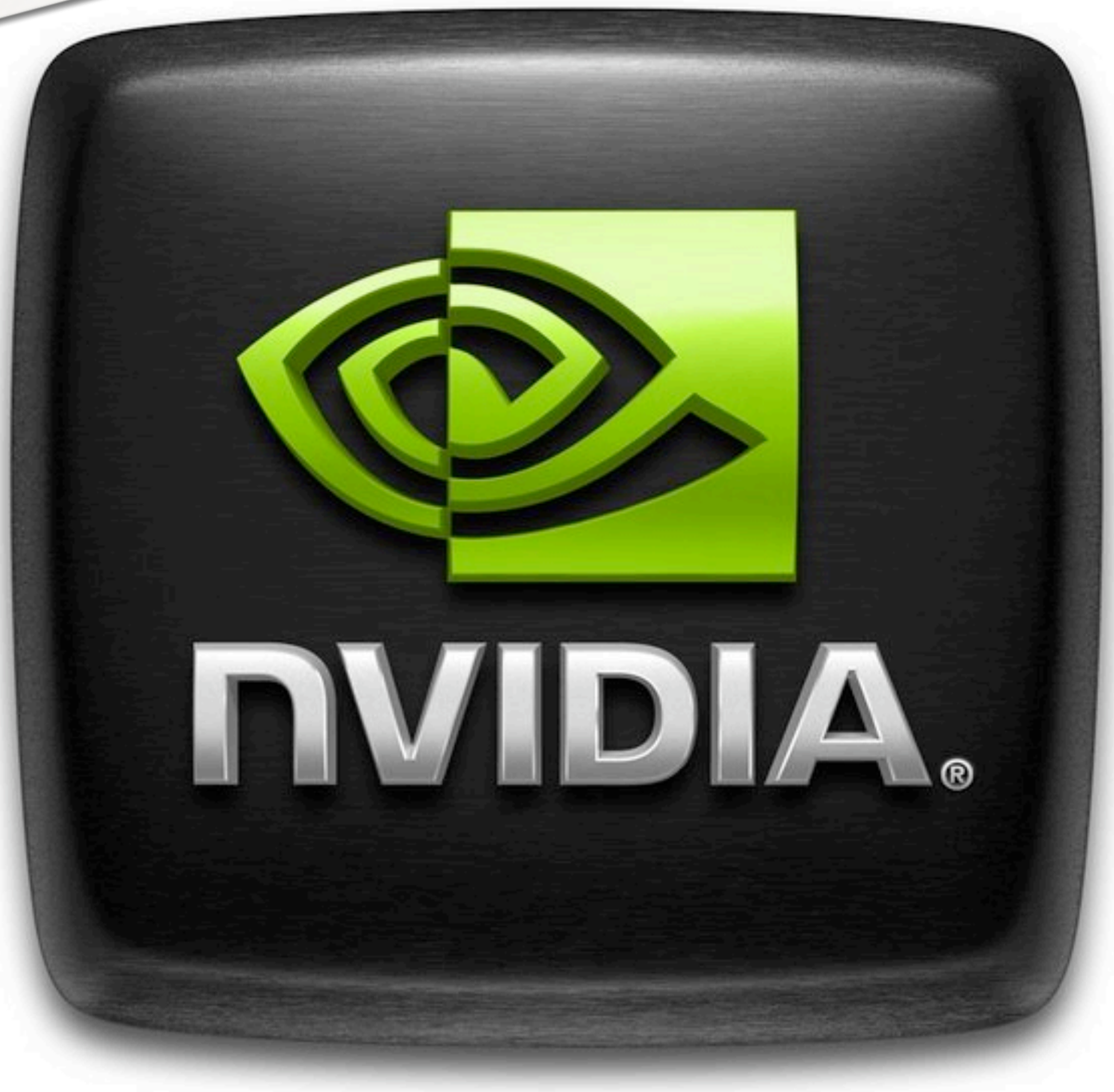
DiCarlo Lab @ MIT

David Cox

The Visual Neuroscience Group

@ The Rowland Institute at Harvard

Acknowledgements



"Thank You"

COME AGAIN

and bring a Friend!