An Introduction to GPGPU

with a case study on CUDA

Christian Lessig (lessig@dgp.toronto.edu)

UNIVERSITY of TORONTO

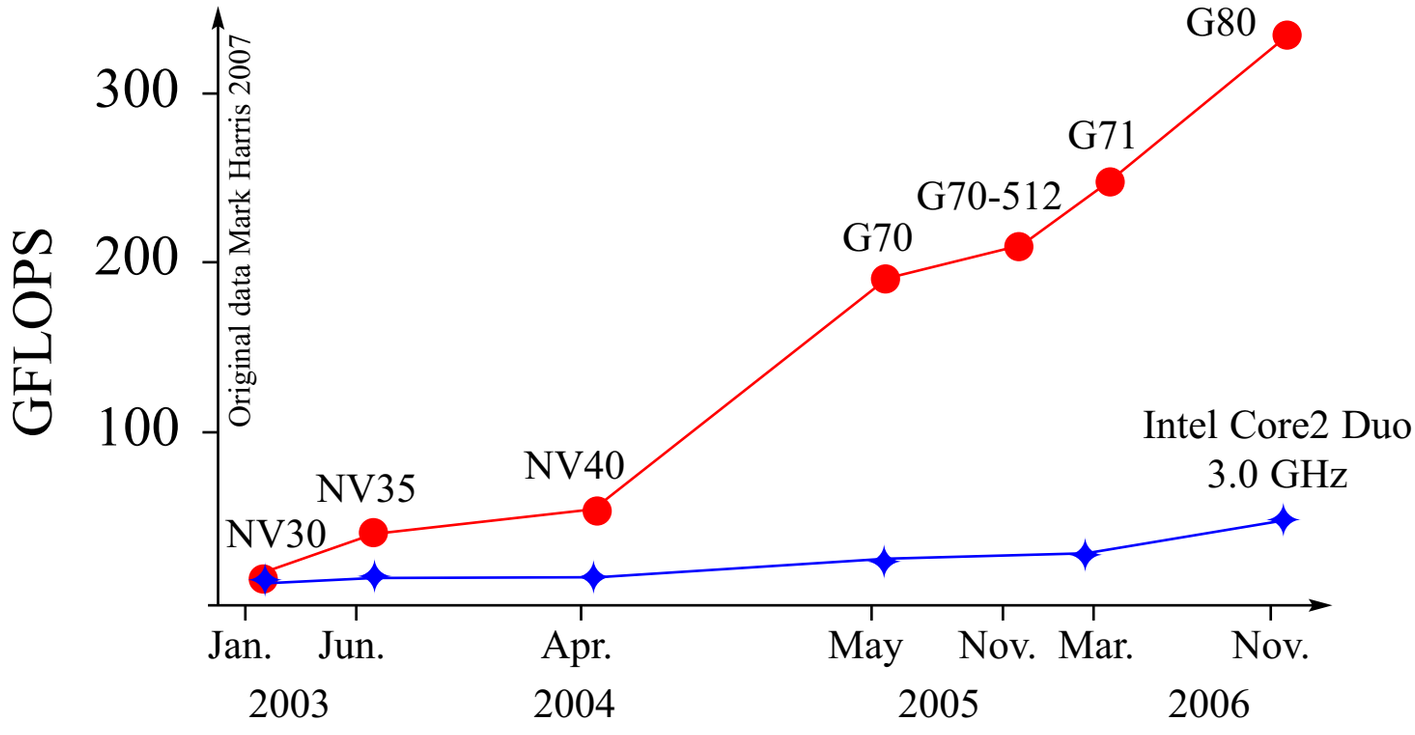
dgp
dynamic graphics project

GPGPU

- General Purpose Computations on GPUs
- GPUs for applications beyond rasterization
  - Global illumination
  - Computer vision
  - Signal processing
  - Simulation
  - Computational biology / finance
  - ...

Motivation

- Exponentially growing compute power



Motivation

- Exponentially growing compute power
- Very high memory bandwidth

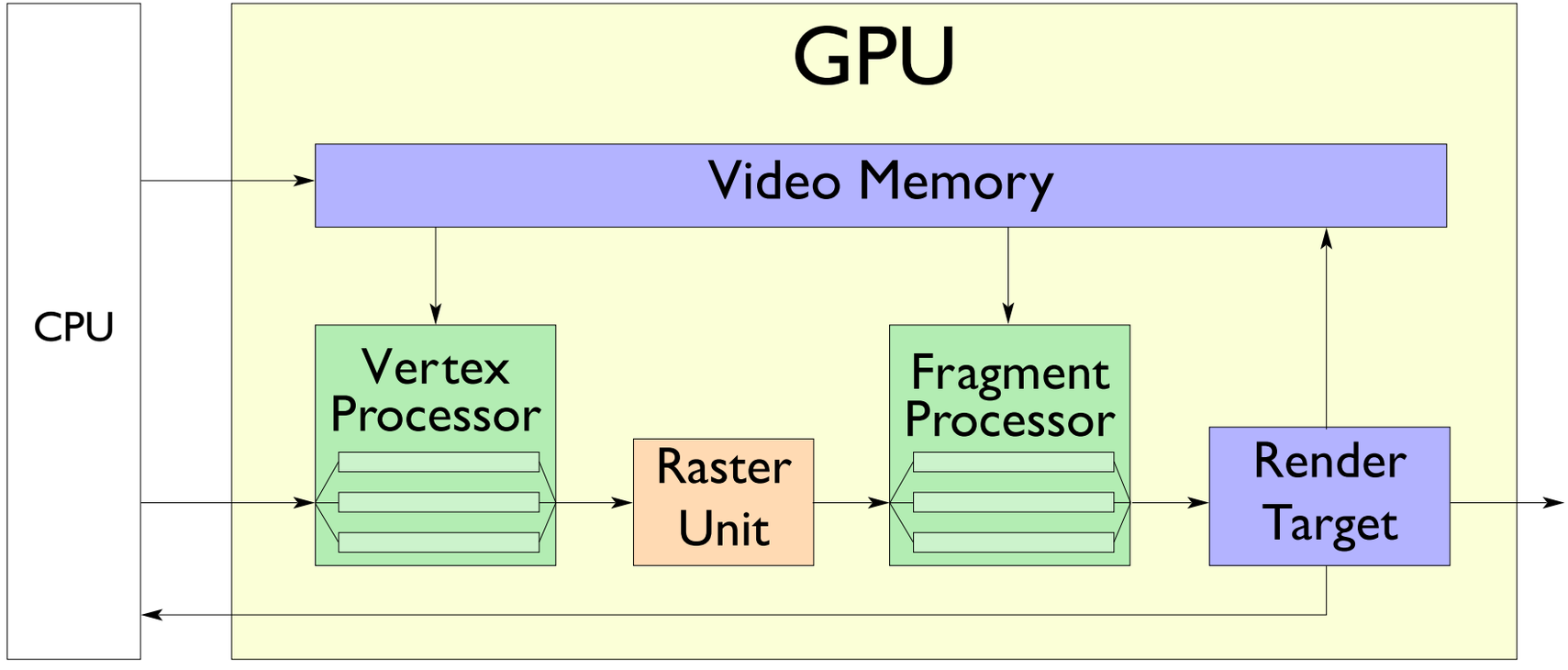
Motivation

- Exponentially growing compute power
- Very high memory bandwidth
- Ubiquity
  - Available in most PCs and workstations

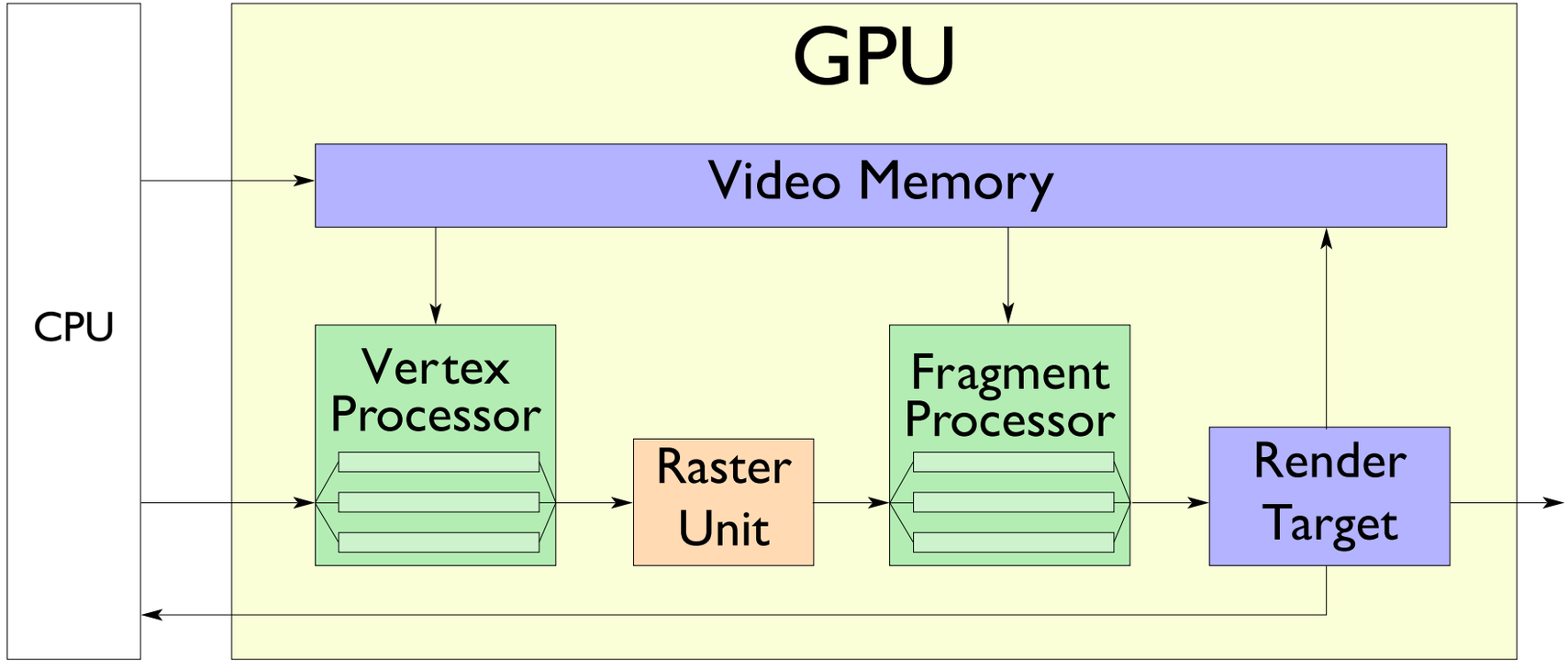
Motivation

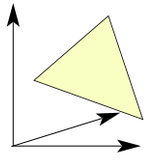
- Exponentially growing compute power
- Very high memory bandwidth
- Ubiquity
  - Available in most PCs and workstations
- Increasing programmability and functionality
  - Steadily improving precision
  - Full control flow (with small overhead)
  - High-level languages

GPU as Parallel Processor

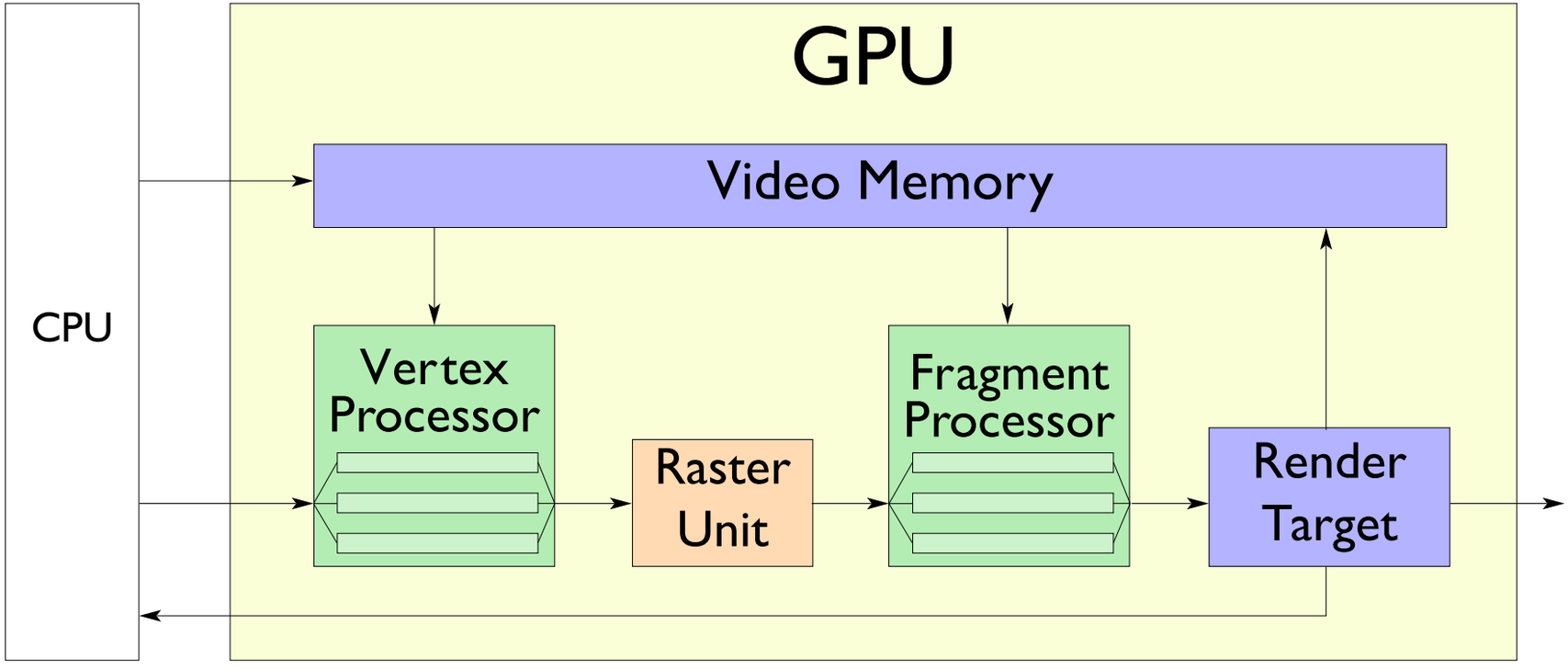


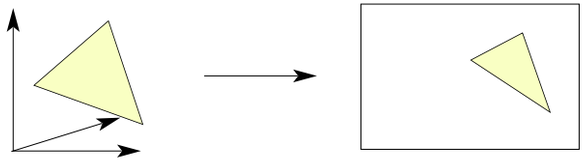
GPU as Parallel Processor



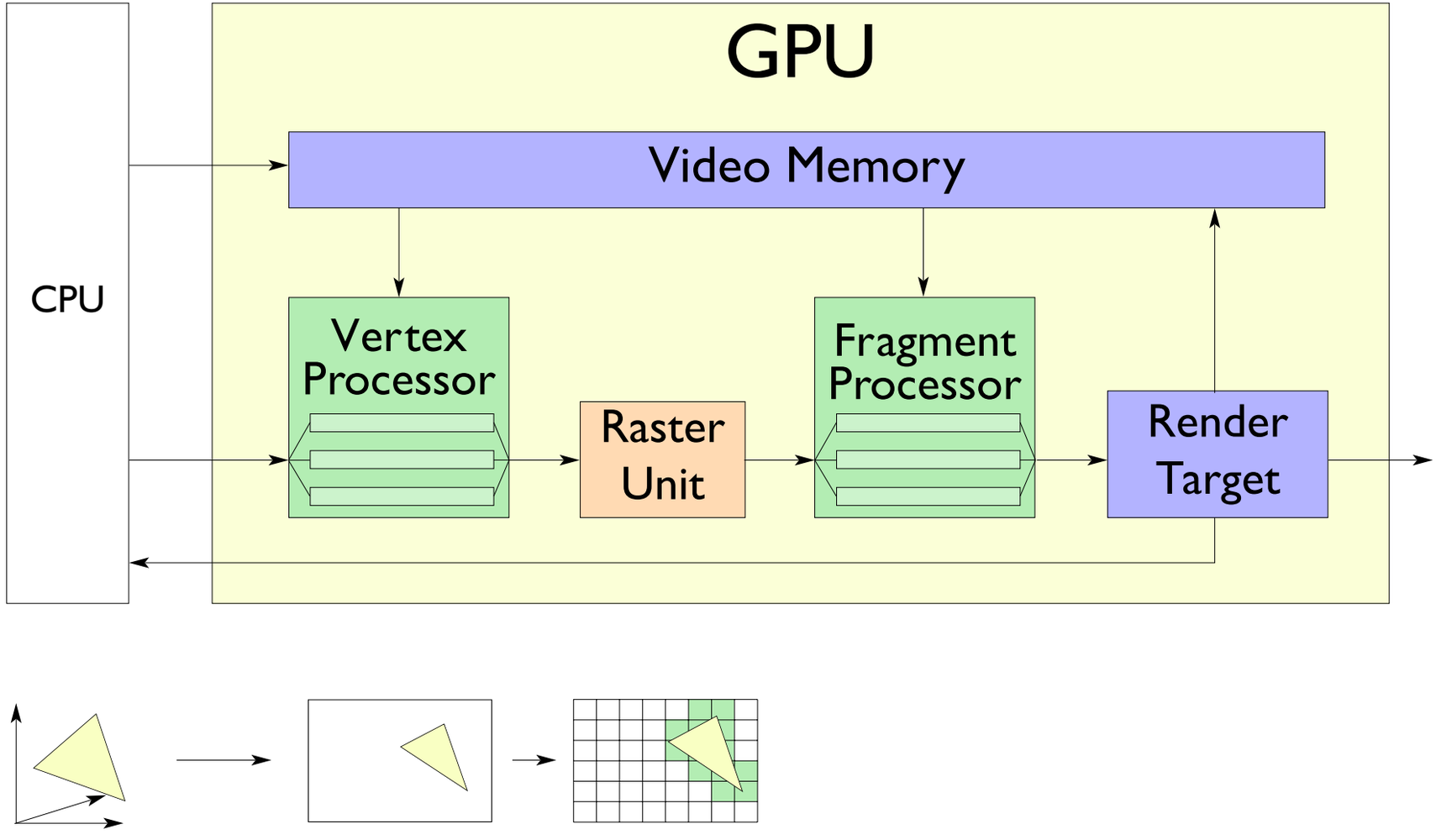


GPU as Parallel Processor

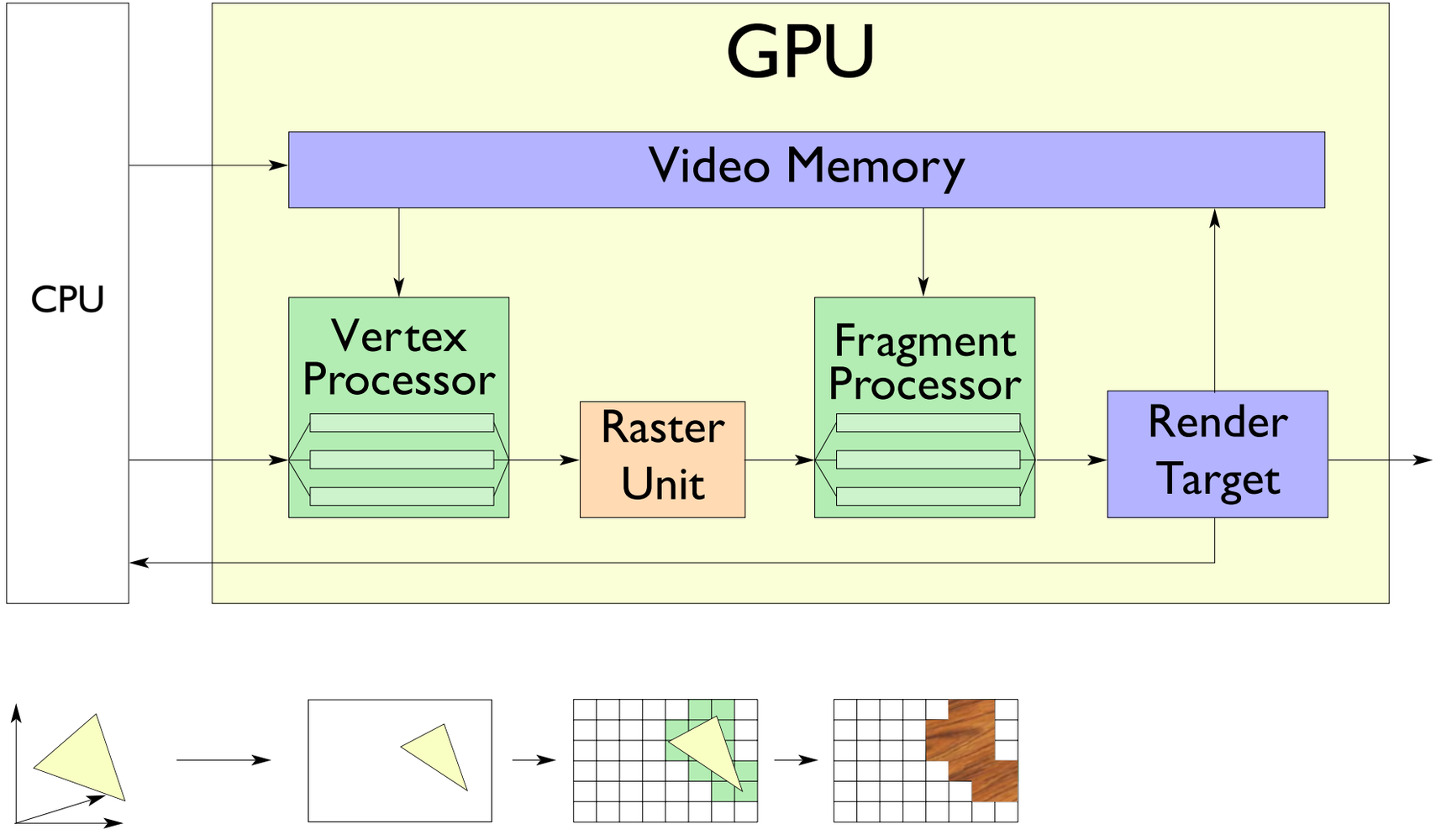




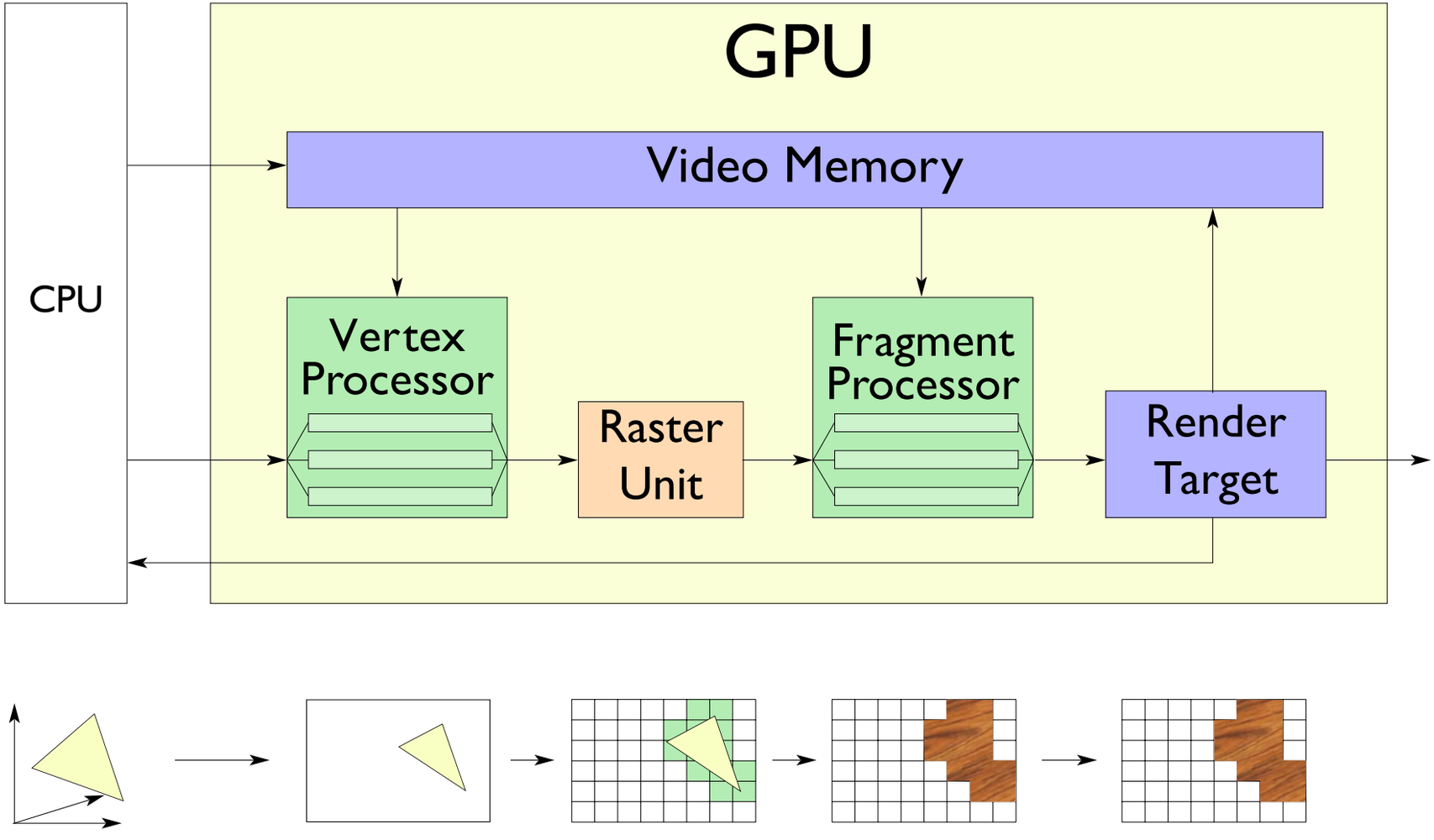
GPU as Parallel Processor



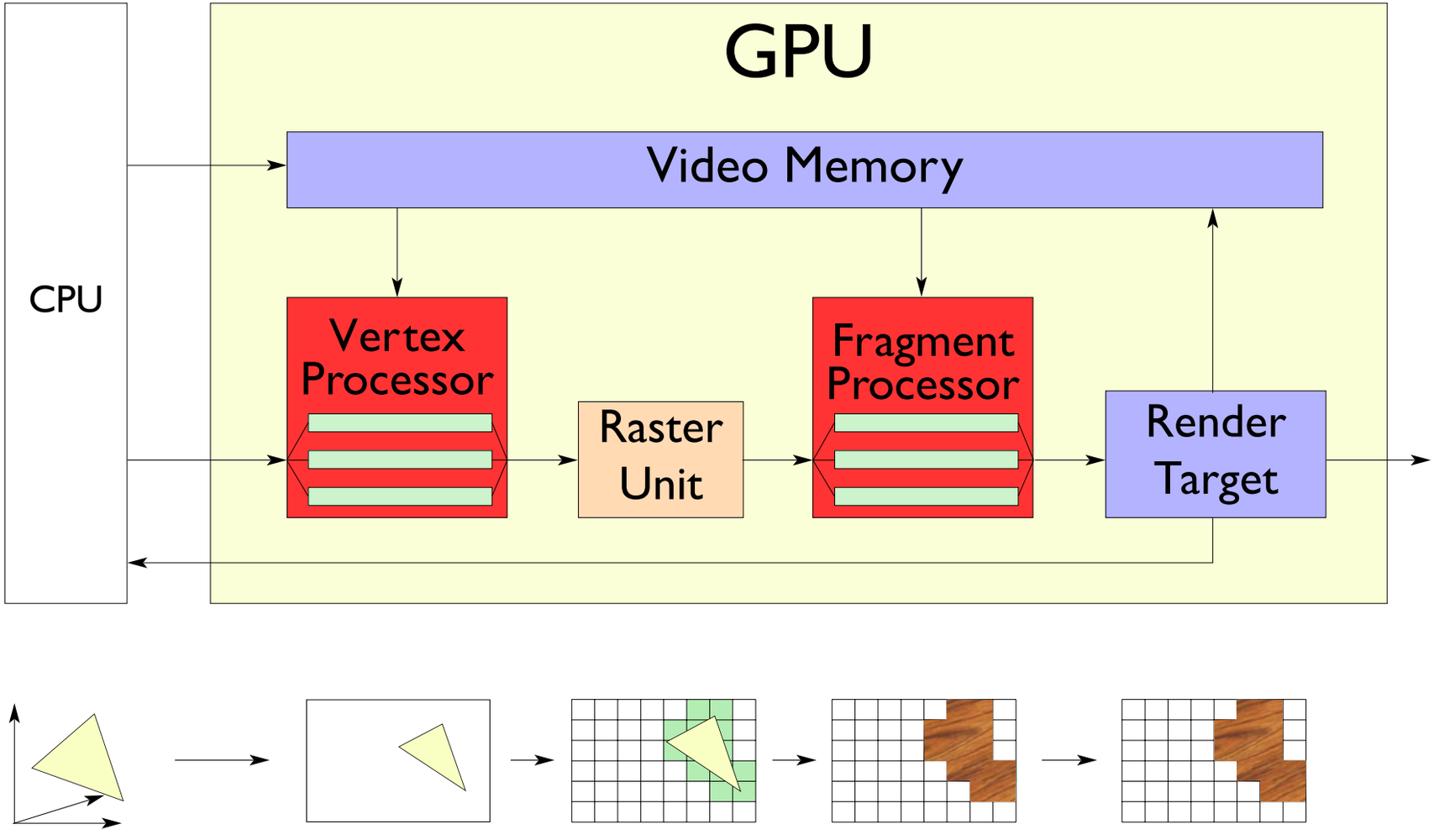
GPU as Parallel Processor



GPU as Parallel Processor



GPU as Parallel Processor



GPU as Parallel Processor

- Inherently parallel architecture
- Threads are processed in batches
  - No explicit thread creation
  - Automatic load balancing
- SPMD / SIMD data parallel programming model
  - Same operations are applied to all data items
  - Resembles stream programming model

GPU as Parallel Processor

- Fine-grain (data) parallelism
  - Millions of vertices and fragments are processed per frame (each with one thread)
  - Thousands of threads necessary to fully utilise available compute power and hide latencies
  - Hundreds of threads are executed in parallel in hardware
  - Thread creation has zero overhead

GPGPU Programming Model

- Match computations with graphics pipeline
  - Input data are stored in floating-point texture
  - Computations are invoked by geometry
  - Computations are performed in the fragment processor
  - Results are written to floating-point texture (bound as render target)

GPGPU Example

- Finite differences for 2D domain

$$\begin{aligned} T(i,j) &= T(i,j) + a_x \Delta T_x + a_y \Delta T_y \\ \Delta T_x &= T(i+1,j) + T(i-1,j) - 2T(i,j) \\ \Delta T_y &= T(i,j+1) + T(i,j-1) - 2T(i,j) \end{aligned}$$

- Domain is discretized
  - Maps naturally to 2D texture
- One pixel corresponds to one point in the discretized domain

GPGPU Example

- Kernel updates T(i,j) for one point

```
uint2 pos = glFragCoord;
```

GPGPU Example

- Kernel updates T(i,j) for one point

```
uint2 pos = glFragCoord;
float self = tex2D( domain, pos.x, pos.y).x;
```

GPGPU Example

- Kernel updates T(i,j) for one point

```
uint2 pos = glFragCoord;
float self = tex2D( domain, pos.x, pos.y).x;

// \Delta T_x
float dx = tex2D( domain, pos.x+1, pos.y).x;
dx += tex2D( domain, pos.x-1, pos.y).x;
dx -= 2.0 * self;
```

GPGPU Example

- Kernel updates T(i,j) for one point

```
uint2 pos = glFragCoord;
float self = tex2D( domain, pos.x, pos.y).x;

// \Delta T_x
float dx = tex2D( domain, pos.x+1, pos.y).x;
dx += tex2D( domain, pos.x-1, pos.y).x;
dx -= 2.0 * self;

// \Delta T_y
...
```

GPGPU Example

- Kernel updates T(i,j) for one point

```
uint2 pos = glFragCoord;
float self = tex2D( domain, pos.x, pos.y).x;

// \Delta T_x
float dx = tex2D( domain, pos.x+1, pos.y).x;
dx += tex2D( domain, pos.x-1, pos.y).x;
dx -= 2.0 * self;

// \Delta T_y
...

glFragColor = self + ax*dx + ay*dy;
```

GPGPU Limitations

- Highly specialised hardware architecture
  - "Fast path" is rendering and shading geometry
- Programs have to be written with graphics API
  - Steep learning curve for non-graphics people
  - Graphics API overhead
- Only gather, no scatter
  - Less flexibility
  - Makes it often necessary to re-design algorithms

GPGPU Limitations

- Computations have to be fully independent
  - No synchronisation, mutexes, ...
- Most GPGPU applications are bandwidth limited
  - GPGPU apps. read 32-bit floating point data from textures which is not common in graphics (and hence not the “fast path”)
  - Waste of compute power

Next Generation GPGPU

- Middleware
  - – Hide complexity of GPGPU through an additional software layer
  - – Examples: SH, Brook, ...
  - + Easy to realize
  - + Easier to write programs
  - + Hardware and vendor independent
  - — Graphics API and additional overhead
  - — Software-only solution

Next Generation GPGPU

- Middleware
- Non-graphics APIs
  - Expose GPU functionality through a non-graphics API
  - Example: **C**lose **T**o **M**etal (ATI)
  - **+** Avoids graphics API overhead
  - **+** Easier to learn for non-graphics people
  - **—** Software only
  - **—** Vendor specific

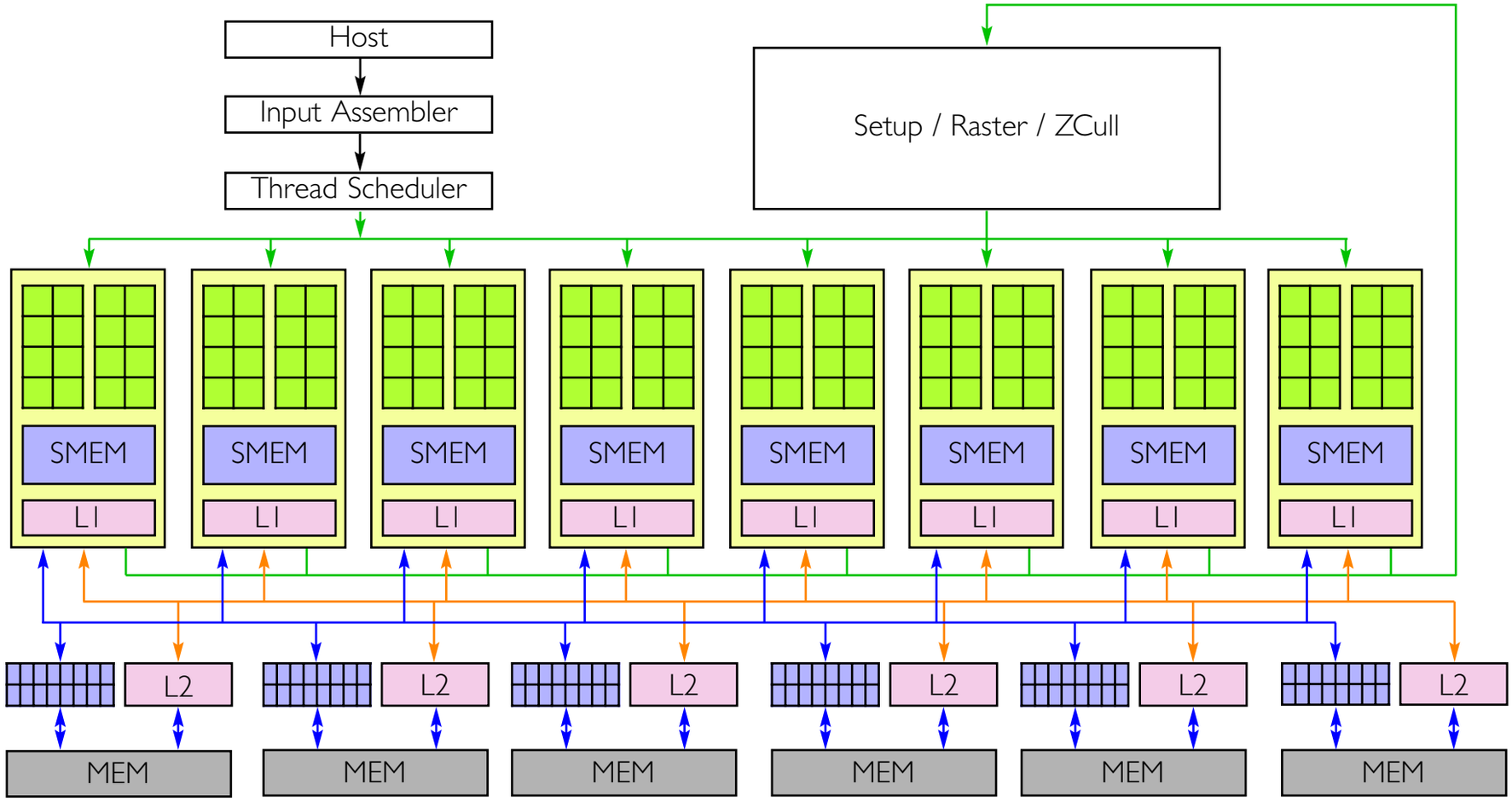
Next Generation GPGPU

- Middleware
- Non-graphics APIs
- GPU as hybrid device
  - – GPU as hybrid graphics and compute device
  - – Example: CUDA (NVIDIA)
  - + In principle, can overcome all limitations of traditional GPGPU
  - — Graphics still the “fast path”
  - — Vendor specific

CUDA Overview

- **C**ompute **U**nified **D**evice **A**rchitecture
- NVIDIA proprietary solution
- Combination of hardware and software features
s
- GPU as highly multithreaded coprocessor for data-parallel computations
  - Thousands of very lightweight threads
- Software provides low-level abstraction
  - Explicit parallelization
  - Explicit memory management

CUDA Hardware



CUDA Hardware

- Parallel program is executed as {1,2,3}D grid of thread blocks

s

- Threads in a thread block can
  - be synchronised using barriers
  - efficiently share data via shared memory
- Each thread has unique {1,2,3}D identifier
  - For example to determine the data that is processed by the thread
- Atomic instructions

CUDA Memory (host-visible)

- Global Memory (RW)
  - Direct-access on-board RAM memory
  - High-latency, uncached
- Texture Memory(RO)
  - RAM memory accessed with special interface
  - Medium-latency, cached
- Constant Memory (RO)
  - Low-latency (coherent access), cached

CUDA Shared Memory

- Communicate data between threads
  - Limited to threads in one thread block
- User-managed cache
- Very low latency (approx. same as registers)
- Invisible to host
  - Has to be initialised by thread block
- Very limited in size: currently 16 KB

CUDA Software

- Extension of C realized as combination of intrinsics and API
  - Compiled using meta-compiler
- Goal: Easy port of C programs to CUDA

CUDA Host Extensions

- Kernel invocation

```
myKernel<<< grid_dim , block_dim >>>(in, out)
```

- Memory management

```
cudaMalloc(), cudaMemcpy(), cudaFree(), ...
```

- Device management

```
cudaGetDeviceCount(), ...
```

- Graphics interoperability

```
cudaGLRegisterBufferObject(), ...
```

- ...

CUDA Device Extensions

- Memory declaration

  ```
  __shared__ float smem[1024]
  ```

- synchronisation (barrier)

  ```
  __synchthreads()
  ```

- Atomic operations

  ```
  atomicAdd(), atomicExch(), atomicXor(), ...
  ```

- Thread identifier

  ```
  threadIdx, blockIdx, blockDim, ...
  ```

- ...

CUDA Program Flow

- Upload data to process into device memory
- Define execution environment (#threads etc.)
- Launch kernel
  - Read input data into shared memory
  - Process data
  - Write result from shared to global memory
- Read result back to host memory

CUDA Example

- Simple 1D convolution

$$a[x] = b[x-1] + 3.0b[x] + b[x+1]$$

CUDA Example

- Simple 1D convolution

$$a[x] = b[x-1] + 3.0b[x] + b[x+1]$$

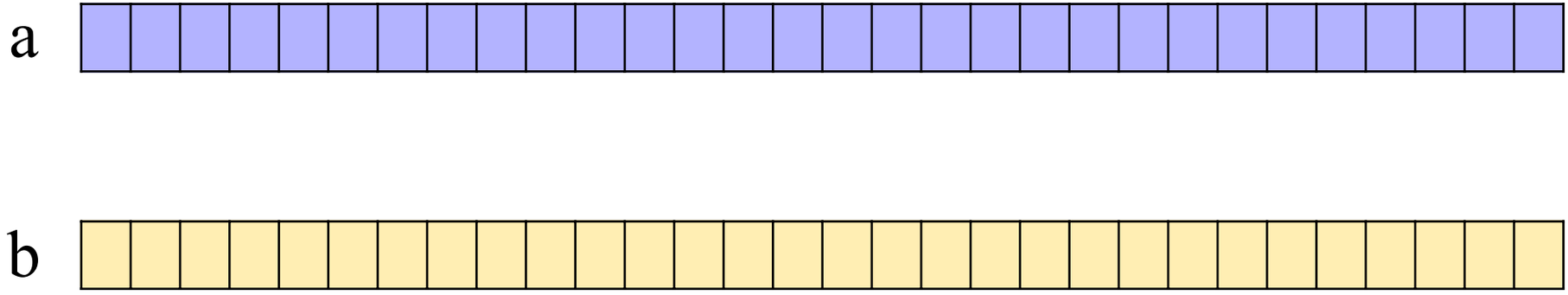
- Assume infinite signals and ignore necessary block overlap

CUDA Example

- Simple 1D convolution

$$a[x] = b[x-1] + 3.0b[x] + b[x+1]$$

- Assume infinite signals and ignore necessary block overlap

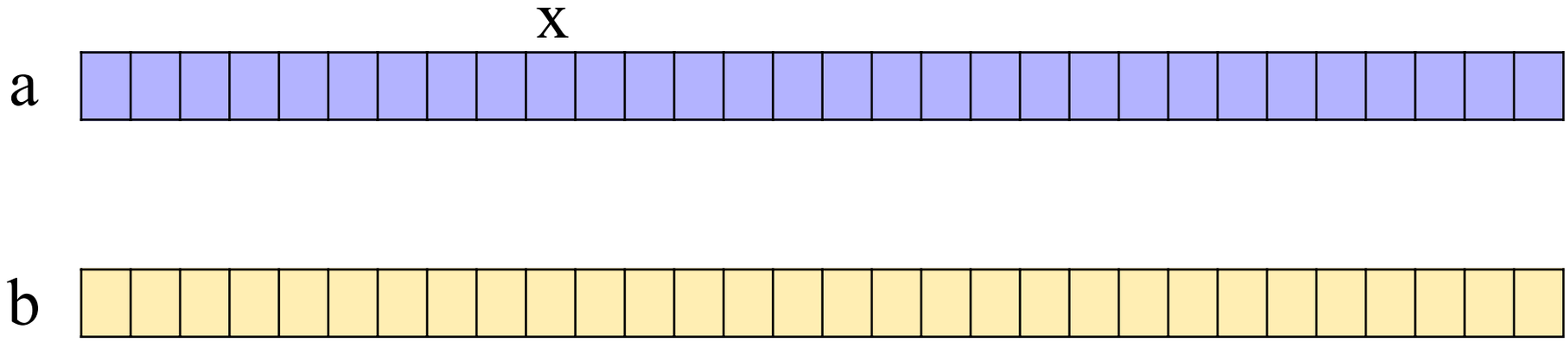


CUDA Example

- Simple 1D convolution

$$a[x] = b[x-1] + 3.0b[x] + b[x+1]$$

- Assume infinite signals and ignore necessary block overlap

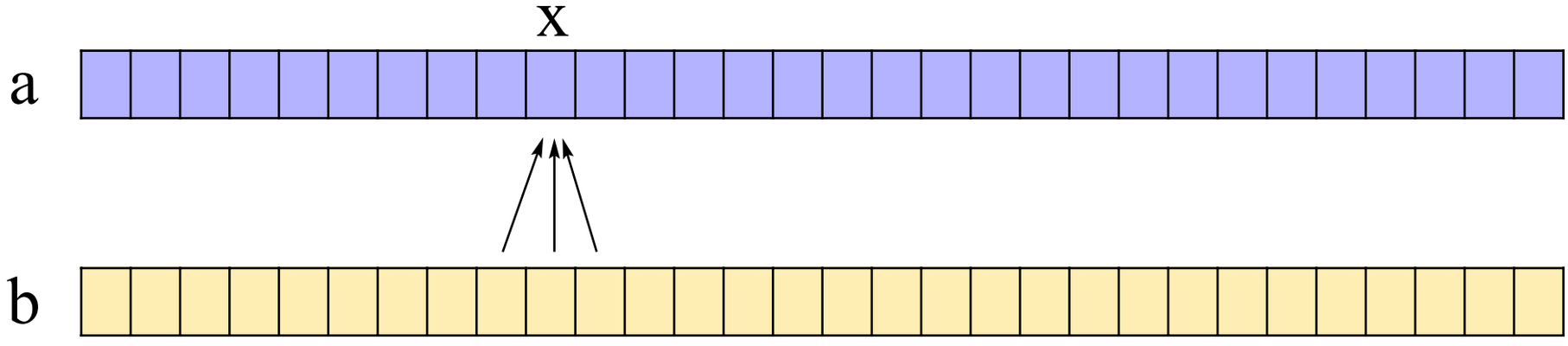


CUDA Example

- Simple 1D convolution

$$a[x] = b[x-1] + 3.0b[x] + b[x+1]$$

- Assume infinite signals and ignore necessary block overlap

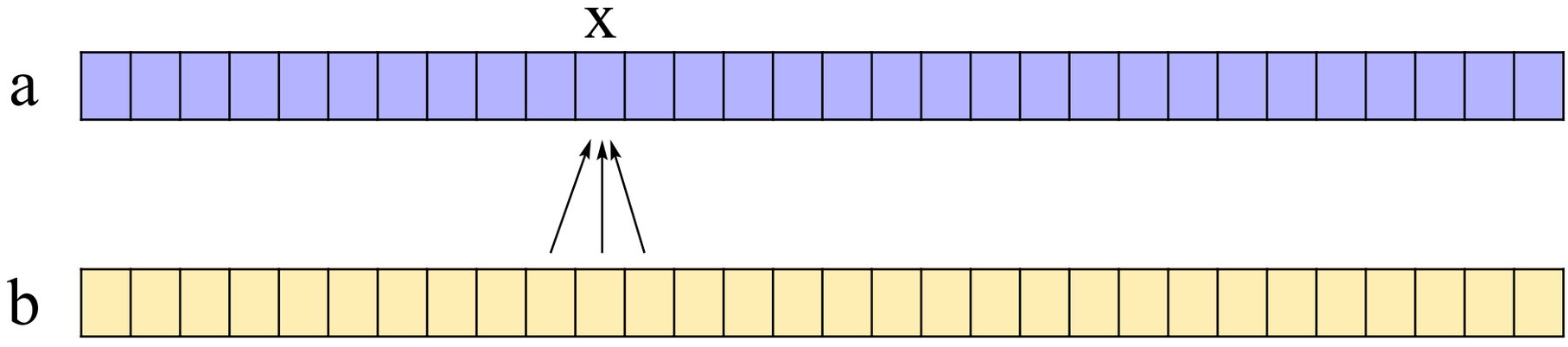


CUDA Example

- Simple 1D convolution

$$a[x] = b[x-1] + 3.0b[x] + b[x+1]$$

- Assume infinite signals and ignore necessary block overlap

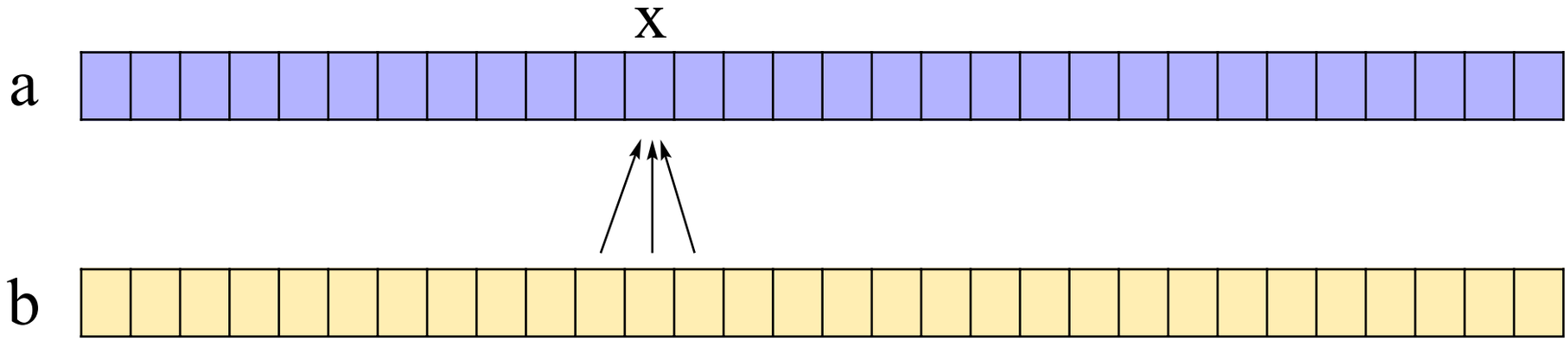


CUDA Example

- Simple 1D convolution

$$a[x] = b[x-1] + 3.0b[x] + b[x+1]$$

- Assume infinite signals and ignore necessary block overlap

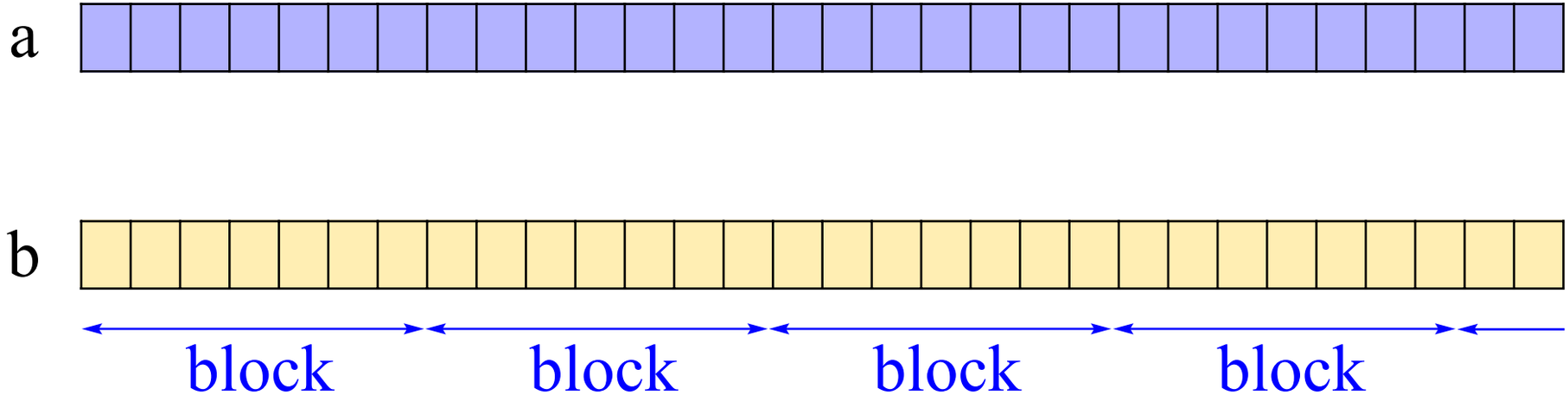


CUDA Example

- Simple 1D convolution

$$a[x] = b[x-1] + 3.0b[x] + b[x+1]$$

- Assume infinite signals and ignore necessary block overlap



CUDA Example

```
__global__ void conv( float* in, float* out) {
```

CUDA Example

```
__global__ void conv( float* in, float* out) {

  __shared__ float smem[1024];
```

CUDA Example

```
__global__ void conv( float* in, float* out) {

  __shared__ float smem[1024];

  tid = threadIdx.x + blockDim.x * blockIdx.x;
  smem[tid] = in[tid];
  __syncthreads();
```

CUDA Example

```
__global__ void conv( float* in, float* out) {

  __shared__ float smem[1024];

  tid = threadIdx.x + blockDim.x * blockIdx.x;
  smem[tid] = in[tid];
  __syncthreads();

  // ignore tile boundary
  float res = smem[tid-1];
  res += 3.0 * smem[tid];
  res += smem[tid+1];
```

CUDA Example

```
__global__ void conv( float* in, float* out) {

  __shared__ float smem[1024];

  tid = threadIdx.x + blockDim.x * blockIdx.x;
  smem[tid] = in[tid];
  __syncthreads();

  // ignore tile boundary
  float res = smem[tid-1];
  res += 3.0 * smem[tid];
  res += smem[tid+1];

  out[tid] = res;
}
```

CUDA Example

```
__host__ void run() {

  // read input data
  ...

  unsigned int mem_size = signal_size *
                                  sizeof(float);
```

CUDA Example

```
__host__ void run() {

  // read input data
  ...

  unsigned int mem_size = signal_size *
                                  sizeof(float);

  float* d_in, d_out;
  cudaMalloc( (void**) d_in, mem_size);
  cudaMalloc( (void**) d_out, mem_size);
  cudaMemcpy( d_in, signal, mem_size,
              cudaMemcpyHostToDevice);
```

CUDA Example

```
// assume signal_size > 512 and not a
// multiple of 512
dim3 threads, blocks;
threads = 512;
blocks = (signal_size / 512) + 1;
```

CUDA Example

```
// assume signal_size > 512 and not a
// multiple of 512
dim3 threads, blocks;
threads = 512;
blocks = (signal_size / 512) + 1;

// run kernel
conv<<<blocks,threads>>>( d_in, d_out);
```

CUDA Example

```
    // assume signal_size > 512 and not a
    // multiple of 512
    dim3 threads, blocks;
    threads = 512;
    blocks = (signal_size / 512) + 1;

    // run kernel
    conv<<<blocks,threads>>>( d_in, d_out);

    // copy result back to the host array result
    cudaMemcpy( result, d_out, mem_size,
                cudaMemcpyDeviceToHost);
}
```

Conclusion

- Traditional GPGPU
  - Potential for significant speedups for data-parallel problems
  - Hard to use

Conclusion

- Traditional GPGPU
  - Potential for significant speedups for data-parallel problems
  - Hard to use
- Next generation GPGPU
  - Alleviates many of the problems of traditional GPGPU
  - Practicality for real-world problems has yet to be shown

Conclusion

- Many open questions for the future:
  - Vendor-independent APIs?
  - Practical debugger?
  - Balancing CPU and GPU?
  - What about task parallelism?
  - How long does it make sense to have one chip for graphics and general purpose computations?
  - ...

Slides available at

www.dgp.toronto.edu/people/lessig/talks/