Vector Processing

Computer Organization

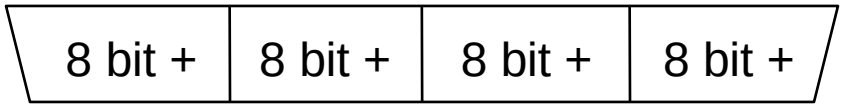
Tuesday, 22 October 2024

Many slides adapted from:
Computer Organization and Design,
Patterson & Hennessy
5th Edition, © 2014, MK
and from Prof. Mary Jane Irwin, PSU

TÉCNICO LISBOA

Multimedia SIMD Extensions

- SIMD: single instruction, multiple data streams
  - single control unit broadcasting operations to multiple datapaths
- The most widely used variation of SIMD is found in almost every microprocessor today
  - Basis of MMX and SSE instructions added to improve the performance of multimedia programs
  - A single, wide ALU is partitioned into many smaller ALUs that operate in parallel



  - Loads and stores are simply as wide as the widest ALU, so the same data transfer can transfer one 32 bit value, two 16 bit values or four 8 bit values
- There are now hundreds of SSE instructions in the x86 to support multimedia operations

Vector Processors

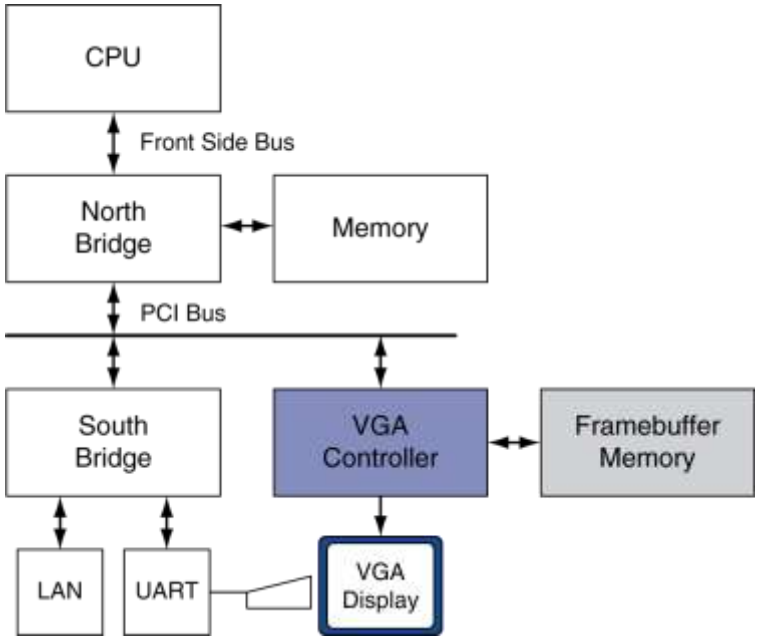
- Vector processors:
  - Highly-pipelined ALU, to get good performance at lower cost
  - Set of vector registers to hold the operands and results
    - Collect the data elements from memory
    - Put them in order into a large set of registers
    - Operate on them sequentially in registers
    - Then write the results back to memory
  - Formed the basis of supercomputers in the 1980's and 90's

- VMIPS: extending the MIPS instruction set to include vector instructions
  - `addv.d` to add two double precision vector register values
  - `addvs.d` and `mulvs.d` to add (or multiply) a scalar register to (by) each element in a vector register
  - `lv` and `sv` do vector load and vector store and load or store an entire vector of double precision data

History of GPUs

- Early video cards
  - Frame buffer memory with address generation for video output
- 3D graphics processing
  - Originally high-end computers (e.g., SGI)
  - Moore's Law ⇒ lower cost, higher density
  - 3D graphics cards for PCs and game consoles
- Graphics Processing Units
  - Processors oriented to 3D graphics tasks
  - Vertex/pixel processing, shading, texture mapping, rasterization

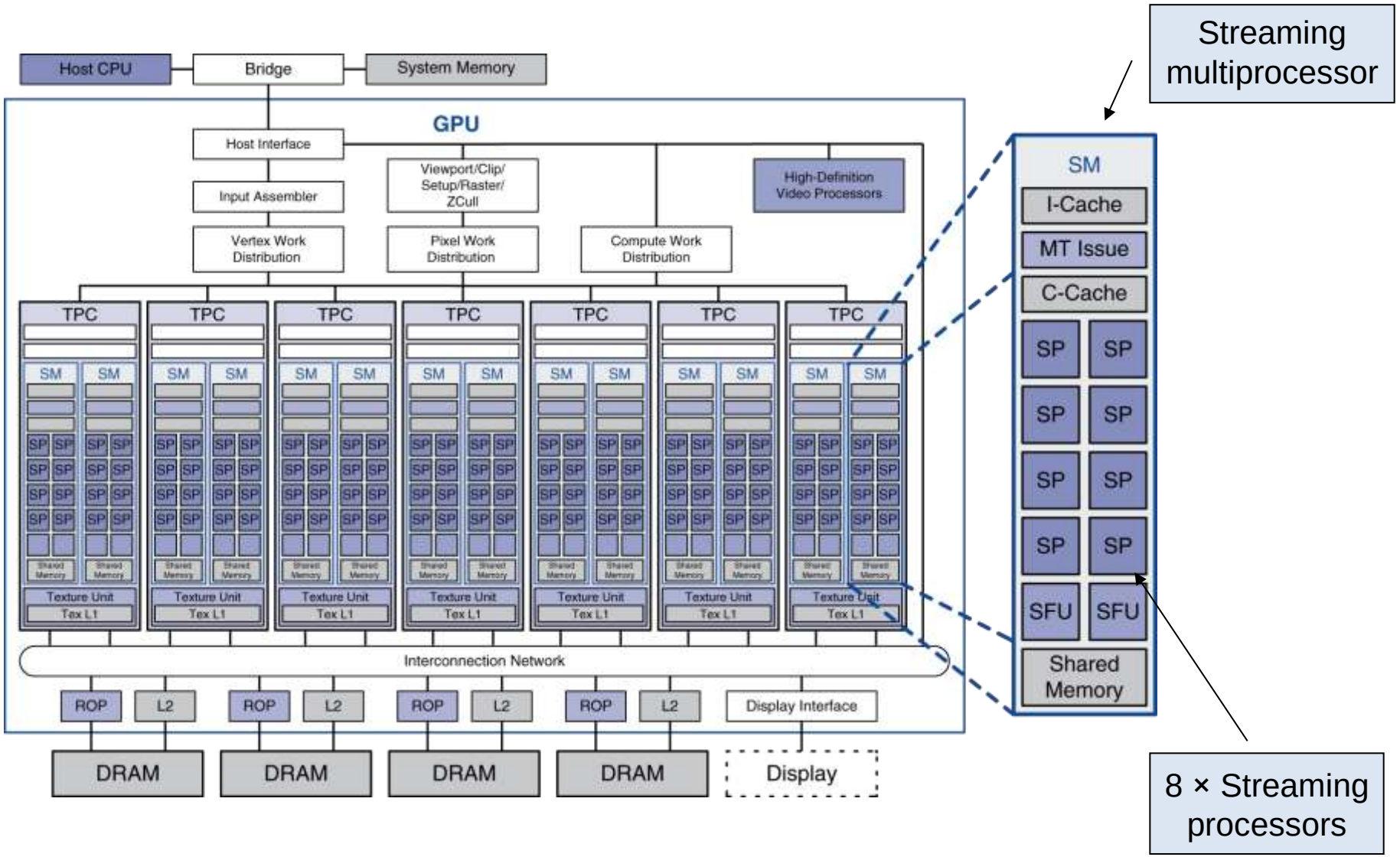
Graphics in the System



GPU Architectures

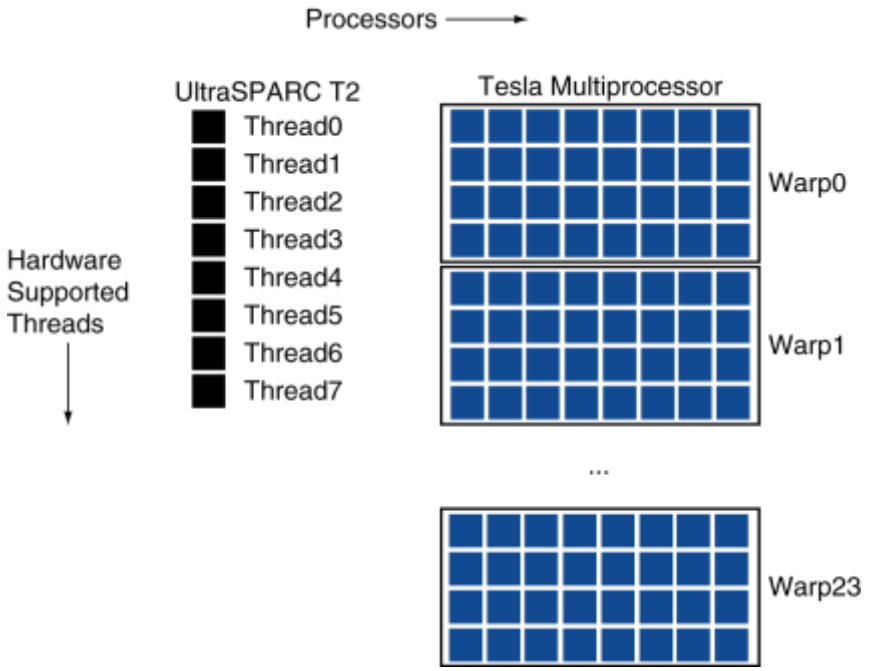
- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code
- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)

Example: NVIDIA Tesla



Example: NVIDIA Tesla

- Streaming Processors
  - Single-precision FP and integer units
  - Each SP is fine-grained multithreaded
- Warp: group of 32 threads
  - Executed in parallel, SIMD style
    - 8 SPs
      × 4 clock cycles
  - Hardware contexts for 24 warps
    - Registers, PCs, …



GPGPU Programming Model

- General Purpose GPU (GPGPU) programming model reflects GPU hardware architecture:
  - GPU seen as massively data parallel co-processor
  - large local memory
  - CPU batches threads to the GPU, together with the data to process
    - GPU threads extremely light-weight (little overhead)
    - GPU requires 1,000s of threads for full efficiency

GPGPU Programming Difficulties

- Drawbacks of the GPGPU approach:
  - Tough learning curve, particularly for those outside graphics
  - Need to SIMD-ipify code
  - Highly constrained memory layout and access model
  - Need for many passes drives up bandwidth consumption

GPGPU Programming Example

```
#include "cuda_runtime.h"

#include "basic_timer.hpp"
#include "idivup.hpp"
#include "mini_cutil.h"

#include <cmath>
#include <cstdio>

// calculates one exp per thread on the GPU.
__global__ void exp_kernel(float * v, int size)
{
    int const t = threadIdx.x +
                  blockIdx.x * blockDim.x;
    if (t < size)
        v[t] = exp(-(float)t) ;
}

int main(int argc, char * argv[])
{
    int n_elem = 10000 ;
    if (argc > 1)  n_elem = atoi(argv[1]);

    // Allocate memory on the device
    float * vector_on_gpu = 0;
    cudaError_t err = cudaMalloc(
        &vector_on_gpu, n_elem*sizeof(float));
    // ... check error
```

```
// Setup execution configuration
dim3 block_size(256) ; // <-- how to pick this number?
dim3 grid_size(idivup(n_elem, block_size.x)) ;

// Launch the GPU computation.
exp_kernel<<<grid_size, block_size>>>(vector_on_gpu, n_elem);

// Wait for the GPU to finish.
err = cudaThreadSynchronize();
// check error ...

float *vector_on_cpu = (float*) malloc(n_elem*sizeof(float));
// Copy the results back to the CPU.
err = cudaMemcpy(
    (void*)vector_on_cpu, /// destination on the CPU
    (void*)vector_on_gpu, /// source on the GPU
    n_elem * sizeof(float), /// copy size
    cudaMemcpyDeviceToHost) ; /// copy direction*//

cudaFree((void*)vector_on_gpu) ; // Free the gpu memory

// Now do the same on the cpu to compare times
basic_timer timer ;
for (int t = 0 ; t < n_elem ; ++t)
    vector_on_cpu[t] = exp(-(float)t) ;
double elapsed = timer.elapsed() ;
printf("cpu time %gs", elapsed) ;

free(vector_on_cpu);
}
```

Vector Processing

Computer Organization

Tuesday, 22 October 2024

Many slides adapted from:
Computer Organization and Design,
Patterson & Hennessy
5th Edition, © 2014, MK
and from Prof. Mary Jane Irwin, PSU

TÉCNICO LISBOA

Other computing styles:

Analog - https://youtu.be/GVsUOuSjvcg

Quantum - https://youtu.be/g_IaVepNDT4

(24 Oct 2023)

Computer Organization

Tuesday, 22 October 2024

TÉCNICO LISBOA