GPUs

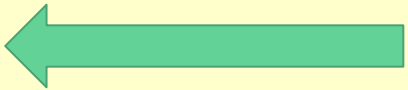
JP Bulman, Edward Krawczyk, Andrew Moore, Alex Osler, Alex Spielman, Cole Winsor, Eda Zhou

CS4515 - Computer Architecture, 2019 April 12

What is a GPU?

Flynn's Taxonomy

- **Single instruction stream, single data stream (SISD)**
- **Single instruction stream, multiple data streams (SIMD)**
  - Vector architectures
  - Multimedia extensions
  - Graphics processor units

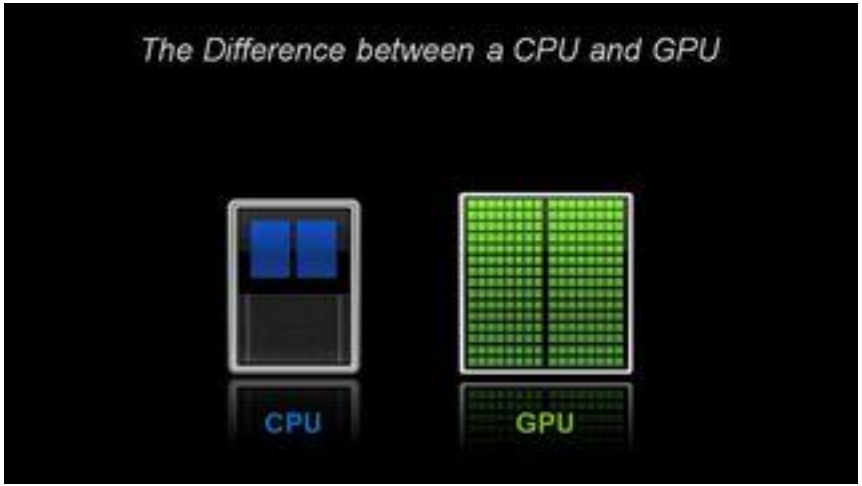


- **Multiple instruction streams, single data stream (MISD)**
  - No commercial implementation
- **Multiple instruction streams, multiple data streams (MIMD)**
  - Tightly-coupled MIMD
  - Loosely-coupled MIMD

These are key terms for all Computer Science topics.

GPU versus CPU

- GPU: Optimizes for special computing case
  - Graphics . . .
  - Or any operation that requires operating on very large independent sets of numbers
- CPU: Optimizes for general computing
- GPU requirements evolved as computers changed



Source: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/

History

- Needed to convert data into images
- Dedicated video chips emerged in the 70s
- GPUs quickly evolved to accomplish more things
  - 3D Graphics
  - Shading
  - Ray Tracing
- Exponential increases in capability

MOS Technology VIC-II

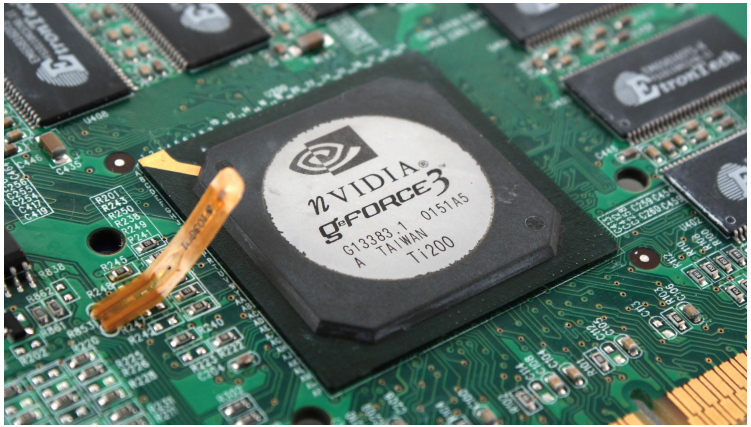
- Released in 1982
- 1 MHz clock
- 16 kB of RAM
- 320 x 200 pixel resolution
- 40 x 25 characters
- 16 colors
- Incapable of 3D

Reality Coprocessor

- Released in 1996
- 62.5 MHz Clock
- 4.5 MB (UMA) RAM
- Vector Processing
- Anti-aliasing, texturing
- Also handled sound!

NVIDIA GeForce 3 Series

- Released in 2001
- First unit with programmable vertex shading
- 64MB DDR RAM
- 200 MHz Clock

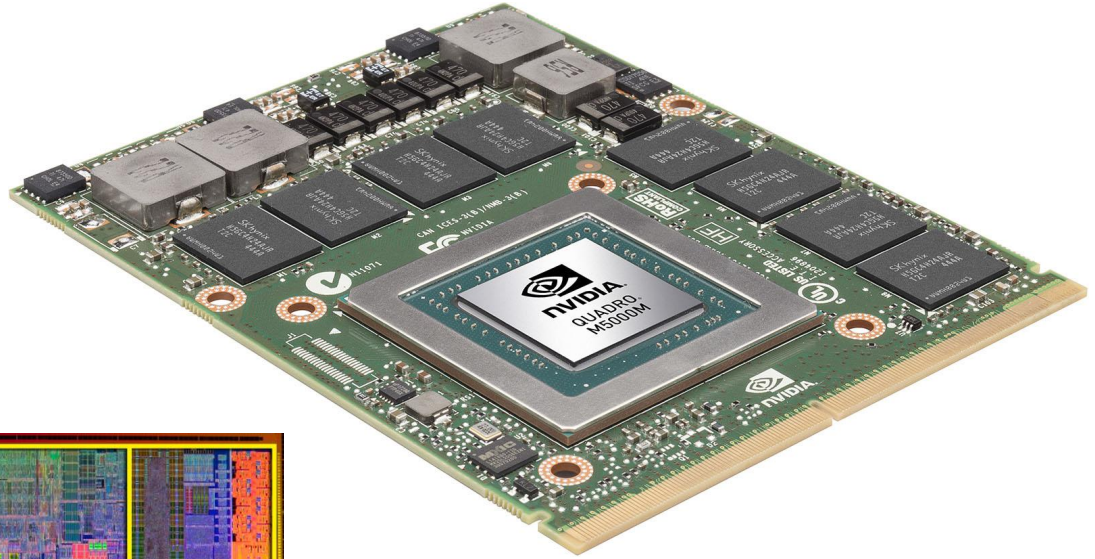


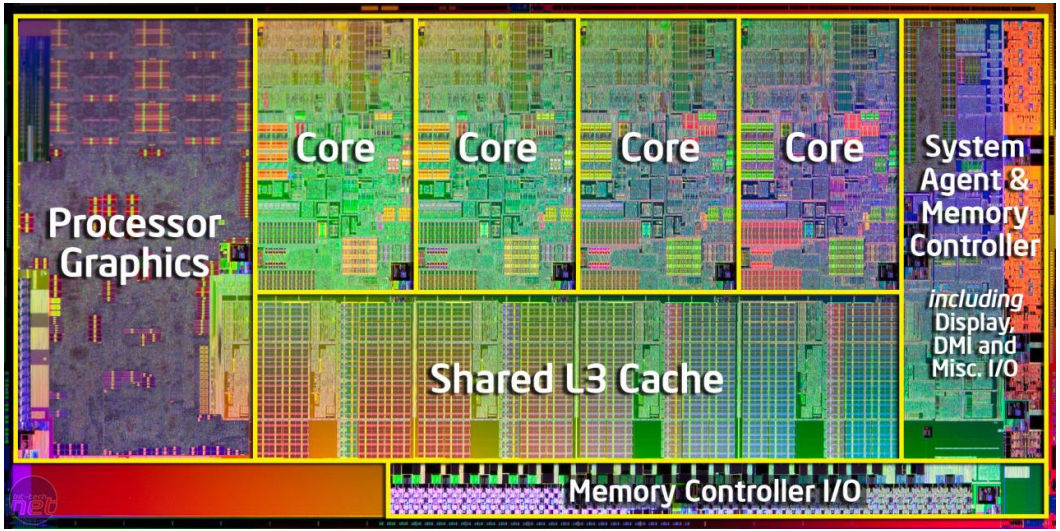
NVIDIA RTX 2080

- Cutting edge technology (2018)
- Ray Tracing
- 1.5 GHz Clock
- 11 GiB GDDR6 RAM

Other Innovations

- Mobile Graphics Chips
- Integrated Graphics





How GPUs operate

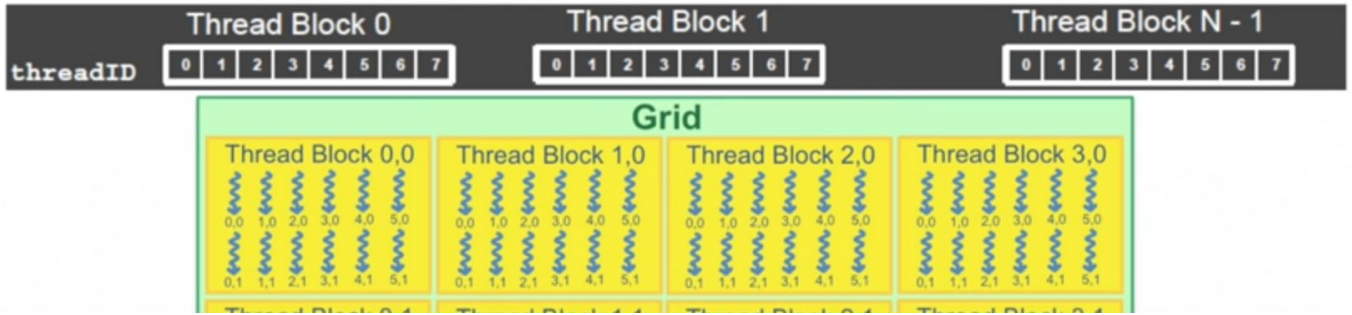
- CPU ran more on basis of sequential serial processing
- GPU moved to the idea of smaller, more efficient cores that run in parallel
- Most popular frameworks include:
  - OpenACC, OpenCL and CUDA by NVIDIA
- Data-level Parallel processing is a large reason for GPUs being so fast at their job

Which will deliver pizza faster?

or

Programming the GPU

- Specialized to leverage data level parallelism in SIMD stream
  - Independent data streams (no order)
- CUDA (Compute Unified Device Architecture)
  - Developed by NVIDIA
- Based around CUDA threads
  - Threads each have an id
    - Get memory addresses and make decisions
  - Organizes threads ➔ thread blocks ➔ grids



- Grid can access memory space private to the block
- Programmers to specify grid block organization on each kernel call.

Programming the GPU

- CUDA has C like syntax
- __host__, __device__ and __global__ distinguish between CPU and GPU
- **Shaders**: Functions that are run on the GPU
- Calling __global__ functions

  name <<<dimGrid, dimBlock>>>(parameters)

- Getting data stream in the shader

  blockIdx, threadIdx, and blockDim

Examples

```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n, double a, double *x, double *y)
{
          for (int i = 0; i < n; ++i)
                y[i] = a*x[i] + y[i];
}
```

Traditional C code

```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
   daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__global__
void daxpy(int n, double a, double *x, double *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if (i < n) y[i] = a*x[i] + y[i];
}
```

Same function using CUDA

Frameworks and Libraries

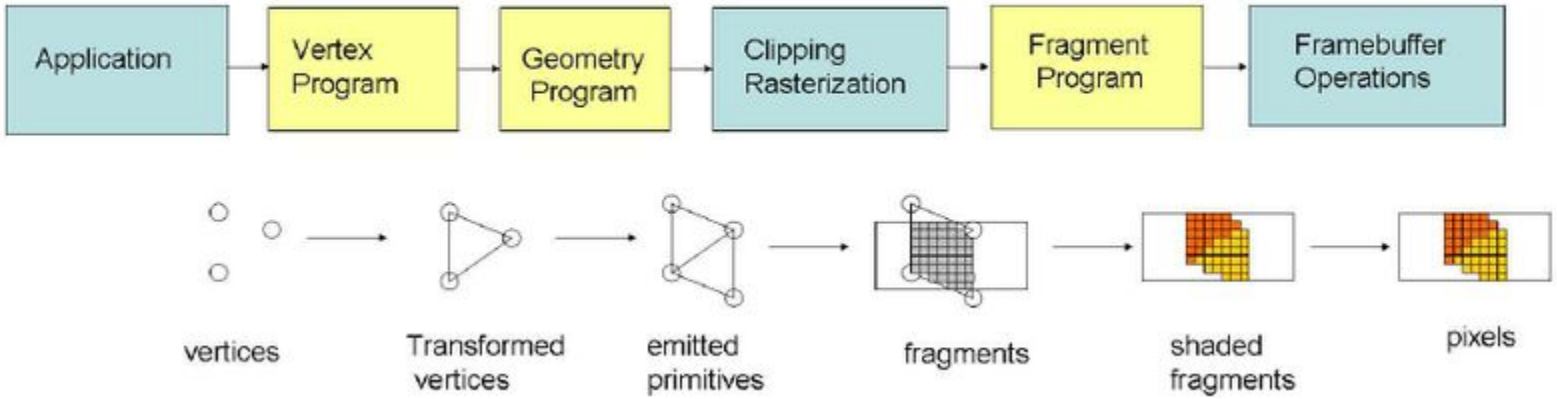
- Frameworks
  - CUDA
  - OpenCL
  - Harlan
- Libraries
  - OpenGL

OpenCL

OpenGL®

Example of Drawing with a GPU

- Main goal of GPU is graphics
- **WebGL**: OpenGL for javascript.
- Graphics Pipeline



GPU versus CPU Clock Cycles

- What should be calculated where?
  - Calculating model matrix
- How much do we need GPUs?
  - XBox 360, Xenos GPU

1.2 billion vertices/sec

1 matrix transformation per vertex, 16 multiplications per transform

1.2 * 16 = 19.2 billion multiplies per second, roughly 115 billion cycles

Xbox CPU: 3.2 GHz

Passing Data to GPU

JavaScript

```
function bindPoints(){
    let vBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, flatten(points), gl.STATIC_DRAW);

    let vPosition = gl.getAttribLocation(program, name: "vPosition");
    gl.vertexAttribPointer(vPosition, size: 4, gl.FLOAT, normalized: false, stride: 0, offset: 0);
    gl.enableVertexAttribArray(vPosition);
}
```

Shader

```
attribute vec2 vTexCoord;
varying vec2 fTexCoord;
uniform mat4 projMatrix;
```

Vertex Shader

- Operates on each Vertex
- Spatial Transformations
- Lighting, Reflections, etc

```
void main() {
    gl_Position = projMatrix * viewMatrix * modelMatrix * vPosition;
    gl_PointSize = vPointSize;

    //Convert the vertex position to eye coordinates
    vec3 pos =  (viewMatrix * modelMatrix * vPosition).xyz;

    vec3 lightPos = (viewMatrix * lightPosition).xyz;

    //Calculate L
    vec3 L = normalize(lightPos.xyz - pos);

    //Calculate V
    vec3 V = normalize(-pos);

    //Convert vertex normal to eye coordinates
    vec3 N = normalize((viewMatrix * modelMatrix * vNormal).xyz);

    //Calculate reflection vector
    vec3 R = (2.0 * dot(L, N) * N) - L;

    vec4 diffuse = diffuseProduct * max(dot(L, N), 0.0);
    vec4 specular = specularProduct * pow(max(dot(V, R), 0.0), shininess);
    vec4 ambient = ambientProduct;
```

Fragment Shader

- Operates on each Pixel
- Color, Textures, Reflection, etc

```
void main()
{
    if(tmapped > 0.0){
        if(texture == 0.0){
            gl_FragColor = texture2D( tex0, fTexCoord );
        }
        if(texture == 1.0){
            gl_FragColor = texture2D( tex1, fTexCoord );
        }
    }
    else{
        vec4 color = fColor;

        if(reflOn > 0.0){
            vec4 texColor = textureCube(texMap, refl);
            color = color*texColor;
        }
        if(refrOn > 0.0){
            vec4 texColor = textureCube(texMap, refr);
            color = color*texColor;
        }
        gl_FragColor = color;
    }
}
```

Questions?

GPU Architecture

- Data Level Parallelism
- SIMD
- Vector Architecture
- GPUs *“share features with vector architectures, they have their own distinguishing characteristics, in part because of the ecosystem in which they evolved”* (page 282)

Vector Architecture

source

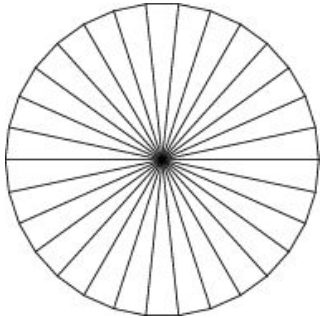
$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} x_1 & x_2 & \cdots & x_m \end{bmatrix}.$$

A column vector transposed is just a row vector

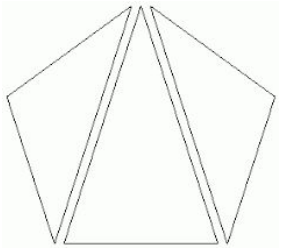
- What is a vector?
  - In math, it is a 1 x n or n x 1 matrix
  - Which for computer scientists, is just a one dimensional array
  - MA 2071
- What is vector architecture?
  - Take pieces of data anywhere from memory and place them into adjacent register files
  - Now their group is a ‘vector’
  - Then, operate on all of the data in parallel
  - Finally, update memory with the corresponding results

Under The Hood

- Points can be represented in space as a vector(x, y, z, w)
  - Homogeneous coordinates
  - w is similar to scalar
    - Done ‘in house’ - even better
  - Advantage of near infinity points
- Make several points and connect them to make shapes
  - Most objects are made from triangles
  - Least number of points needed to make a 2D shape
  - Can give illusion of any shape being made of triangles



source



source

Translations

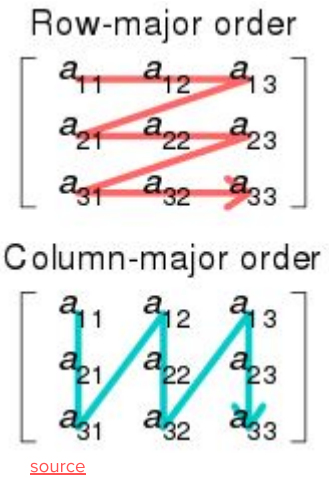
$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix} = \begin{pmatrix} v_x + t_x \\ v_y + t_y \\ v_z + t_z \\ 1 \end{pmatrix}$$

source

- Need to move a point to a new location
  - Deconstructing and reconstructing is inefficient
    - At least twice as much memory and a lot longer than translation
  - Instead, make the point a translation matrix multiplied by itself
- Matrix multiplication
  - Can be done very easily with a GPU
  - This is a major reason why they are used
  - Parallel computation
- (1*vx + 0*vy + 0*vz + tx*1), (0*vx + 1*vy + 0*vz + tz*1), etc.

Memory

- Stride
  - The distance between elements in a single vector register
- Row Major Order versus Column Major Order
  - RMO is where elements in a row are stored next to each other in memory
    - C and C++
  - CMO is where column elements are next to each other
    - FORTRAN, R, MATLAB



source

Vector Architecture

- Data from across memory placed into large ‘vector registers,’ and operated on as a set by a single instruction (SIMD)
- Deep pipelining (hide memory latency)
  - Forwarding is called ‘chaining’

RV64V

- *Vector registers*
- *Vector functional units*
- *Vector load/store unit*
- *Scalar registers*

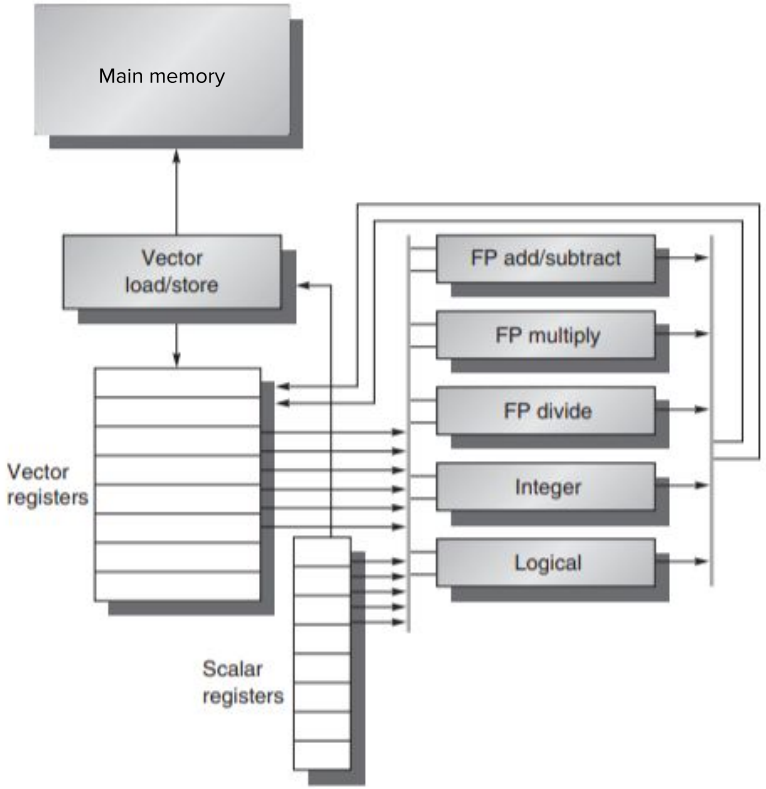


Figure 4.1 (RV64V)

Logical Similarities to Loop unrolling

(Example on p. 288)

```
        fld     f0,a            # Load scalar a
        addi    x28,x5,#256     # Last address to load
Loop:   fld     f1,0(x5)        # Load X[i]
        fmul.d  f1,f1,f0        # a × X[i]
        fld     f2,0(x6)        # Load Y[i]
        fadd.d  f2,f2,f1        # a × X[i] + Y[i]
        fsd     f2,0(x6)        # Store into Y[i]
        addi    x5,x5,#8        # Increment index to X
        addi    x6,x6,#8        # Increment index to Y
        bne     x28,x5,Loop     # Check if done
```

Here is the RV64V code for DAXPY:

```
        vsetdcfg  4*FP64        # Enable 4 DP FP vregs
        fld       f0,a          # Load scalar a
        vld       v0,x5         # Load vector X
        vmul      v1,v0,f0      # Vector-scalar mult
        vld       v2,x6         # Load vector Y
        vadd      v3,v1,v2      # Vector-vector add
        vst       v3,x6         # Store the sum
        vdisable                # Disable vector regs
```

Performance Pitfalls - Branches

- Sparse matrices
- Branches
    - Control dependencies in a loop

```
for (i = 0; i < 64; i++)
    if (X[i] != 0)
        X[i] = X[i] - Y[i]
```

- Branch solutions:
    - Predicate registers
    - Vector-mask control

Benefits of Vector Architecture

- Uses data level parallelism
  - Can apply a single instruction to an entire data set (vector) all in parallel
- Most latency only happens once per vector load rather than once per element
  - O(1) vs O(n)
  - This is possible because of pipelining
- Simple in terms of energy consumption and design
  - Compared to something like superscalar processors

- **BUT....** Without sufficient memory bandwidth, vector execution does not increase performance

High Level: Static vs Dynamic Typing

- A statically typed language is one that has information about variables types at compile time
  - `int i = 1;`
  - Examples like this indicate to the compiler what type i is
  - Languages like Java, C, C++, etc

- A dynamically typed language is one whose data types are determined during run time
  - `var j = 3.141;`
  - The compiler will not determine the type of the variable until it is referenced or accessed
  - Languages like Python, Javascript, Ruby

Dynamic Register Typing

- Each vector register has information associated with it
  - Size of the data and what type
  - Programs dynamically adjust registers before execution
- Normally, the instruction would provide information about the data
  - Instead, this allows for more variety and simplicity in instructions
- No redundant instructions now
  - Usually, you would have several instructions for the same operation, but with different types
  - Now, one operation for all types
  - No ‘load double’ instructions

Parallel Pipelines in Vector Processors

- Called ‘Lanes’
  - Parallel pipelines in functional units, allowing 2+ results per clock cycle
- *Convoy* - a set of vector instructions that can execute together, as they don’t contain structural hazards
  - Chaining removes read-after-write hazards in a convoy
- RV64V: all vector instructions only allow element N of one v. reg. To take part in operations with element N of other v. regs

Lanes

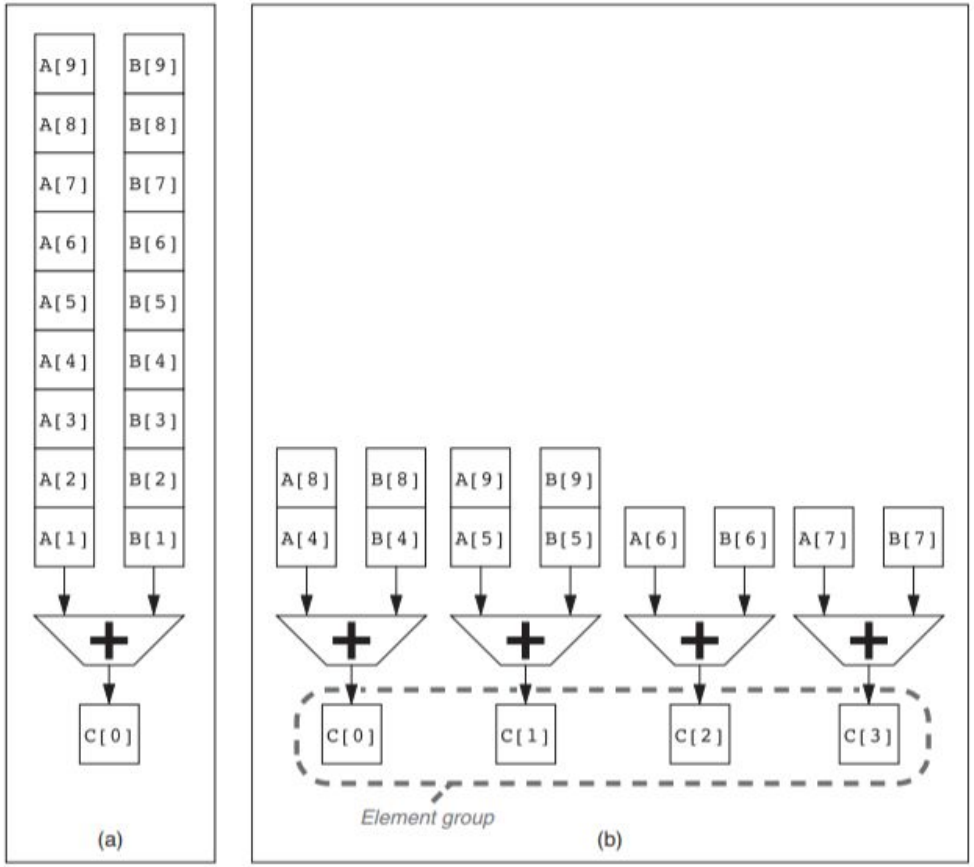


Figure 4.4

Dependencies

- Analyze loop dependencies
- 2 dependencies in example
  - S2 is dependent on the result of S1
    - Still parallelizable with hardware
  - Each iteration relies on the result of the iteration before (S1 relies of S1)
    - Can’t parallelize with this dependency
  - Cannot vectorize everything

```
for (i=0; i<100; i=i+1) {
      A[i+1] = A[i] + C[i];   /* S1 */
      B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

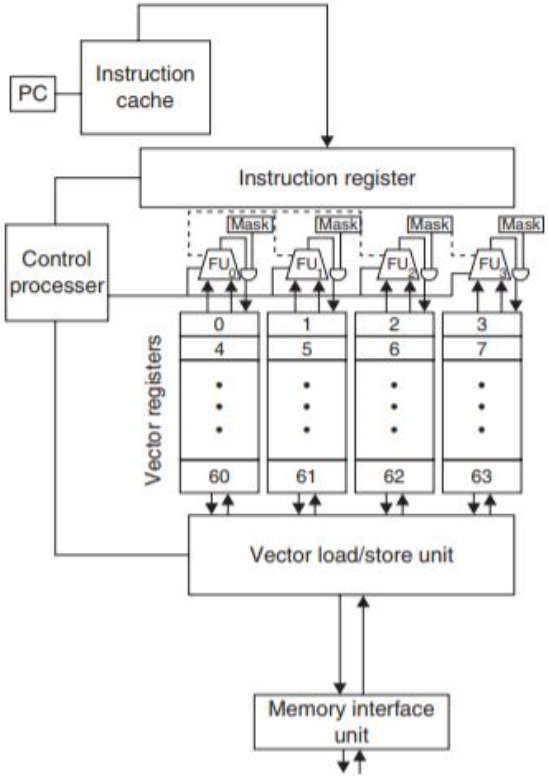
Questions?

Moving from Vector to GPU Architecture

- GPUs transfer data, deal with conditionals, the same as vector registers
- All GPU loads are gather instructions and all GPU stores are scatter instructions
- GPUs additionally add multithreading within each SIMD processor, to hide memory latency
  - Have additional registers to support multithreading
  - Via a scoreboard
  - Two levels of scheduling
- No Control Processor

GPU Architecture

Figure 4.22



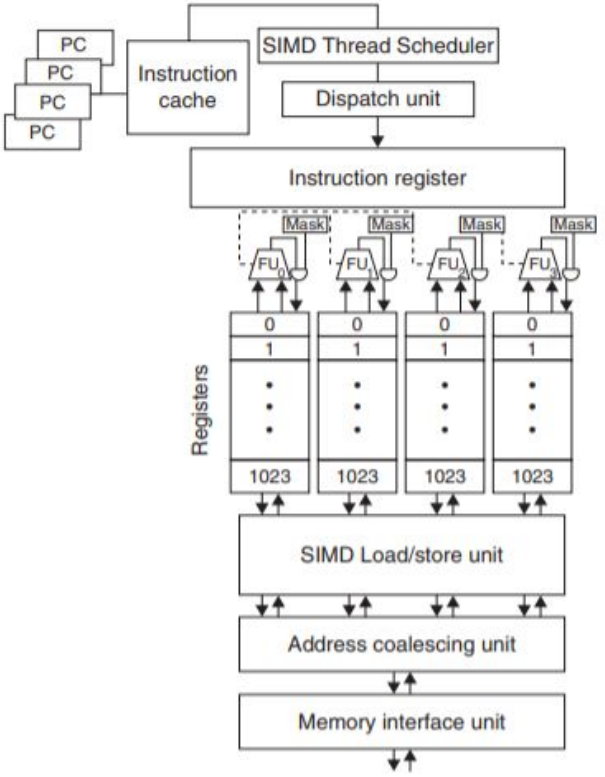
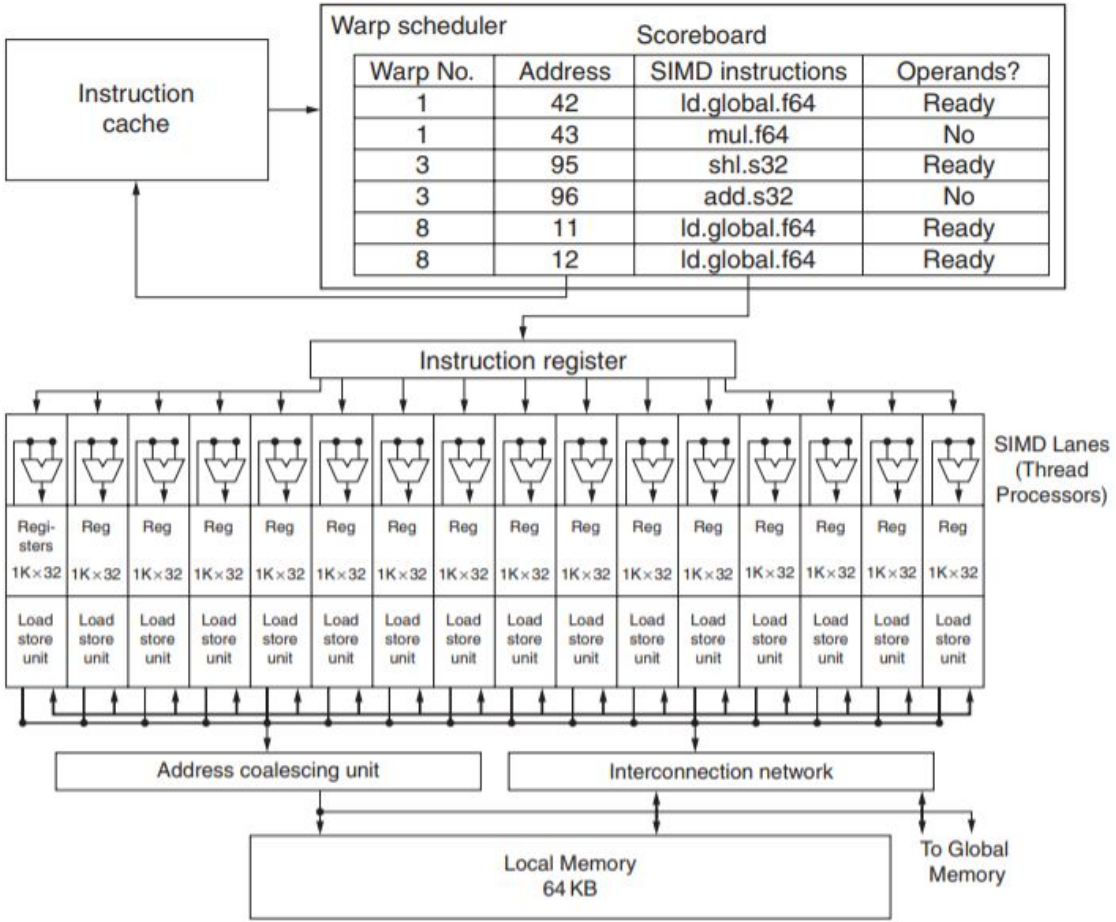


Figure 4.14, a

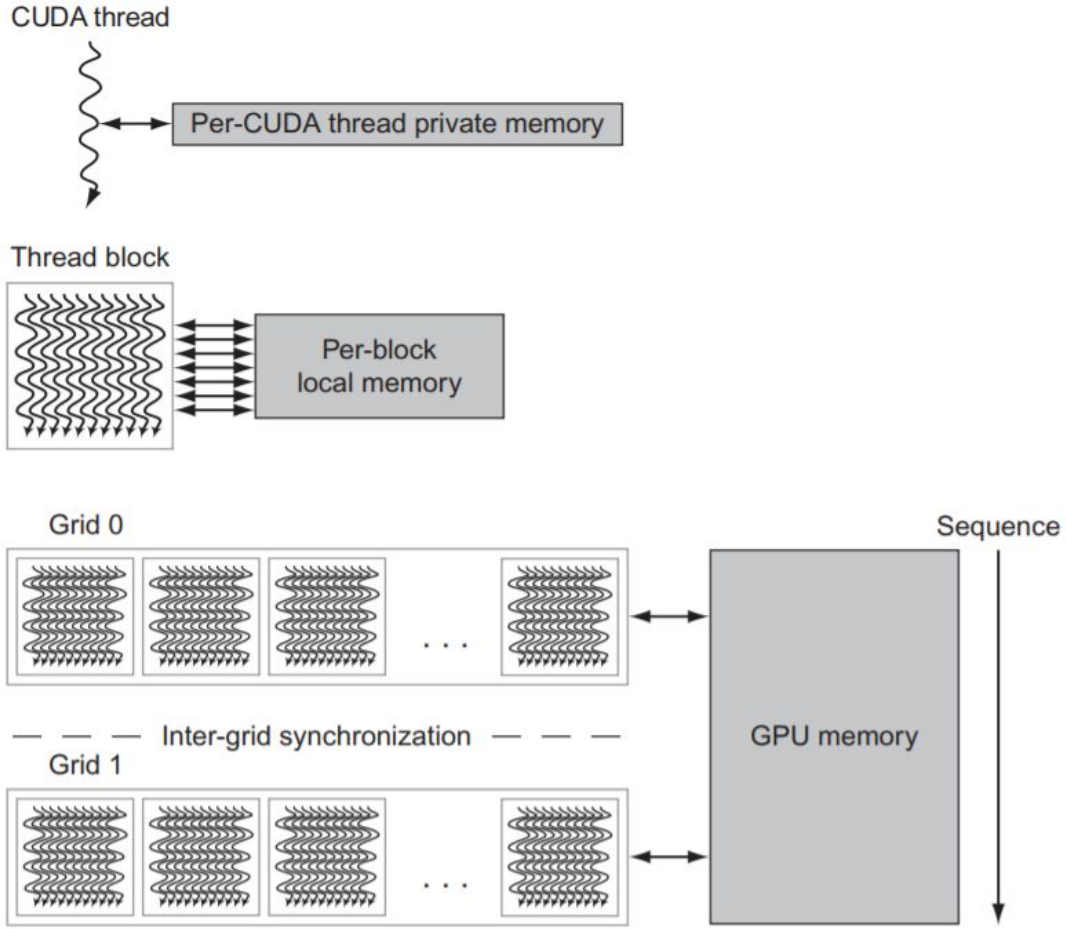
Multithreaded SIMD

processor



GPU Memory

- Gather-Scatter
- Figure 4.18, GPU memory structures



NVIDIA GeForce GTX 1080 Ti Founders Edition

- GTX is most common consumer graphics card brand
- Released March 2017
- Launched at $699
- Pascal Architecture
- GP102 chip

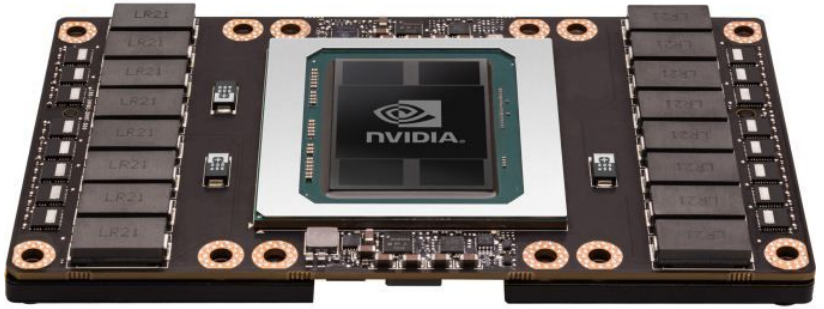
	GTX 1080 Ti
CUDA Cores	3584
Texture Units	224
ROPs	88
Core Clock	1481MHz
Boost Clock	1582MHz
TFLOPs (FMA)	11.3 TFLOPs
Memory Clock	11Gbps GDDR5X
Memory Bus Width	352-bit
VRAM	11GB
FP64	1/32
FP16 (Native)	1/64
INT8	4:1
TDP	250W
GPU	GP102
Transistor Count	12B
Die Size	471mm2
Manufacturing Process	TSMC 16nm

GTX 1080 Ti: GP102 Chip

GTX 1080 Ti: Pascal Architecture

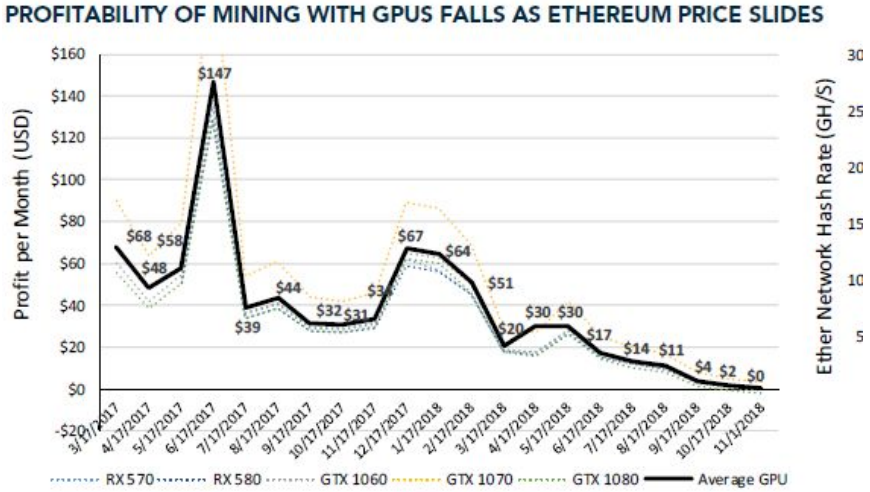
- All Chips
  - Better floating point performance
  - Unified virtual memory between GPUs and CPUs
- GP102 Chip
  - Support for INT8 Dot Product operations
  - Faster INT8 operations
- GP100 Chip
  - Fast HBM2 memory
  - NVLink for faster communication between discrete GPUs

GP102 chip



GP100 Chip

Current Uses



- Graphics processing!!!
- Cryptocurrency mining [1]
  - Decreasing in popularity
- Asymmetric cryptography
  - Cryptographic accelerator cards are expensive [2]
- AI, machine learning, deep learning
  - Tesla's self-driving cars (now use custom application-specific integrated circuit (ASIC)) [3]
- Data science
  - RAPIDS libraries built on NVIDIA CUDA-X AI [4]

Graph source:
https://www.cnbc.com/2018/11/13/this-chart-shows-how-cryptocurrency-mining-on-your-own-is-no-longer-profitable.html

Questions?

References

Textbook: Computer Architecture: A Quantitative Approach, 6th Edition, John Hennessy & David Patterson

GPU Architecture:

[1] https://www.quora.com/What-is-the-difference-between-CPU-cores-and-GPU-cores-in-terms-of-computation

[2] https://iq.opengenus.org/key-ideas-that-makes-graphics-processing-unit-gpu-so-fast/

[3] Appendix F - OpenGL Programming Guide, glprogramming.com/red/appendixf.html#name1.

GPU Example

[1] https://www.anandtech.com/show/11180/the-nvidia-geforce-gtx-1080-ti-review

[2] https://www.anandtech.com/show/10222/nvidia-announces-tesla-p100-accelerator-pascal-power-for-hpc

[3] https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html

[4] https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

Current Uses:

[1] https://www.investopedia.com/tech/gpu-cryptocurrency-mining/

[2] https://www.iacr.org/archive/ches2008/51540081/51540081.pdf

[3] https://www.forbes.com/sites/jeanbaptiste/2018/08/15/why-tesla-dropped-nvidias-ai-platform-for-self-driving-cars-and-built-its-own/#212fcba76722

[4] https://www.nvidia.com/en-us/deep-learning-ai/solutions/data-science/