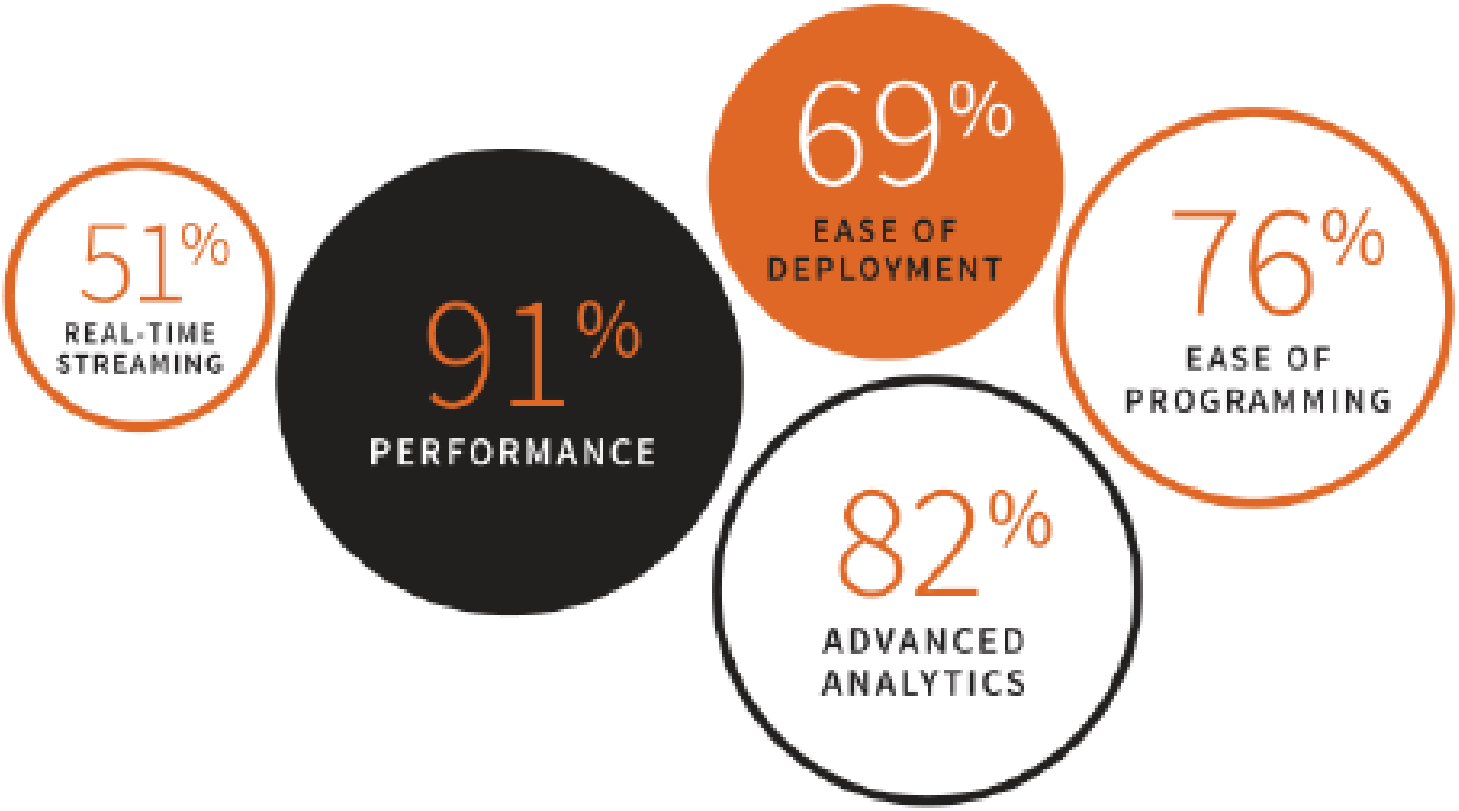
EE542

Lecture 11: Hardware Accelerators in Cloud

Internet and Cloud Computing

Young Cho
Department of Electrical Engineering
University of Southern California

Priority for Cloud Computer Users



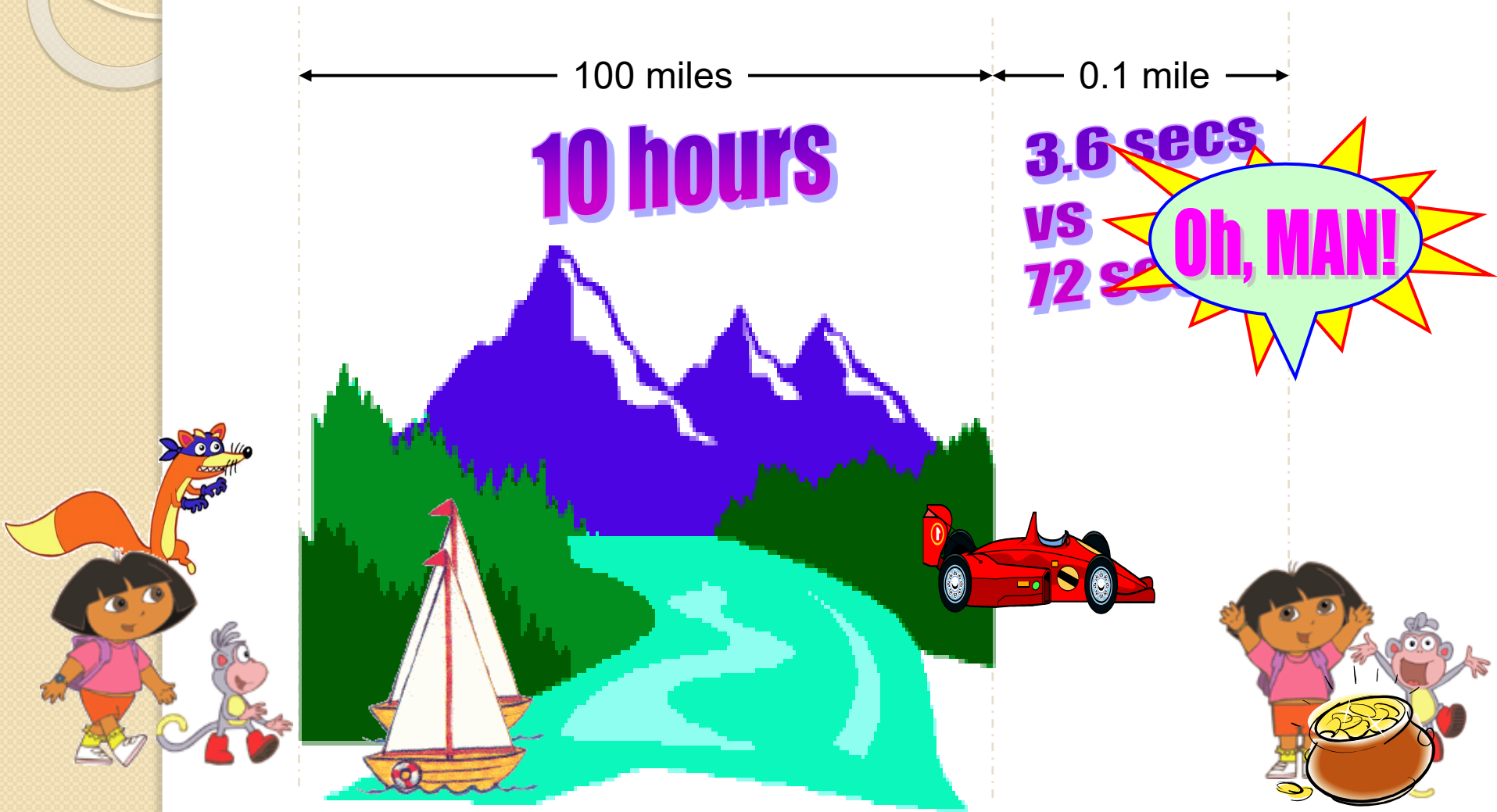
Overall Performance

- Trade-off
  - Speed vs Power vs Area vs Cost
- HW Accelerations
  - NSP (Native Signal Processing)
  - GPU (Graphics Processing Units)
  - FPGA (Field Programmable Gate Arrays)

True Performance Measurement

Sail Boat Speed: 10 miles/hour

Race Car: 100 miles/hour
Running: 5 miles/hour



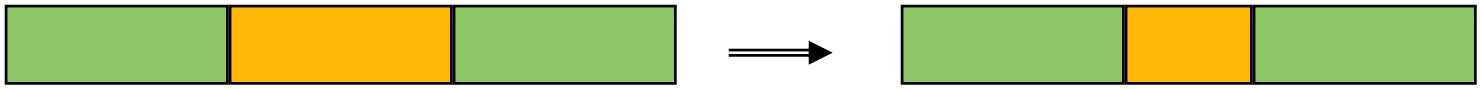
Amdahl's Law

$$ExTime_{new} = ExTime_{old} \times \left[\left(1 - Fraction_{enhanced}\right) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}} \right]$$

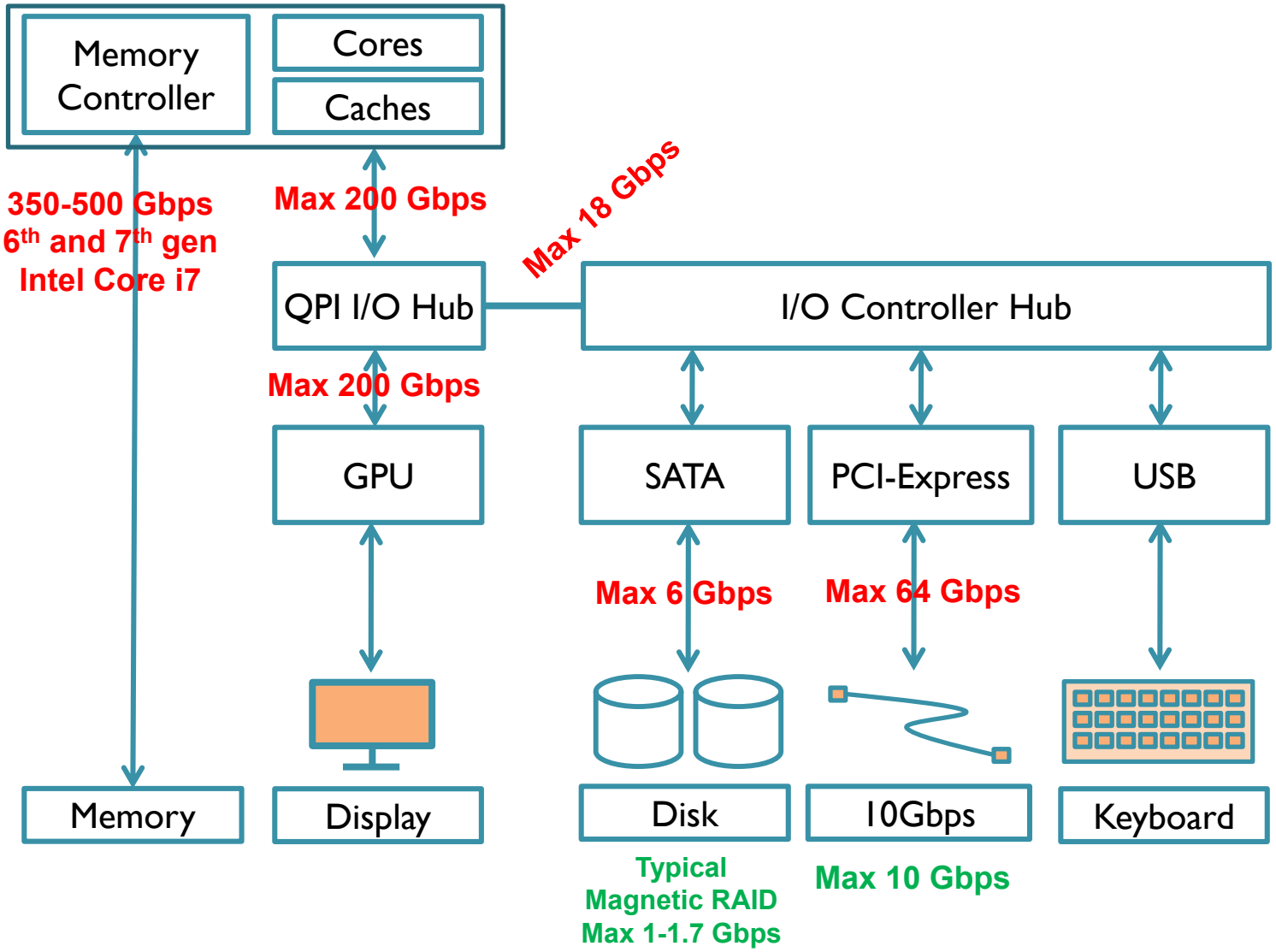
$$Speedup_{overall} = \frac{ExTime_{old}}{ExTime_{new}} = \frac{1}{\left(1 - Fraction_{enhanced}\right) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

Best you could ever hope to do:

$$Speedup_{maximum} = \frac{1}{\left(1 - Fraction_{enhanced}\right)}$$



Bottleneck before Network



Example Solutions

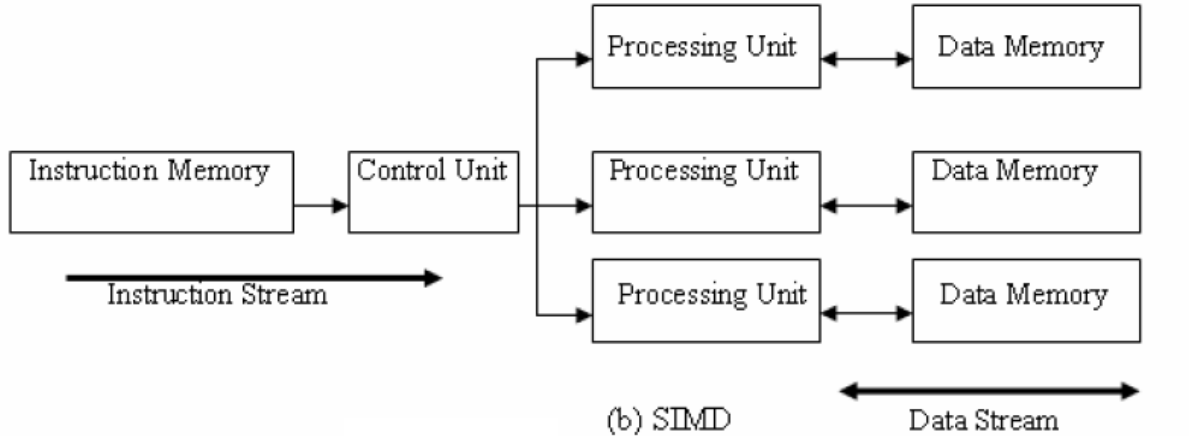
- Intel Optane SSD 905P
  - ~$2,000
  - 1.5TB
  - One of the first PCIe SSDs
  - Non-Volatile Memory Express
  - ~18Gbps Reads/~14Gbps Writes
  - PCIe 3.0: ~30Gbps
- Western Digital Black SSD
  - ~$220
  - 1TB
  - Expansion card
  - PCIe 3.0 x8: ~64Gbps
  - ~30Gbps Reads/~13Gbps Writes
- Crucial PCIe 4.0 M.2 SSD
  - ~$120
  - 1TB
  - ~50Gbps Reads/~40Gbps Writes

Native Signal Processing Extensions

- HW Accelerated Special Instructions
  - Most General Purpose Processors
- Intel
  - MMX (MultiMedia eXtensions)
  - SSE (Streaming SIMD Extensions)
  - SSE2, SSE3, Supplemental SSE3, SSE4
  - AVX (Advanced Vector eXtensions)
- AMD
  - 3DNow!
- PowerPC
  - AltiVec

SIMD architectures

- A data parallel architecture
- Applying the same instruction to many data
  - Save control logic
  - A related architecture is the vector architecture
  - SIMD and vector architectures offer high performance for vector operations.



Vector operations

- Vector addition Z = X + Y

  for (i=0; i<n; i++) z[i] = x[i] + y[i];

$$\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ \dots\dots \\ x_n + y_n \end{pmatrix}$$

- Vector scaling Y = a * X

  for(i=0; i<n; i++) y[i] = a*x[i];

$$a * \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} = \begin{pmatrix} a * x_1 \\ a * x_2 \\ \dots\dots \\ a * x_n \end{pmatrix}$$

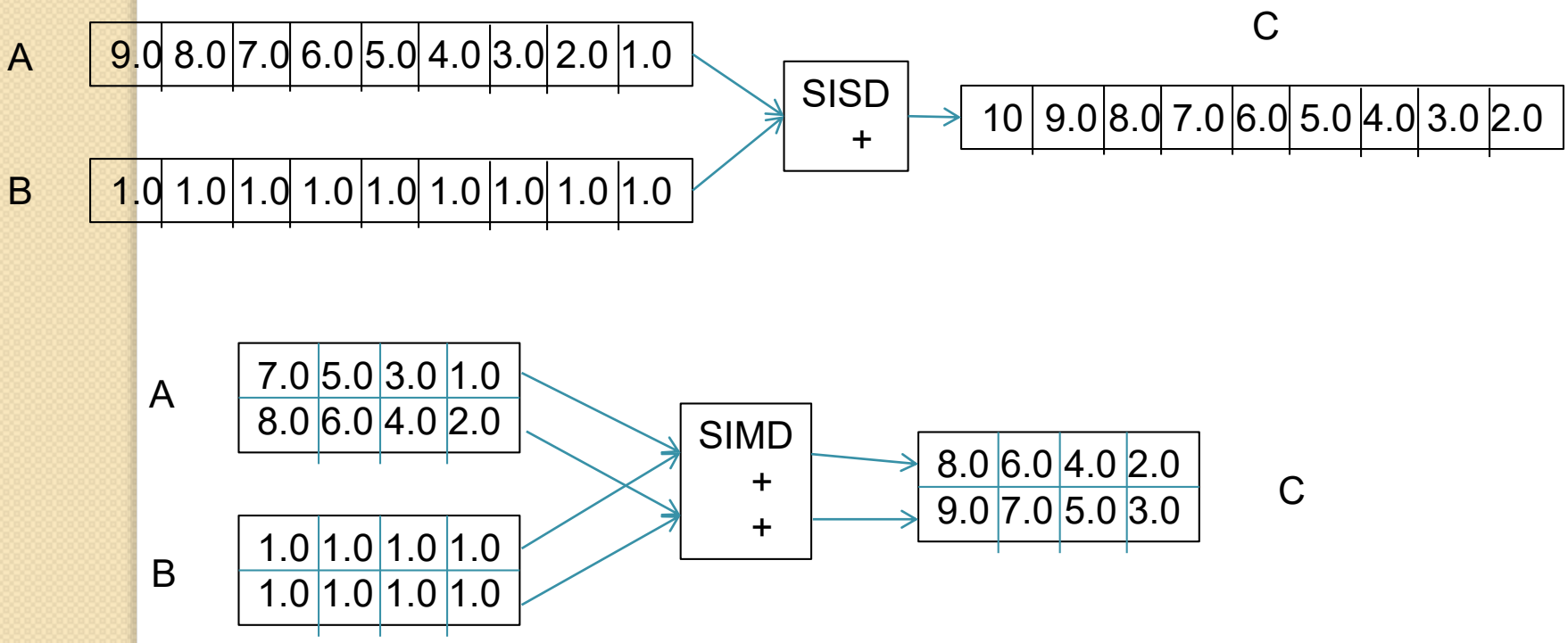
- Dot product

  for(i=0; i<n; i++) r += x[i]*y[i];

$$\begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} \bullet \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix} = x_1 * y_1 + x_2 * y_2 + \dots\dots + x_n * y_n$$

SISD and SIMD vector operations

- C = A + B
  - For (i=0;i<n; i++) c[i] = a[i] + b[i]



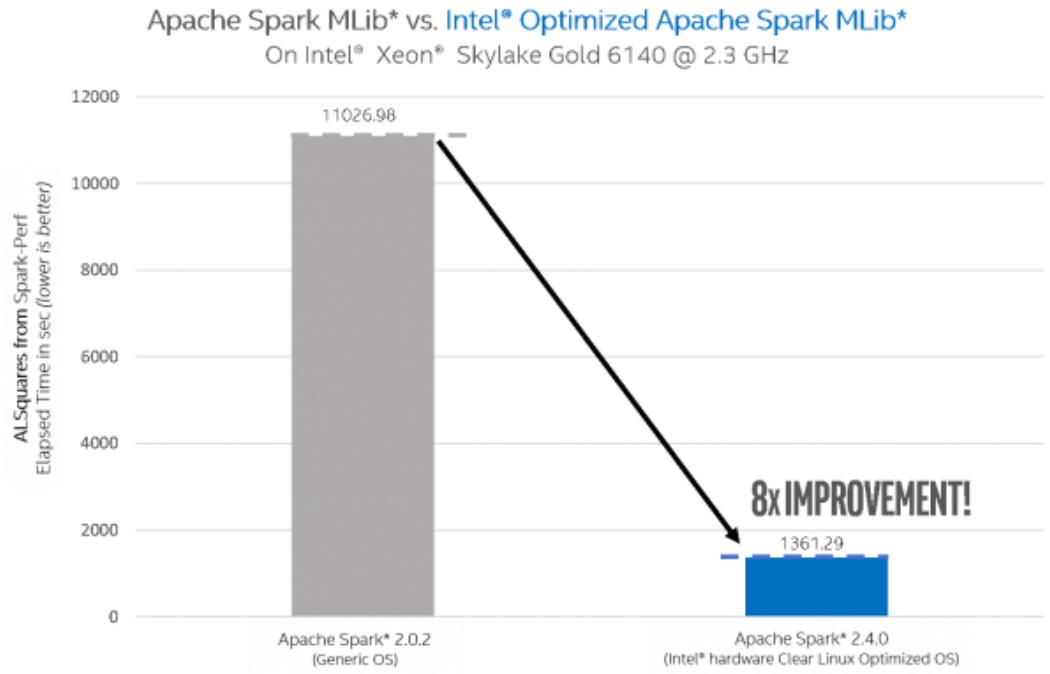
Normal vs. SSE C Code

```
// Scalar
inline void
normalize(std::vector<vec3>& data)
{
  for (int i=0;i<data.size();i++)
  {
    vec3& m = data[i];
    m /= sqrtf( m[0]*m[0] + m[1]*m[1]
+ m[2]*m[2] );
  }
}
```

```
// SSE
inline void normalize(std::vector<mat4x3>&
data)
{
  for (int i=0;i<data.size();i++)
  {
    vec4 mx = data[i][0];
    vec4 my = data[i][1];
    vec4 mz = data[i][2];

    vec4 multiplier = 1.0f/Sqrt(mx*mx + my*my
+ mz*mz);

    mx*=multiplier;
    my*=multiplier;
    mz*=multiplier;

    data[i][0]=mx;
    data[i][1]=my;
    data[i][2]=mz;
  }
}
```

NSP Acceleration on Cloud

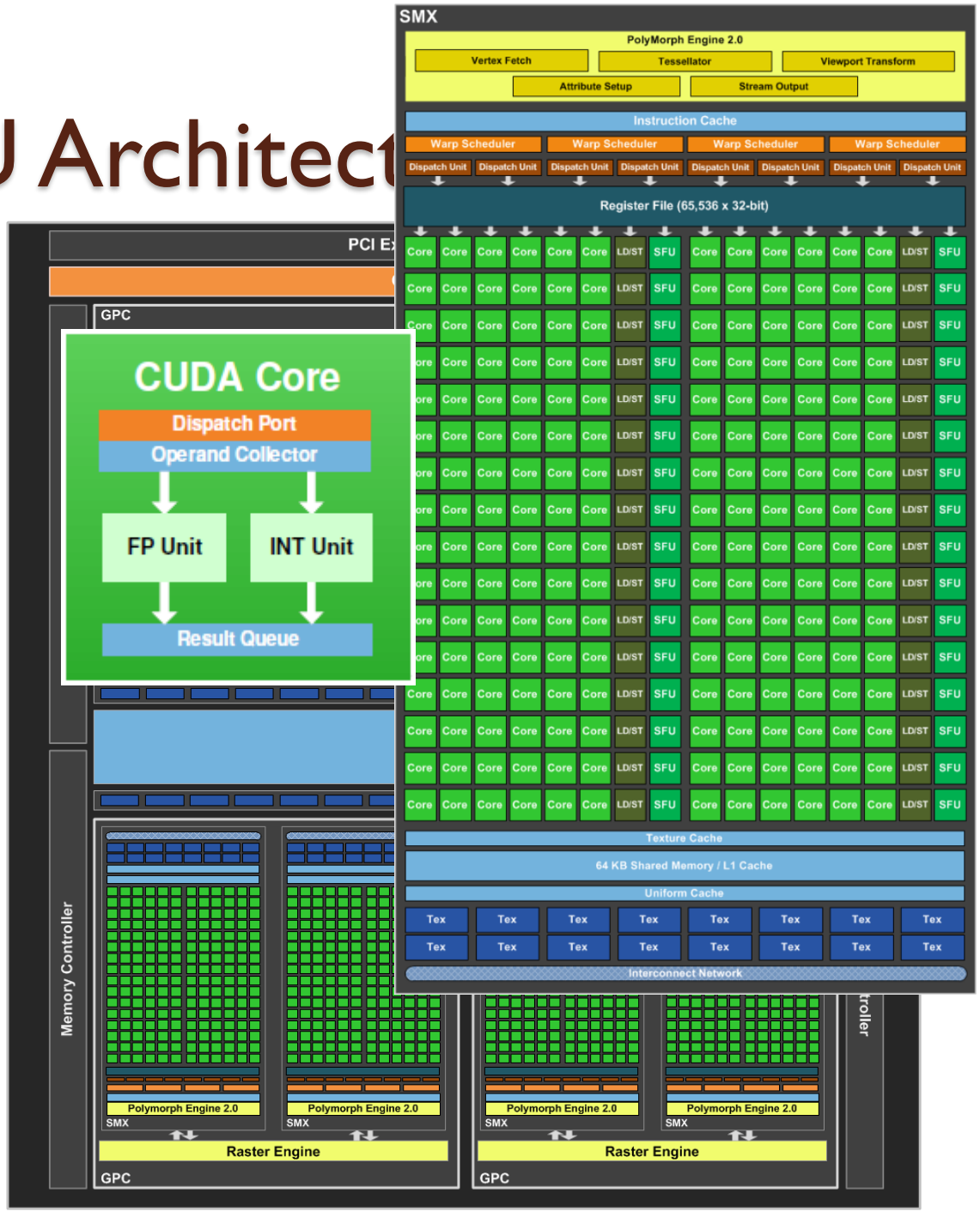
- Integration of NSP tuned OS
  - Clear Linux OS for Intel Architecture
- Processor Specific Tuning
  - Intel AVX512 (Intel AVX-512)
  - Intel Memory Protection Extensions



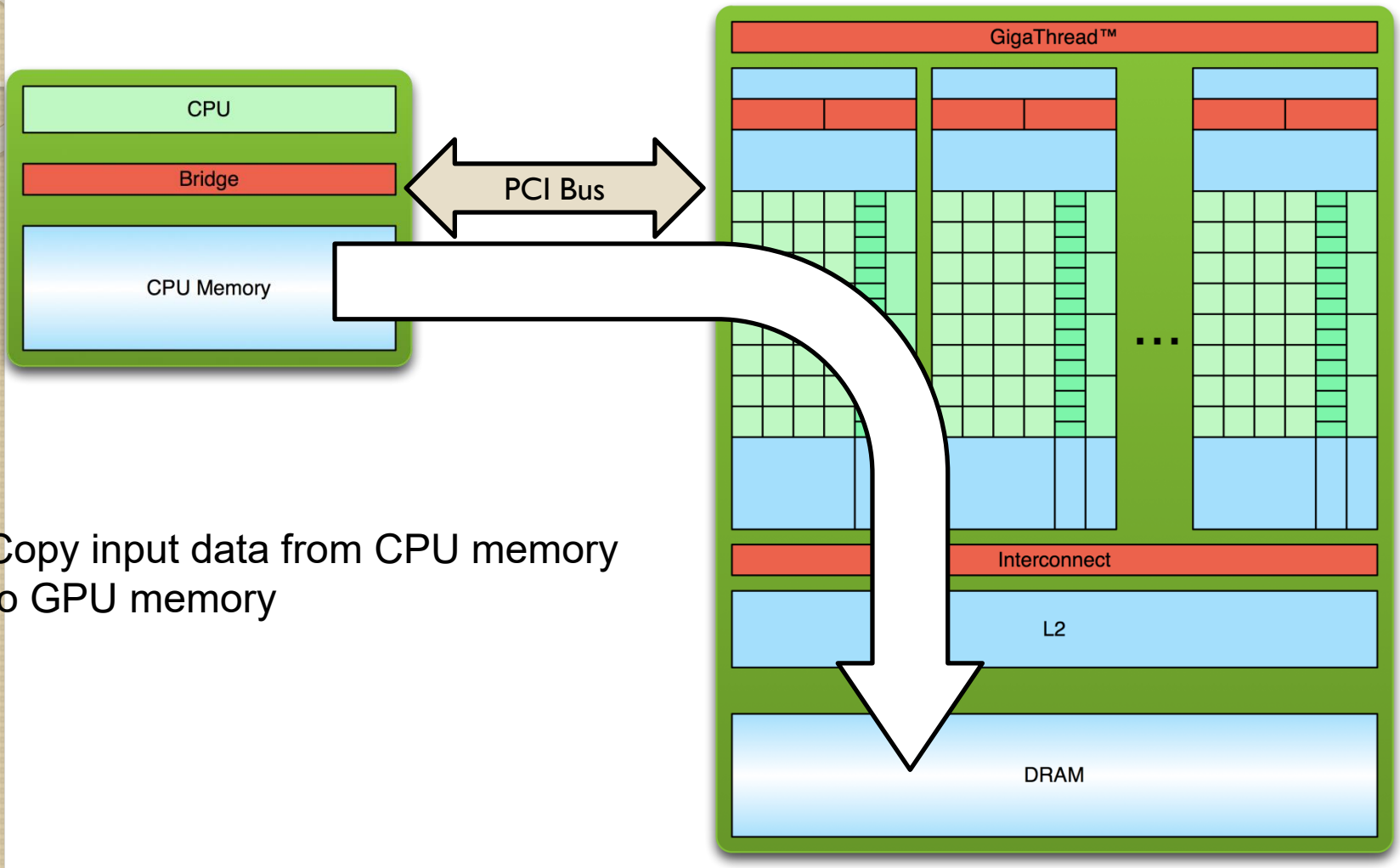
Graphics Processing Unit

- NVidia Architecture
  - Tesla – 960 cores, 1.4 Ghz, 1200W, yr 2008
  - Fermi – 1792 cores, 1.15 Ghz, 900W, yr 2011
  - Kepler – 4992 cores, 560 Mhz, 300W, yr 2014
  - Maxwell – 4096 cores, 899 Mhz, 300W, yr 2015
  - Pascal – 3840 cores, 1.3 Ghz, 250W, yr 2016
  - Volta – 5120 cores, ?? Ghz, 300W, yr 2017
  - Turing – 2560 cores, ?? Ghz, 70W, yr 2018
- AMD Architecture
  - TeraScale 1,2,3 – 3300 cores, 830 Mhz, 375W, yr. 2011
  - GCN gen 1,2,3 – 8600 cores, 1 Ghz, 350W, yr 2016
  - GCN gen 4 – 5000 cores, 1.34 Ghz, 185W, yr 2017
  - GCN gen 5 – 4800 cores, 1.68 Ghz, 345W, yr 2017

GPU Architect

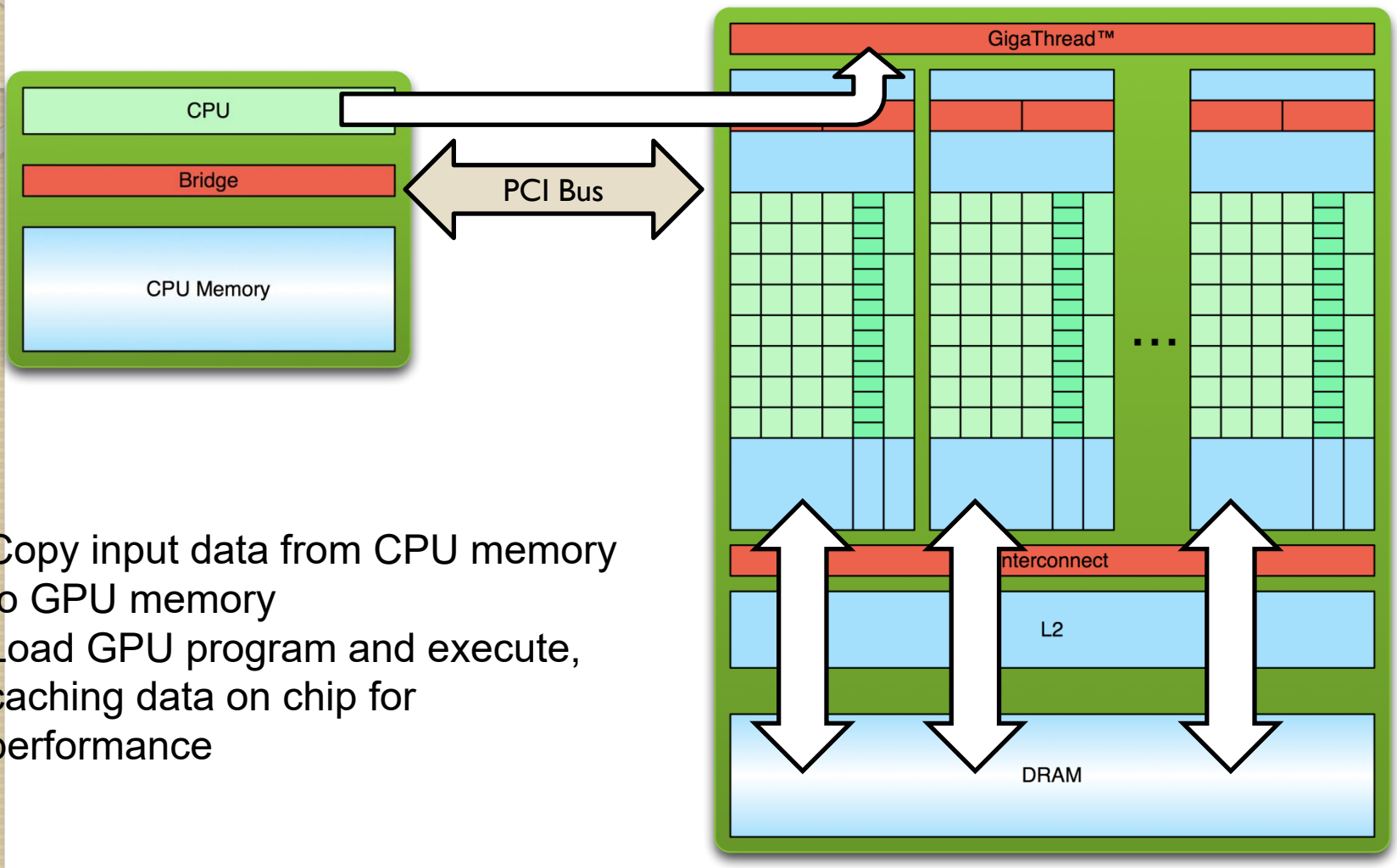


Simple Processing Flow



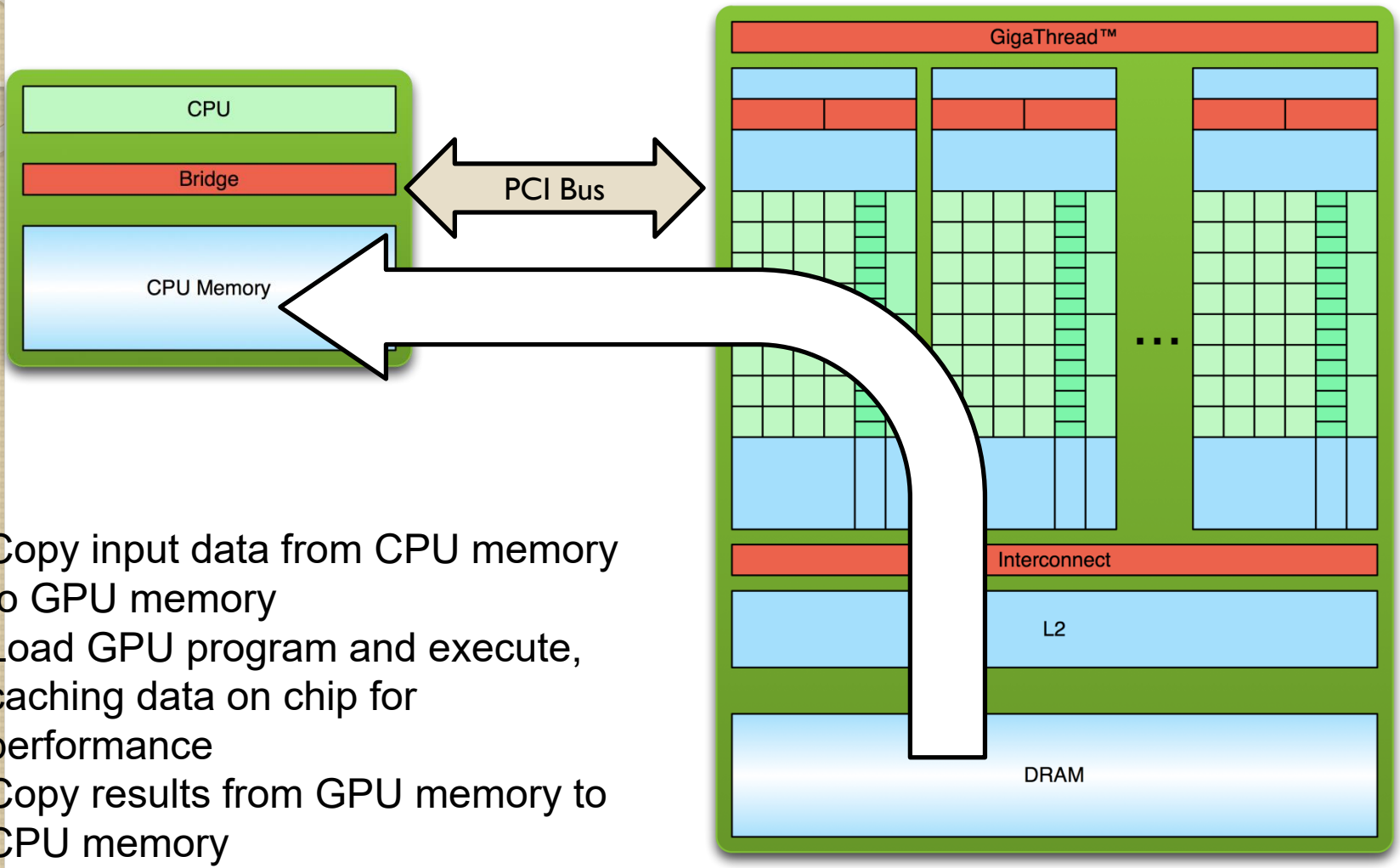
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

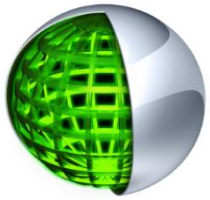
CUDA Parallel Computing Platform

www.nvidia.com/getcuda

Programming Approaches

- **Libraries** — "Drop-in" Acceleration
- **OpenACC Directives** — Easily Accelerate Apps
- **Programming Languages** — Maximum Flexibility

Development Environment



Nsight IDE
Linux, Mac and Windows
GPU Debugging and Profiling

CUDA-GDB debugger
NVIDIA Visual Profiler

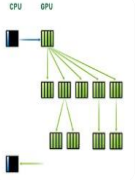
Open Compiler Tool Chain

Enables compiling new languages to CUDA platform, and CUDA languages to other architectures

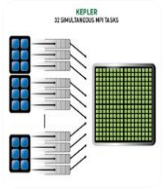
Hardware Capabilities

SMX

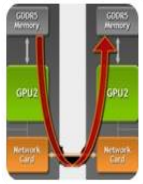
Dynamic Parallelism



HyperQ



GPUDirect



Hello World!

```
int main(void) {
        printf("Hello World!\n");
        return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc hello_world.cu
$ a.out
Hello World!
$
```

Addition on the Device: `add()`

- Returning to our `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- Let's take a look at main()…

Addition on the Device: main()

```
int main(void) {
        int a, b, c;                    // host copies of a, b, c
        int *d_a, *d_b, *d_c;           // device copies of a, b, c
        int size = sizeof(int);

        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Setup input values
        a = 2;
        b = 7;
```

Spark and CUDA

- Hand-tuned GPU program in CUDA

```
_global_ void yourGPUKernal(double *in, double *out, long size) {
  long i = threadIdx.x + blockIdx.x * blockDim.x;
  out[i] = in[i] * PI; }
```

```
val mapFunction = new CUDAFunction(…, "yourGPUKernel.ptx")
val output = data.mapExtFunc(…, mapFunction)
```

- Spark program with automatic translation to GPU code

```
val output = data.map(p => Point(p.x * 2, p.y * 2))
```

GPU Execution

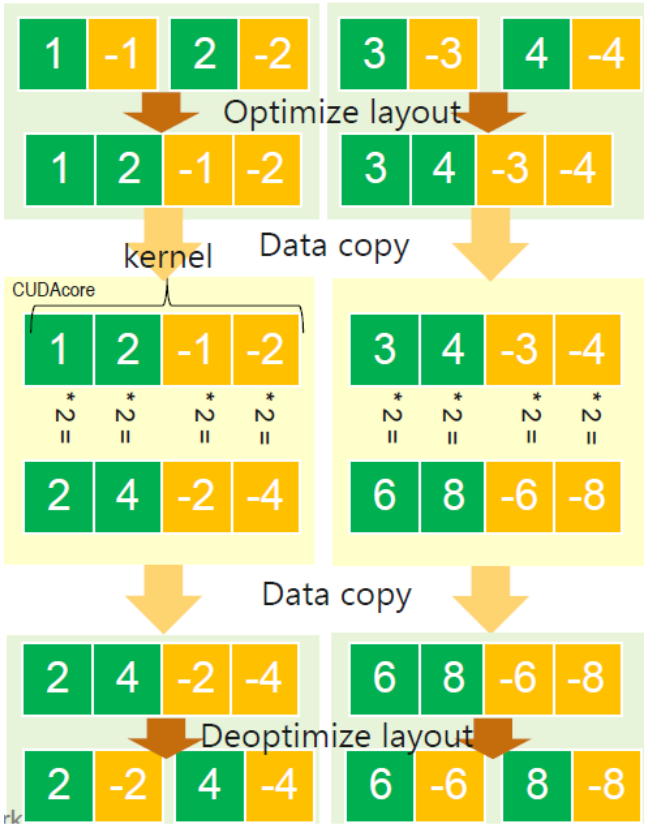
- Prep Data

```
.mapExtFunc(
    p => Point(p.x*2, p.y*2),
    mapFunction)
```

- CUDA code

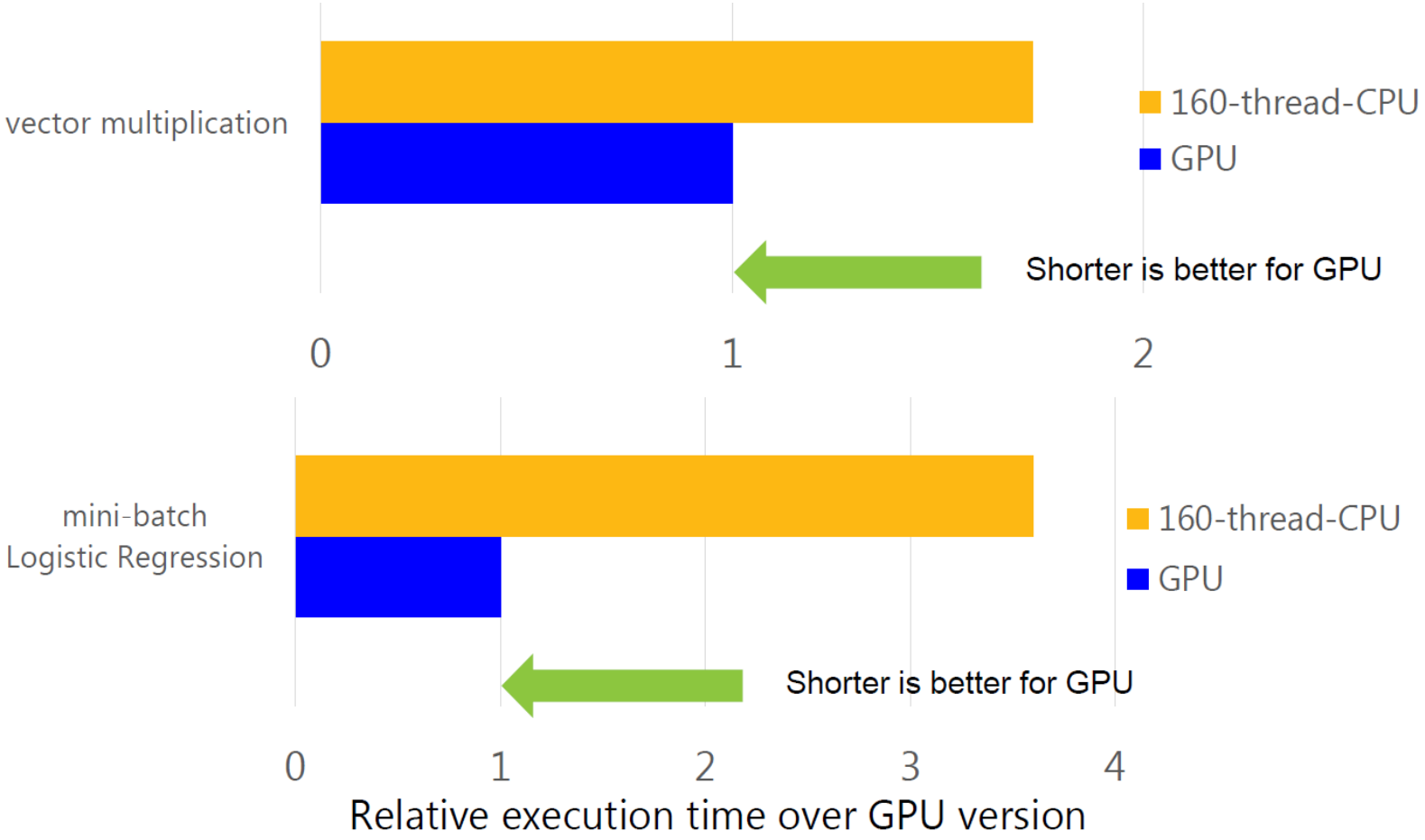
```
__global__ void multiplyBy2(…){
    outx[i] = inx[i] * 2;
    outy[i] = iny[i] * 2;
}
```

- Run on GPU



Spark on GPU Performance

- 2-3x for Tesla over Power8 160 SMT



An Interesting Side Note

- Clear Example of Unsolved IC Power Problem
- NVidia GTX 1070 (Advertised as a lower power than GTX1080)
  - Pasal GP104 GPU - exactly the same GPU as GTX1080
  - Total of 20 Streaming Multiprocessors (SM) Clusters
  - 15 out of 20 SM Clusters are enabled
  - Why 5 SMs disabled?? (20 SMs are enabled in GTX 1080)
- Our Research Approach
  - Benchmark SW for each SM
  - If command to disable SMs based on ID
  - By disabling one SM at a time, we ranked efficiencies
- Research Conclusion
  - 4 best SMs consume the same amount of power as 3 worst SMs
  - Average of 33% more power needed by bad processors!!

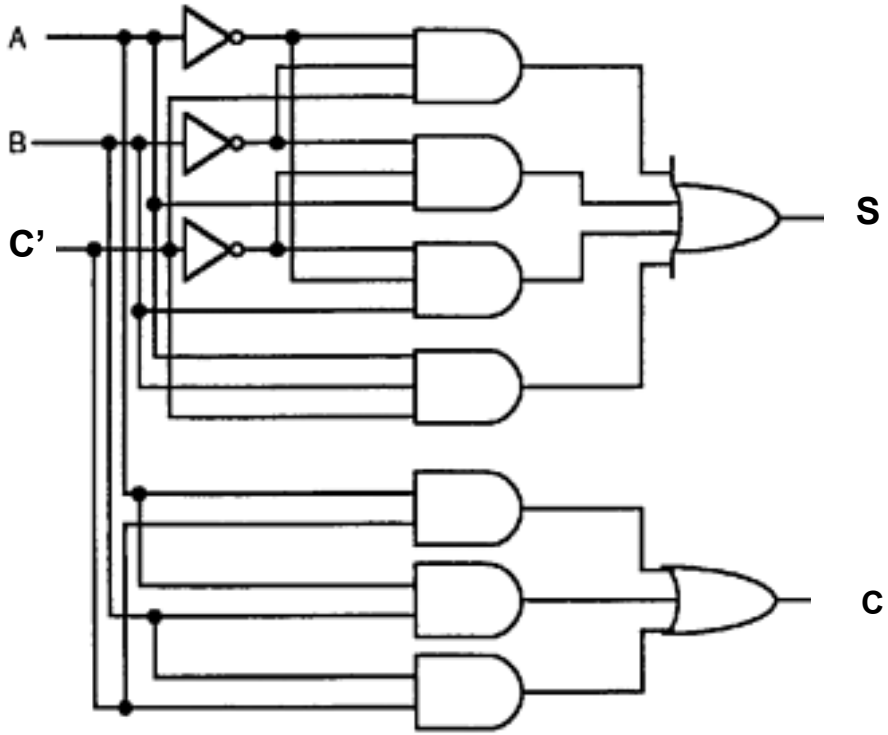
Field Programmable Gate Arrays

- Great for Prototyping and Testing
  - Enable logic verification without high cost of fab
  - Reprogrammable → Research and Education
  - Meets most computational requirements
  - Options for transferring design to ASIC
- Technology Advances
  - Huge FPGAs are available
    - Up to 200,000 Logic Units
  - Above clocking rate of 500 MHz
- Competitive Pricing

Designing with FPGAs

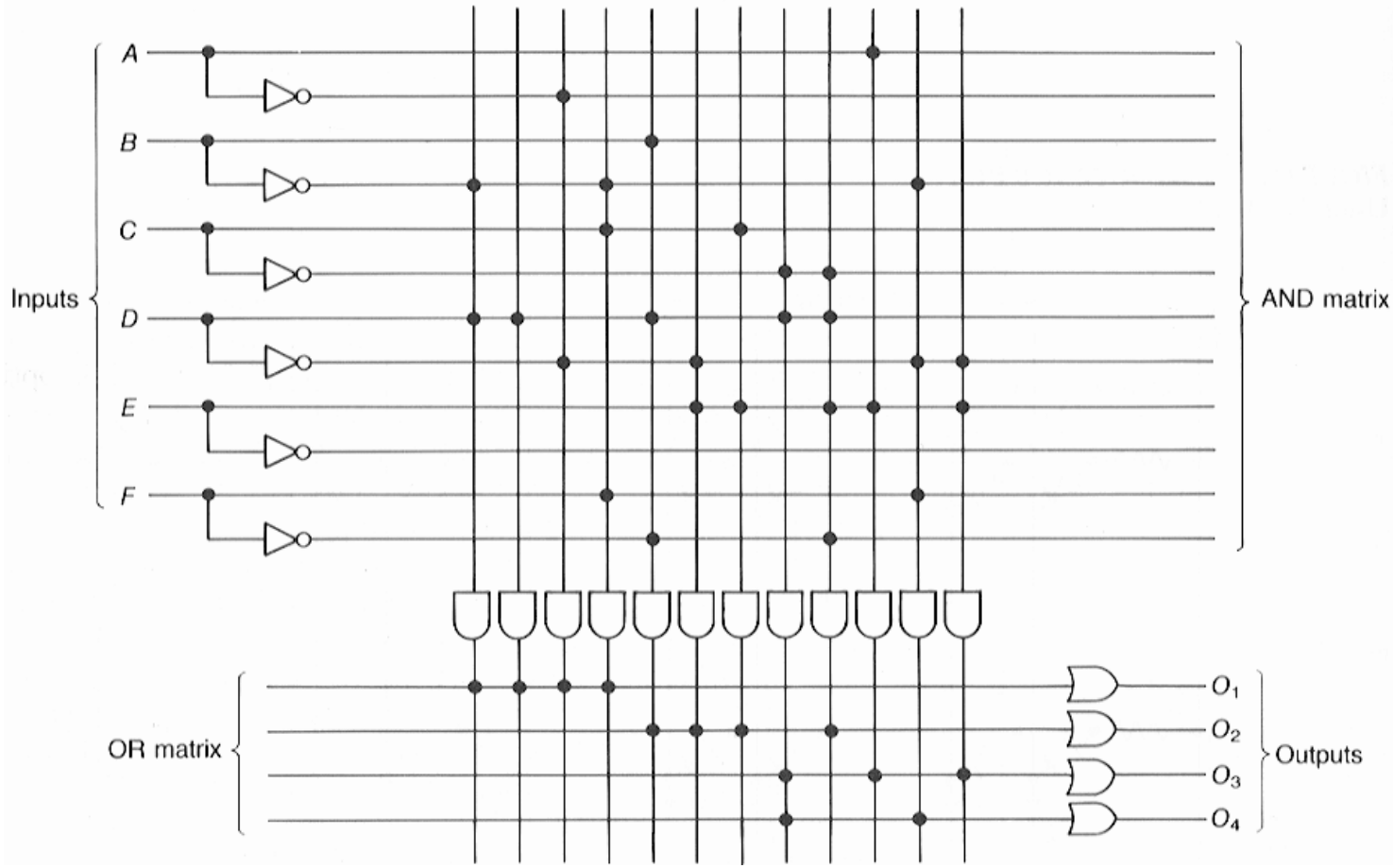
- Opportunities
  - Hardware logics are programmable
  - Immediate testing on the actual platform
- Challenges
  - Programming Environment
    - Think and design in 2-D instead of 1-D
    - Consider hardware limitations
  - Hardware Synthesis
    - Smart language interpreter and translator
    - Efficient HW resource utilization

Full-Adder Using Array of Logics

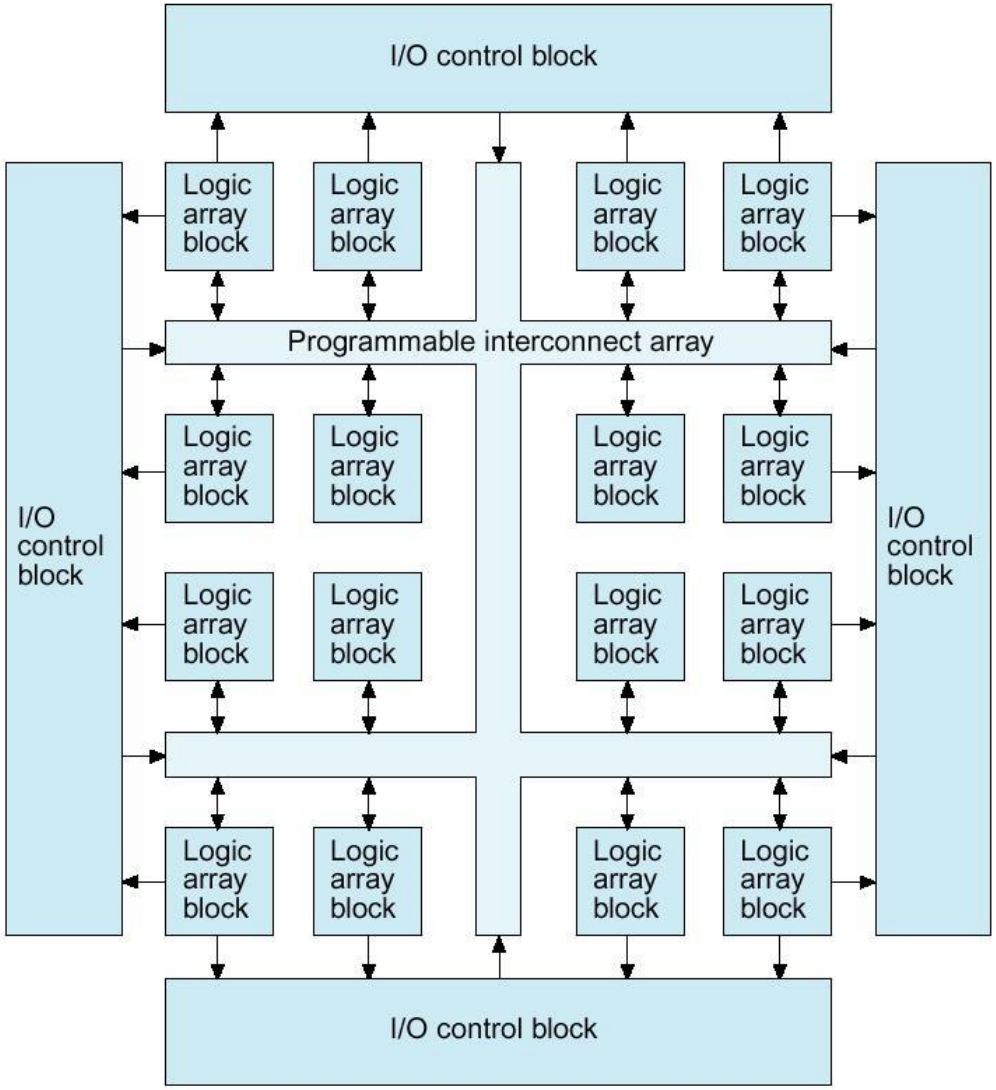


Input			Output	
C'	A	B	S	C
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

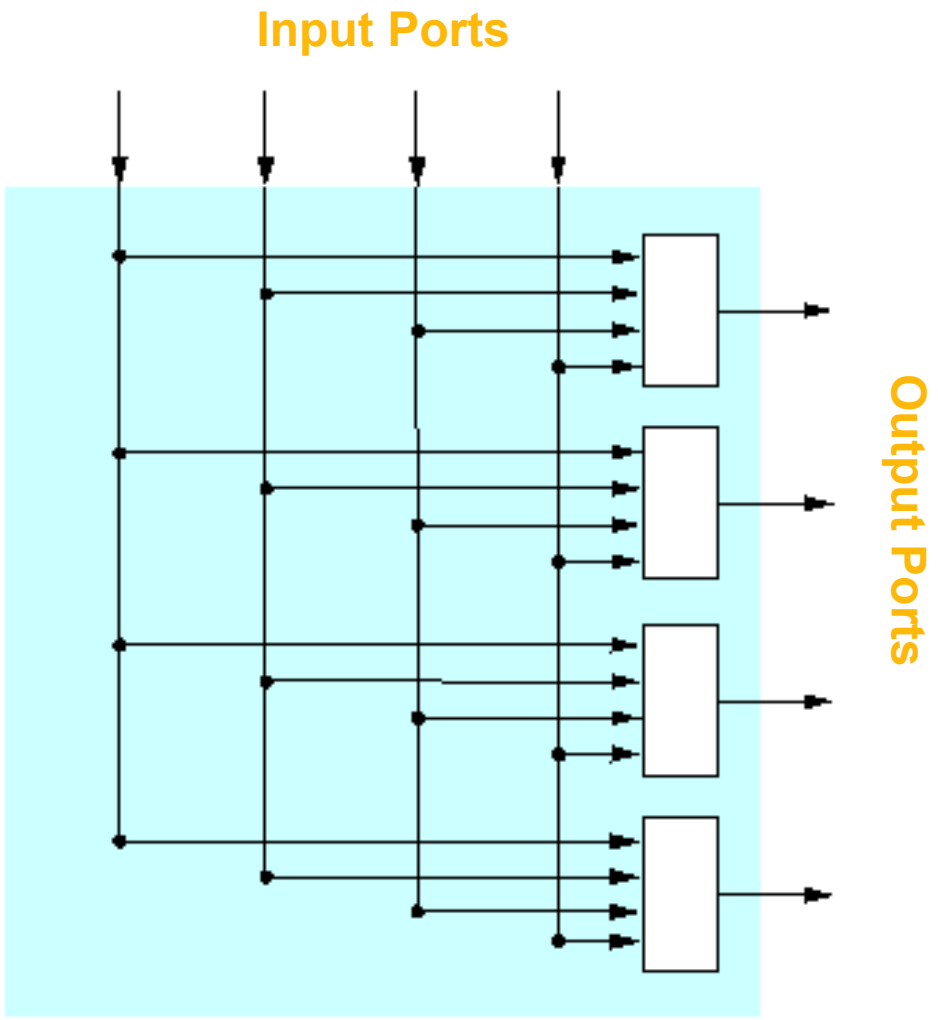
Programmable Logic (PLA/PAL/PLD)



More Complex Programmable Logic



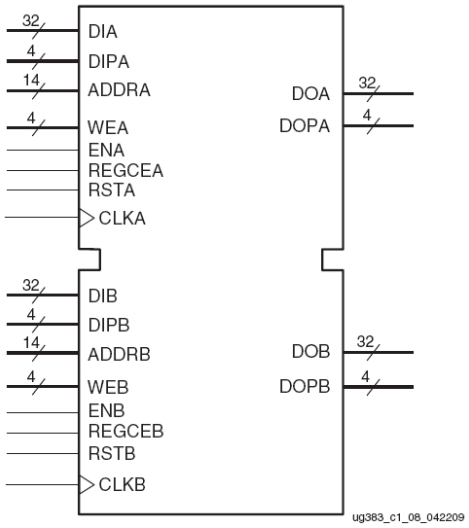
Simple Wire Switch (4x4 Crossbar)



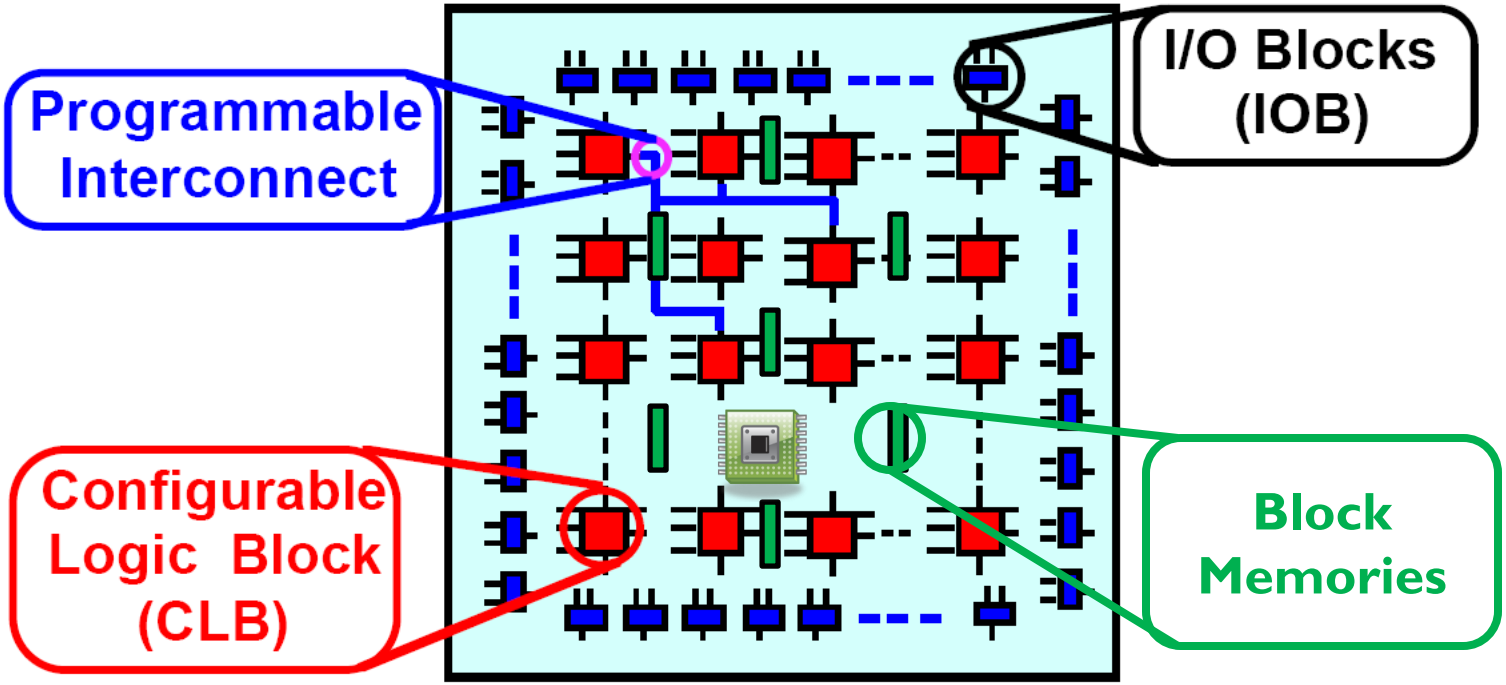
Block Memory (18 Kbit)

		Port A								
		No Parity Bits						With Parity Bits		
		16K x 1	8K x 2	4K x 4	2K x 8	1K x 16	512 x 32	2K x 9	1K x 18	512 x 36
Port B	16K x 1	All Allowed						None Allowed		
	8K x 2									
	4K x 4									
	2K x 8									
	1K x 16									
	512 x 32									
	2K x 9	None Allowed						All Allowed		
	1K x 18									
	512 x 36									

Combinations	Memory Depth	Data Width	Parity Width	Data Input Data Output	ADDR	Total RAM (Kb)
18 Kb Block RAM With and Without Parity						
512 x 32	512	32	NA	[31:0]	[13:5]	16
512 x 36	512	32	4	[35:0]	[13:5]	18
1K x16	1024	16	NA	[15:0]	[13:4]	16
1K x18	1024	16	2	[17:0]	[13:4]	18
2K x 8	2045	8	NA	[7:0]	[13:3]	16
2K x 9	2048	8	1	[8:0]	[13:3]	18
4K x 4	4096	4	NA	[3:0]	[13:2]	16
8K x 2	8192	2	NA	[1:0]	[13:1]	16
16K x 1	16384	1	NA	[0:0]	[13:0]	16



FPGA Architecture



Potential Acceleration

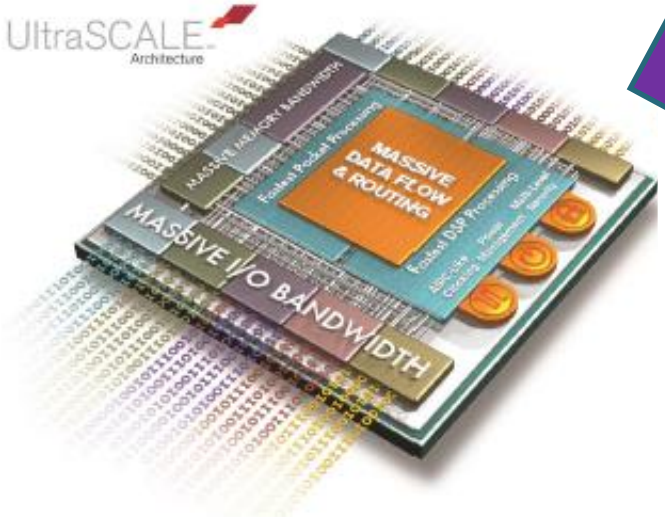
A Typical Sequential Processor

$$\frac{1.2\ \text{GHz}}{2016\ \text{clock cycles}} = 595\ \text{KSPS}$$

A Custom Parallel Processor

$$\frac{600\ \text{MHz}}{1\ \text{clock cycle}} = 600\ \text{MSPS}$$

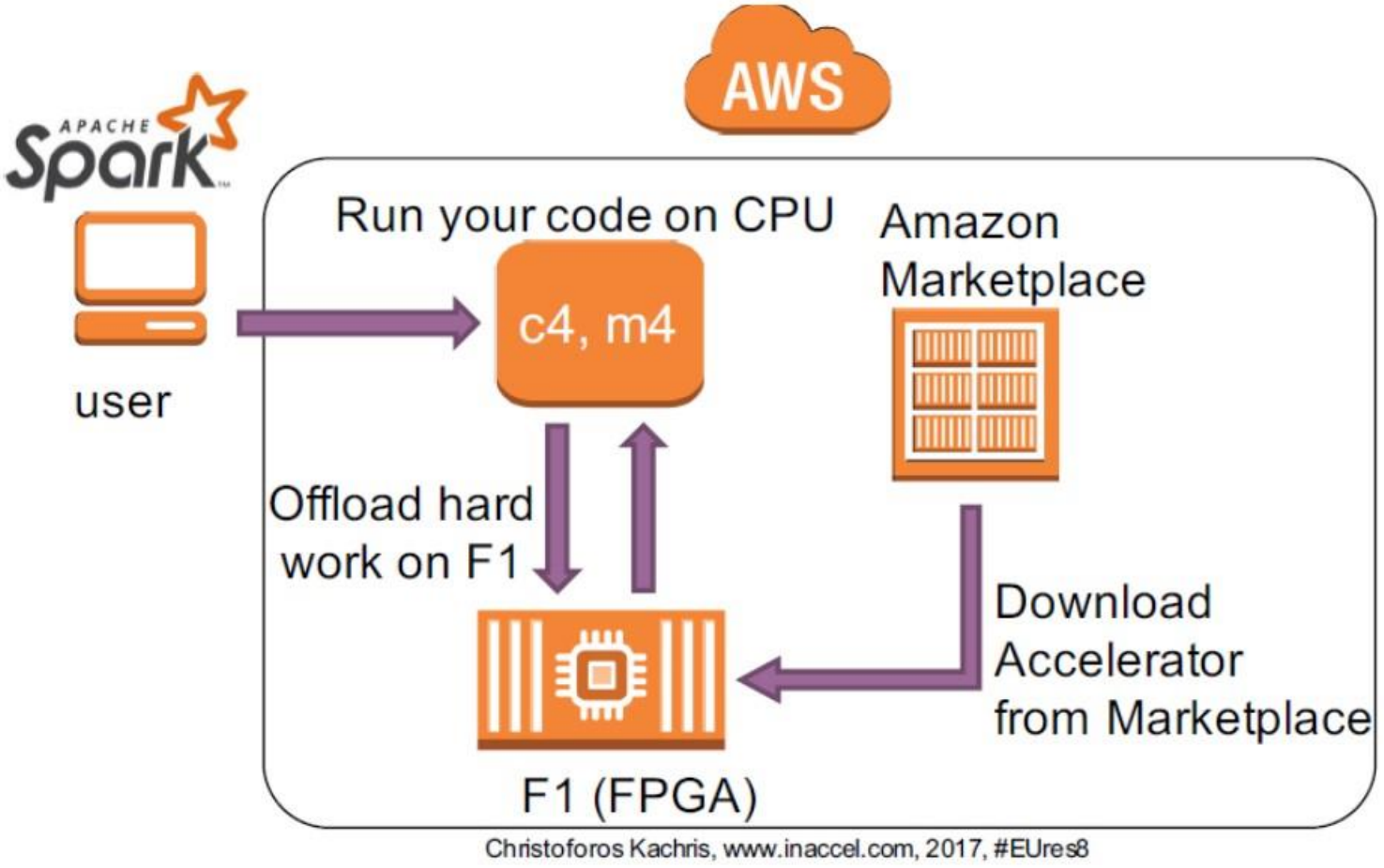
Hardware/Software Co-design




```
module filter1 (clock, rst, strm_in, out)
  integer i,j; //index for loops
  always@(posedge clock)
  for (i=0; i<NUMUNITS; i=i+1)
    tmp_kernel[j] = k[i*OFFSETX];
```

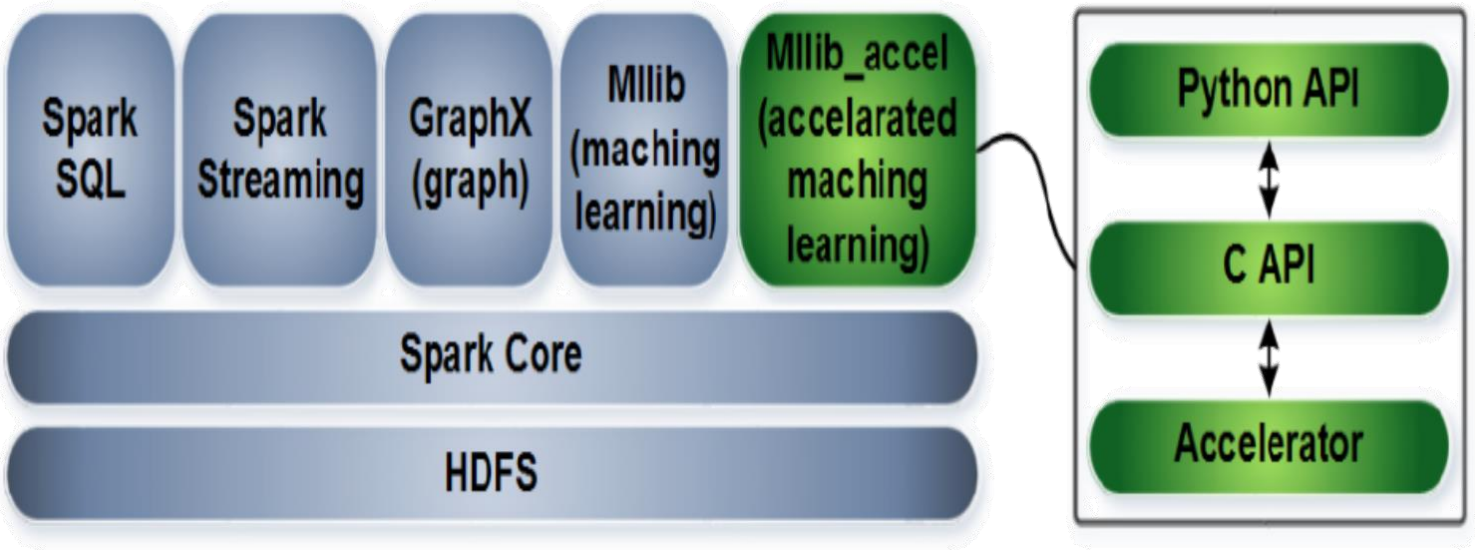
Amazon AWS f1

Spark plus FPGA in AWS f1

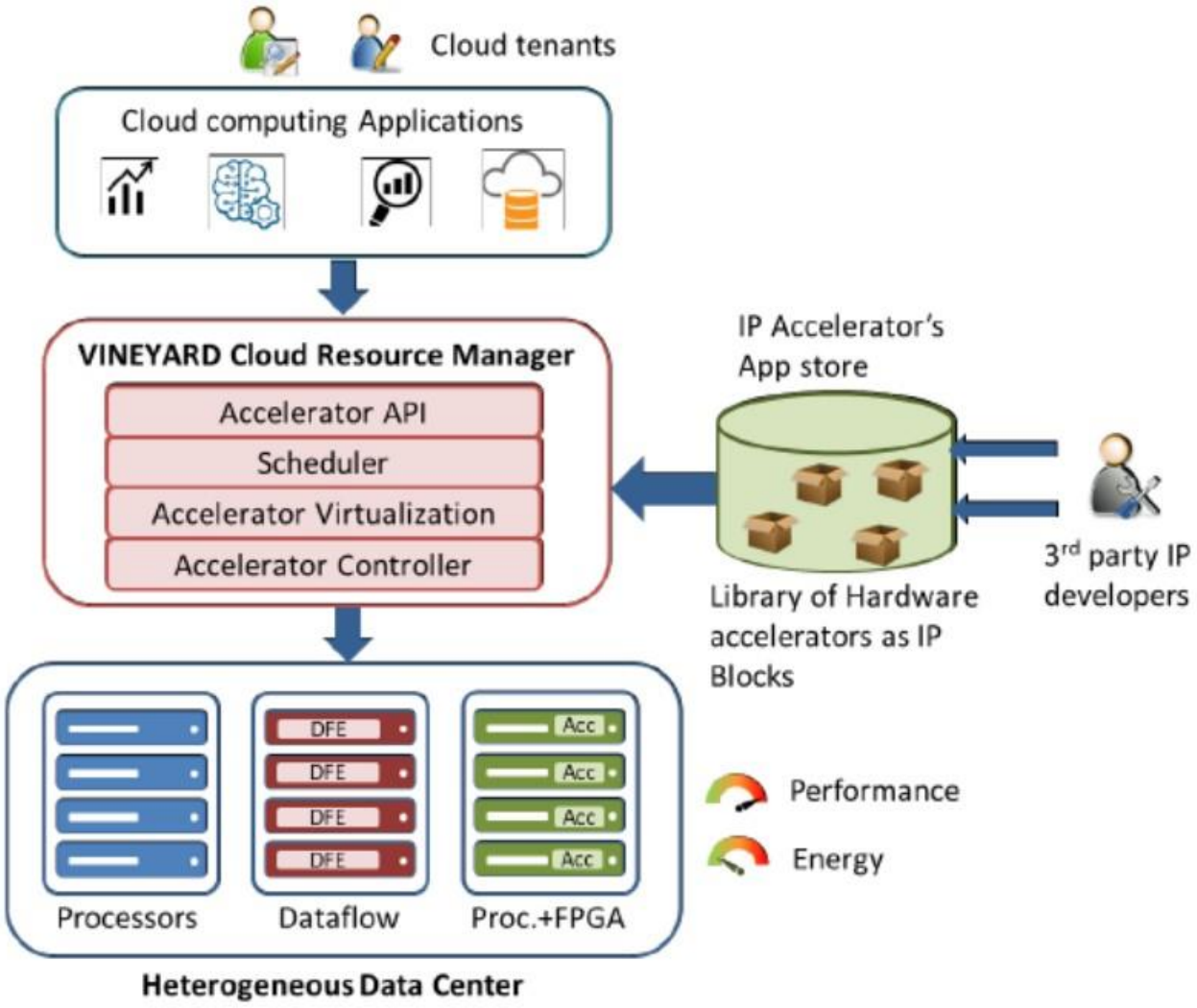


Christoforos Kachris, www.inaccel.com, 2017, #EUres8

Spark Mllib Extension



Seamless FPGA Integration



FPGA in AWS f1