Parallel computing and GPU introduction

黃子桓 ‹tzhuan@gmail.com›

Agenda

- Parallel computing
- GPU introduction
- Interconnection networks
- Parallel benchmark
- Parallel programming
- GPU programming

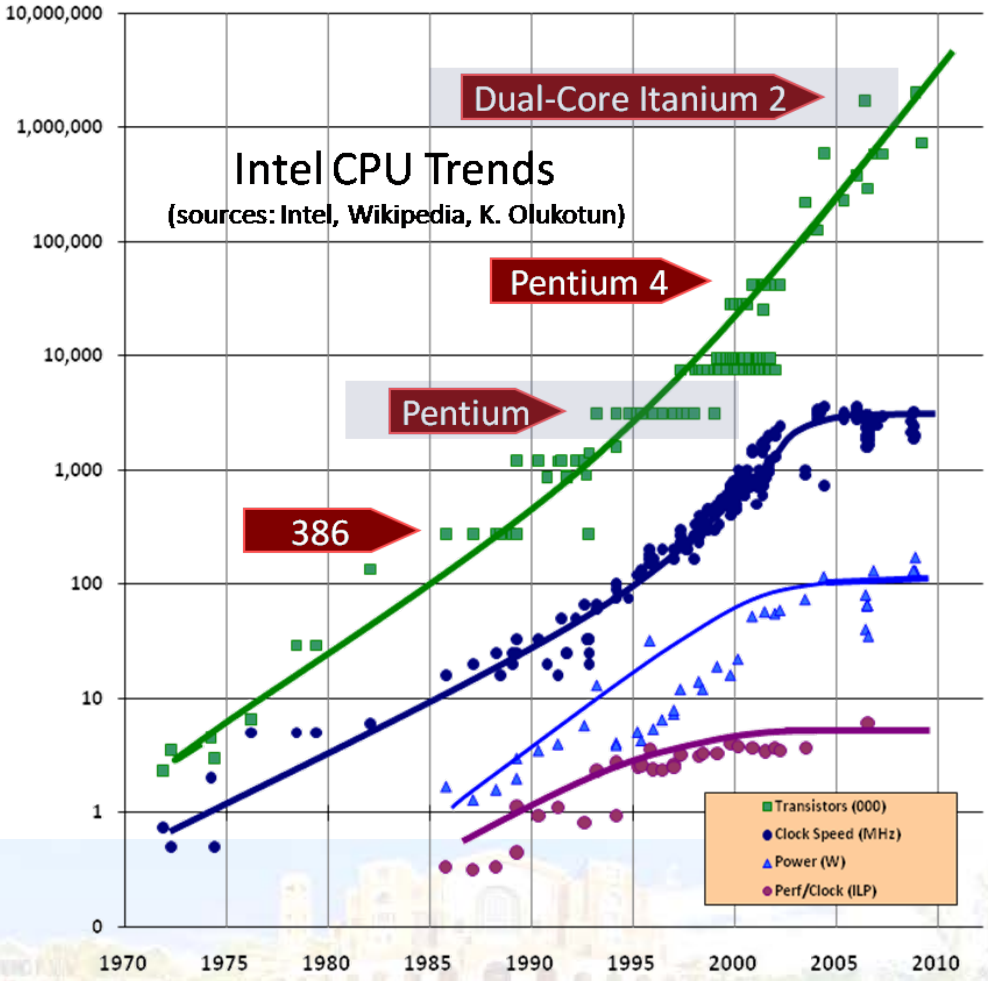
Parallel computing

Goal of computing

Faster, faster and faster

Why parallel computing?

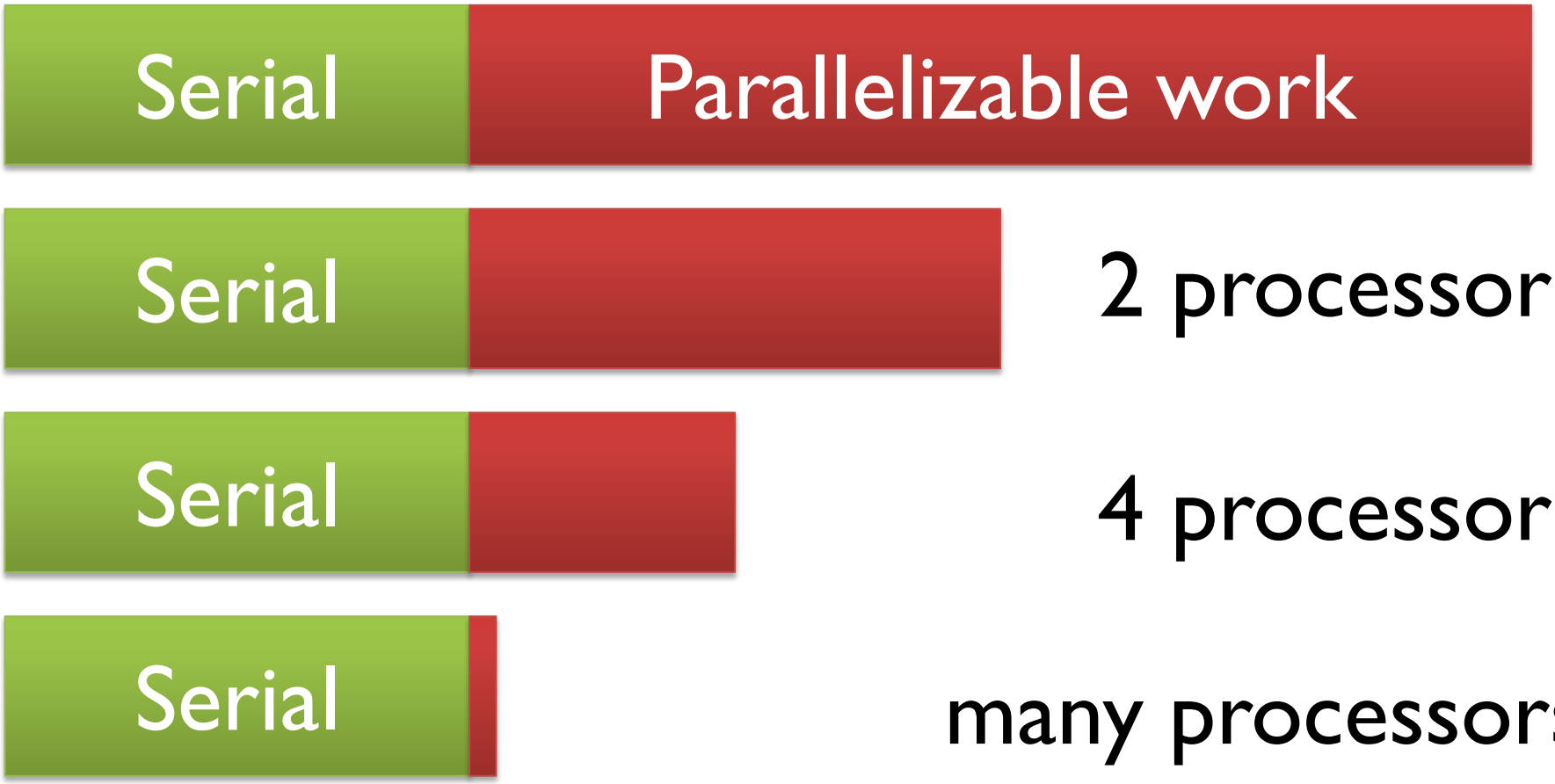
- Moore's law is dead (for CPU frequency)



Top500 (Nov 2013)

1. Tianhe-2(NUDT)
   - 3,120,000 cores (Intel Xeon E5, Intel Xeon Phi)
2. Titan (Cray)
   - 560,640 cores (Opetron 6274, NVIDIA K20x)
3. Sequoia (IBM)
   - 1,572,864 cores (Power BQC)
4. K computer (Fujitsu)
   - 705,024 cores (Sparc64)
5. Mira (IBM)
   - 786,432 cores (Power BQC)

Amdahl's law


Serial
Parallelizable work
Serial
2 processors
Serial
4 processors
Serial
many processors

Amdahl's law

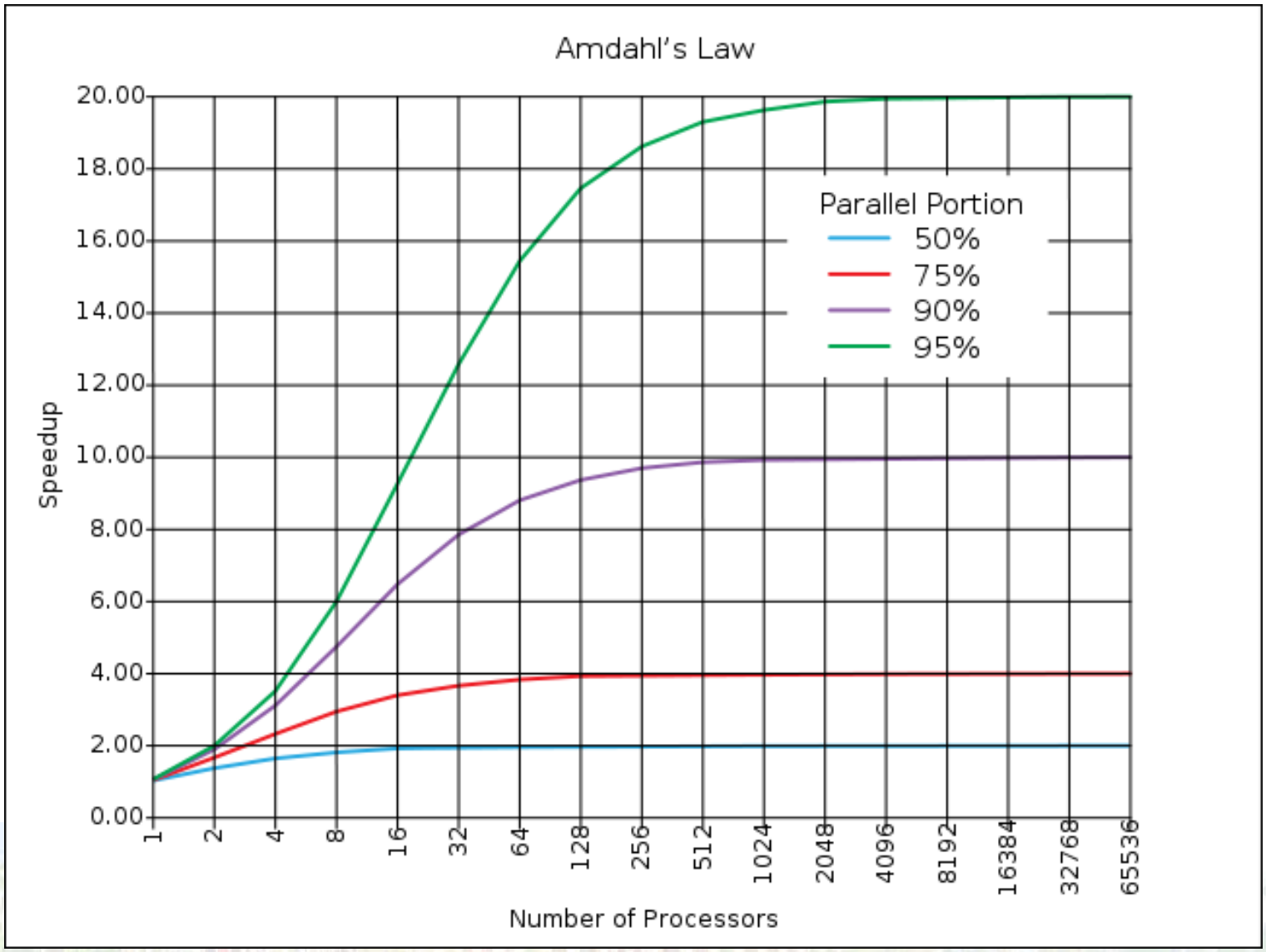
$$\text{Total speedup} = \frac{1}{(1-P)+P/S}$$

Parallelizable work

Speedup for Parallelizable work

Example:
P: 0.8 (80% work is parallelizable)
S: 8 (8 processors)
Total speedup: 3.33x

Amdahl's law


Amdahl's Law
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Parallel Portion
50%
75%
90%
95%
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors

Scaling exapmle

- Workload: sum of 10 scalars, and 10×10 matrix sum
- Single processor
  - Time = (10+100)×Tadd = 110×Tadd
- 10 processors
  - Time = 10×Tadd + (100/10)×Tadd = 20×Tadd
  - Speedup = 110/20 = 5.5× (55% of potential)
- 100 processors
  - Time = 10×Tadd + (100/100)×Tadd = 11×Tadd
  - Speedup = 110/11 = 10× (10% of potential)

Scaling example

- What if matrix size is 100×100?
- Single processor
  - Time = (10+10000)×Tadd = 10010×Tadd
- 10 processors
  - Time = 10×Tadd + (10000/10)×Tadd = 1010×Tadd
  - Speedup = 10010/1010 = 9.9× (99% of potential)
- 100 processors
  - Time = 10×Tadd + (10000/100)×Tadd = 110×Tadd
  - Speedup = 10010/110 = 91× (91% of potential)

Scalability

- The ability of a system to handle growing amount of work

- Strong scaling
  - Fixed total problem size
  - Run a fixed problem faster
- Weak scaling
  - Fixed problem size per processor
  - Run a bigger (or smaller) problem

Parallel computing system

- Parallelization design for processors
- Hardware multithreading
- Multi-processor system
- Cluster computing system
- Grid computing system

Parallelization design for processors

- Instruction level parallelism

```
add $t0, $t1, $t2
mul $t3, $t4, $t5
```

- Data level parallelism

```
add 0($t1), 0($t2), 0($t3)
add 4($t1), 4($t2), 4($t3)
add 8($t1), 8($t2), 8($t3)
```

Flynn's taxonomy

	Single instruction	**Multiple instruction**
Single data	SISD (Single-core processor)	MISD (very rare)
Multiple data	SIMD (Superscalar, vector processor, GPU, etc.)	MIMD (Multi-core processor)

SIMD

- Operate element-wise on vectors of data
  - MMX and SSE instructions in x86
  - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time, each with different data address
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications

Example: dot product

```
    mov esi, dword ptr [src]
    mov edi, dword ptr [dst]
    mov ecx, Count
start:
    movaps xmm0, [esi] //a3, a2, a1, a0
    mulps xmm0, [esi + 16] //a3*b3,a2*b2,a1*b1,a0*b0
    haddps xmm0, xmm0 //a3*b3+a2*b2,a1*b1+a0*b0,
                      //a3*b3+a2*b2,a1*b1+a0*b0
    movaps xmm1, xmm0
    psrldq xmm0, 8
    addss xmm0, xmm1
    movss [edi],xmm0
    add esi, 32
    add edi, 4
    sub ecx, 1
    jnz start
```

Vector processors

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- Example: Vector extension to MIPS
  - 32 × 64-element registers (64-bit elements)
  - Vector instructions
    - lv, sv: load/store vector
    - addv.d: add vectors of double
    - addvs.d: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

Example: DAXPY (Y = a × X + Y)

- Conventional MIPS code

```
        l.d   $f0,a($sp)      ;load scalar a
        addiu r4,$s0,#512     ;upper bound of what to load
loop:   l.d   $f2,0($s0)      ;load x(i)
        mul.d $f2,$f2,$f0     ;a × x(i)
        l.d   $f4,0($s1)      ;load y(i)
        add.d $f4,$f4,$f2     ;a × x(i) + y(i)
        s.d   $f4,0($s1)      ;store into y(i)
        addiu $s0,$s0,#8      ;increment index to x
        addiu $s1,$s1,#8      ;increment index to y
        subu  $t0,r4,$s0      ;compute bound
        bne   $t0,$zero,loop  ;check if done
```

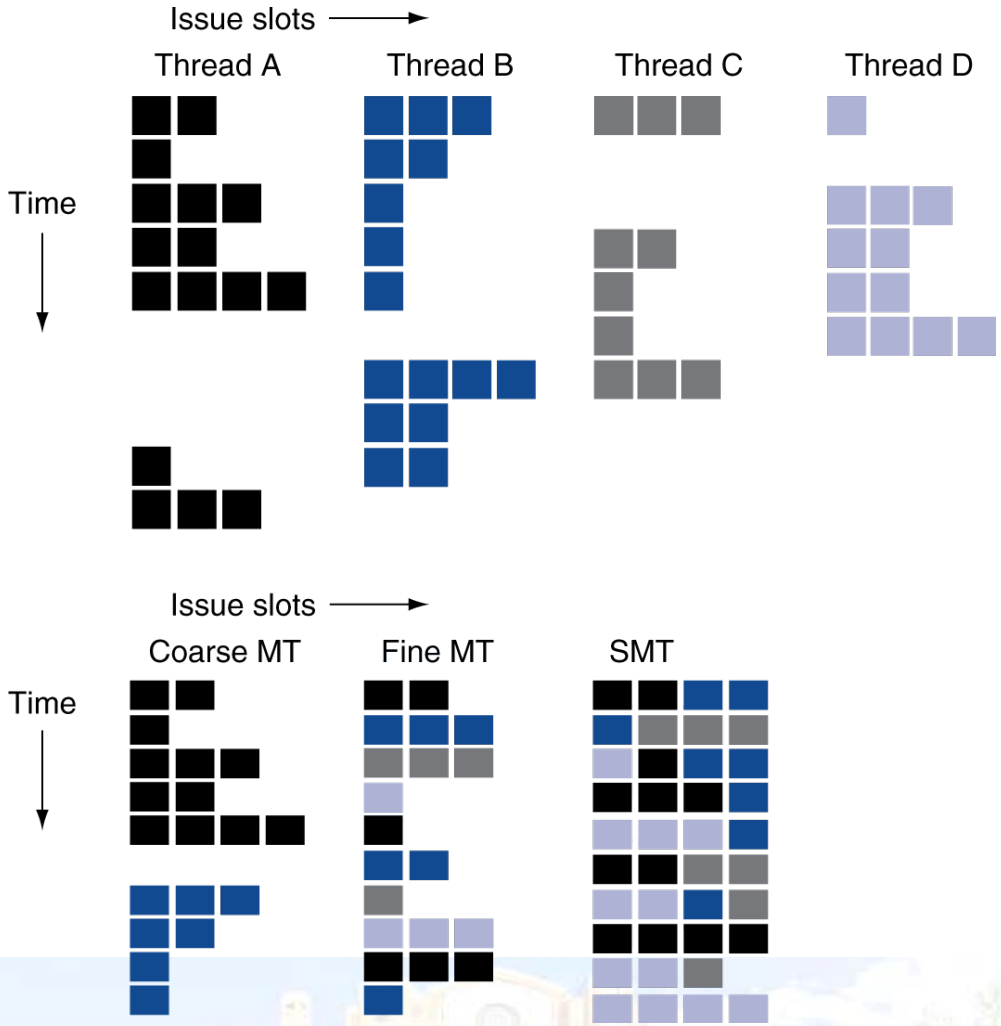
- Vector MIPS code

```
        l.d     $f0,a($sp)    ;load scalar a
        lv      $v1,0($s0)    ;load vector x
        mulvs.d $v2,$v1,$f0   ;vector-scalar multiply
        lv      $v3,0($s1)    ;load vector y
        addv.d  $v4,$v2,$v3   ;add y to product
        sv      $v4,0($s1)    ;store the result
```

Hardware multithreading

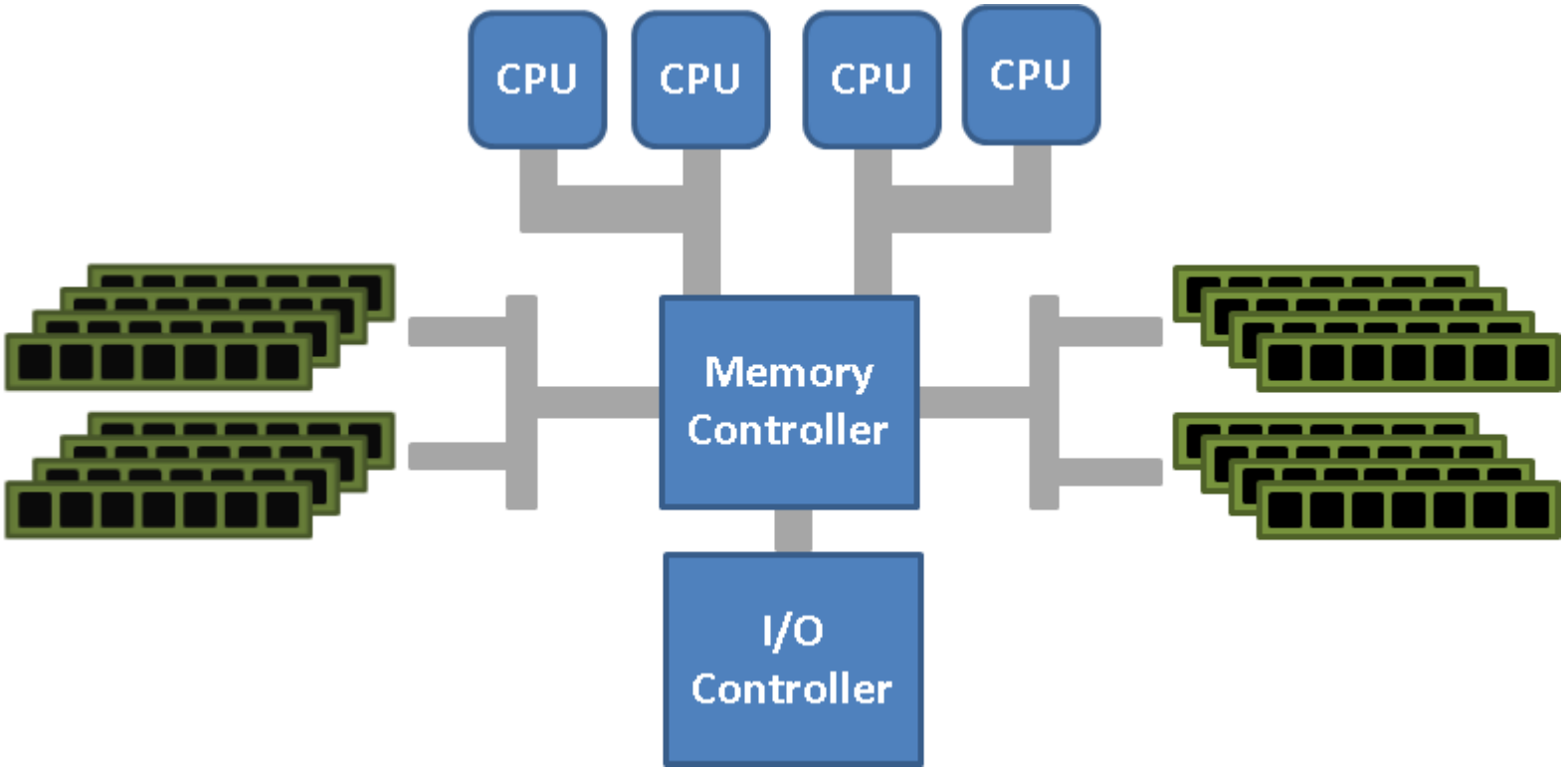
- Allows multiple threads to share the functional units of a single processor in an overlapping fashion
  - Coarse-grained multithreading
    - Switches threads only on costly stall
  - Fine-grained multithreading
    - Interleaved execution of multiple threads
  - Simultaneous multithreading
    - Multiple-issue, dynamically scheduled processor to exploit thread-level parallelism

Hardware multithreading



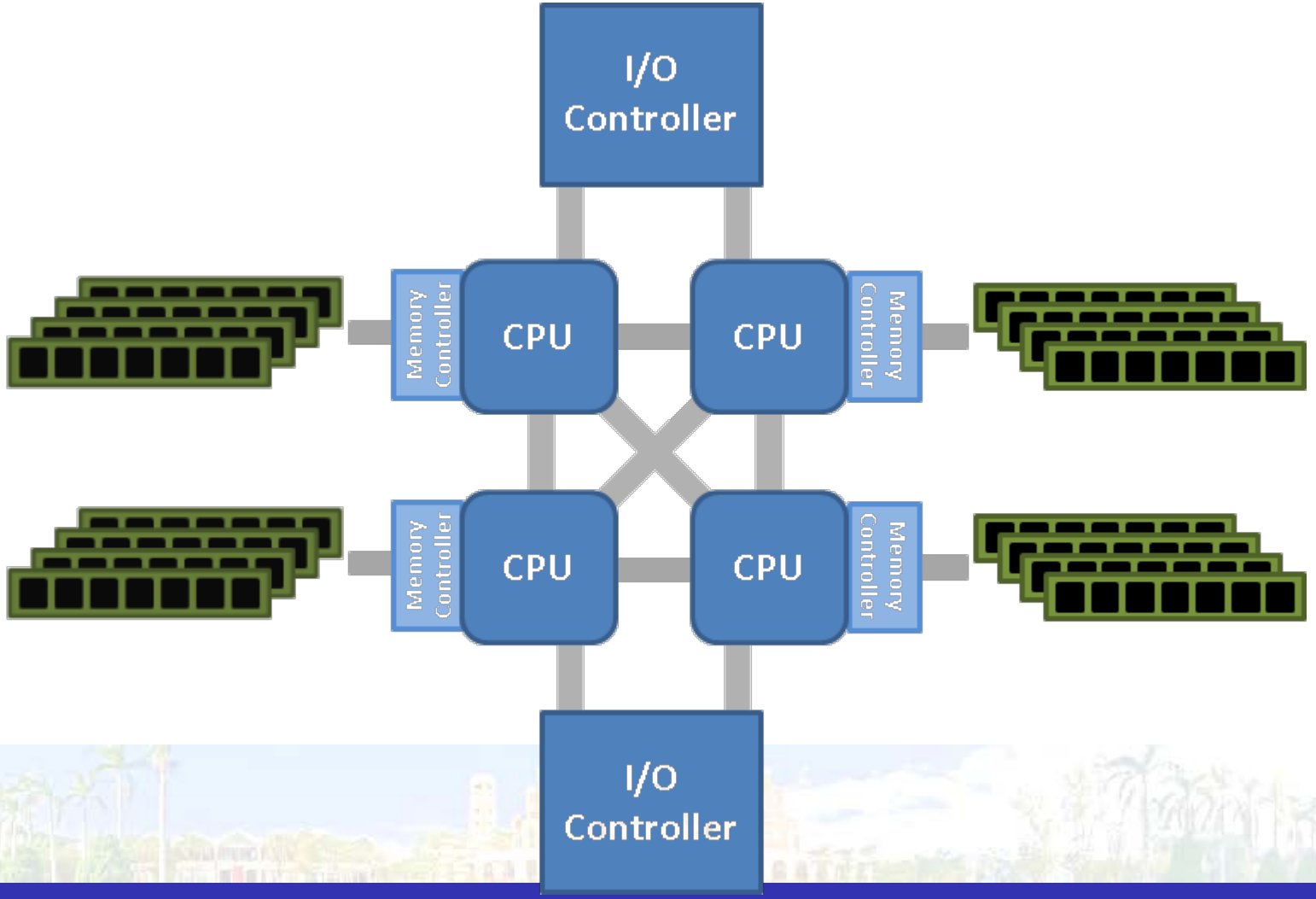
Multi-processor system

- Shared memory multi-processor

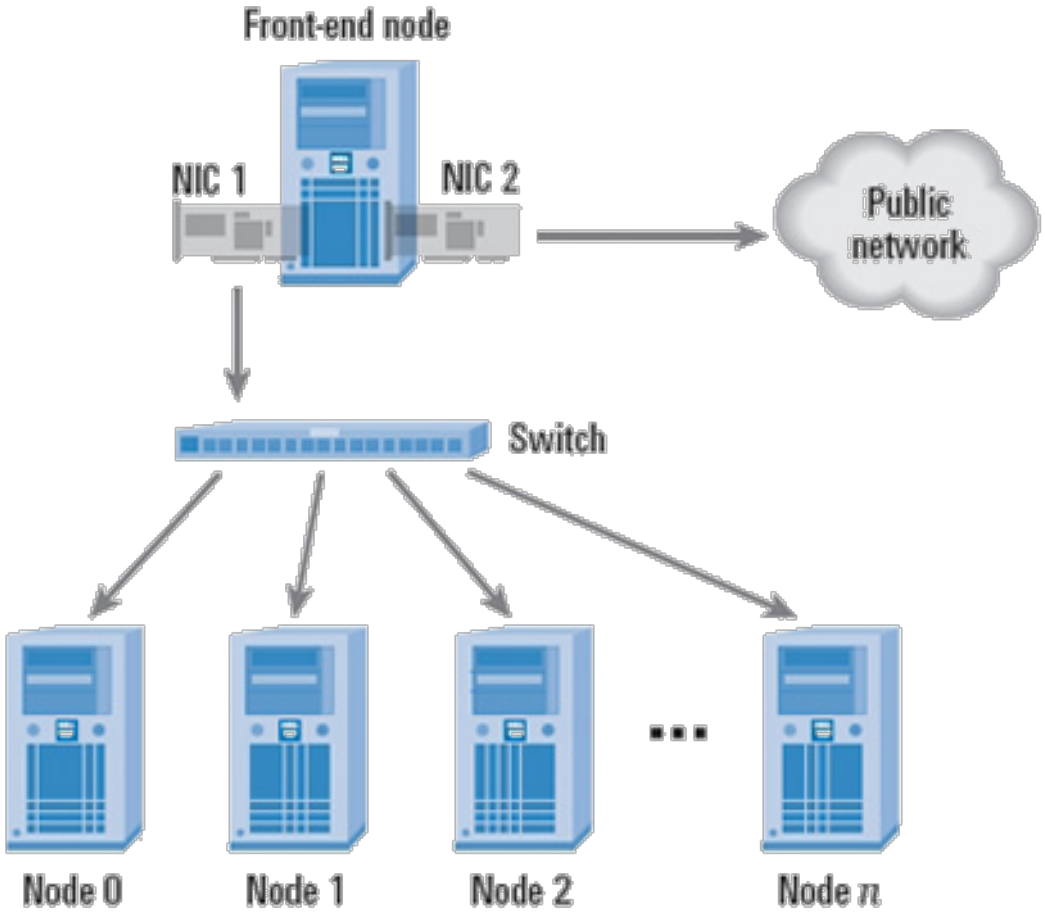


Multi-processor system

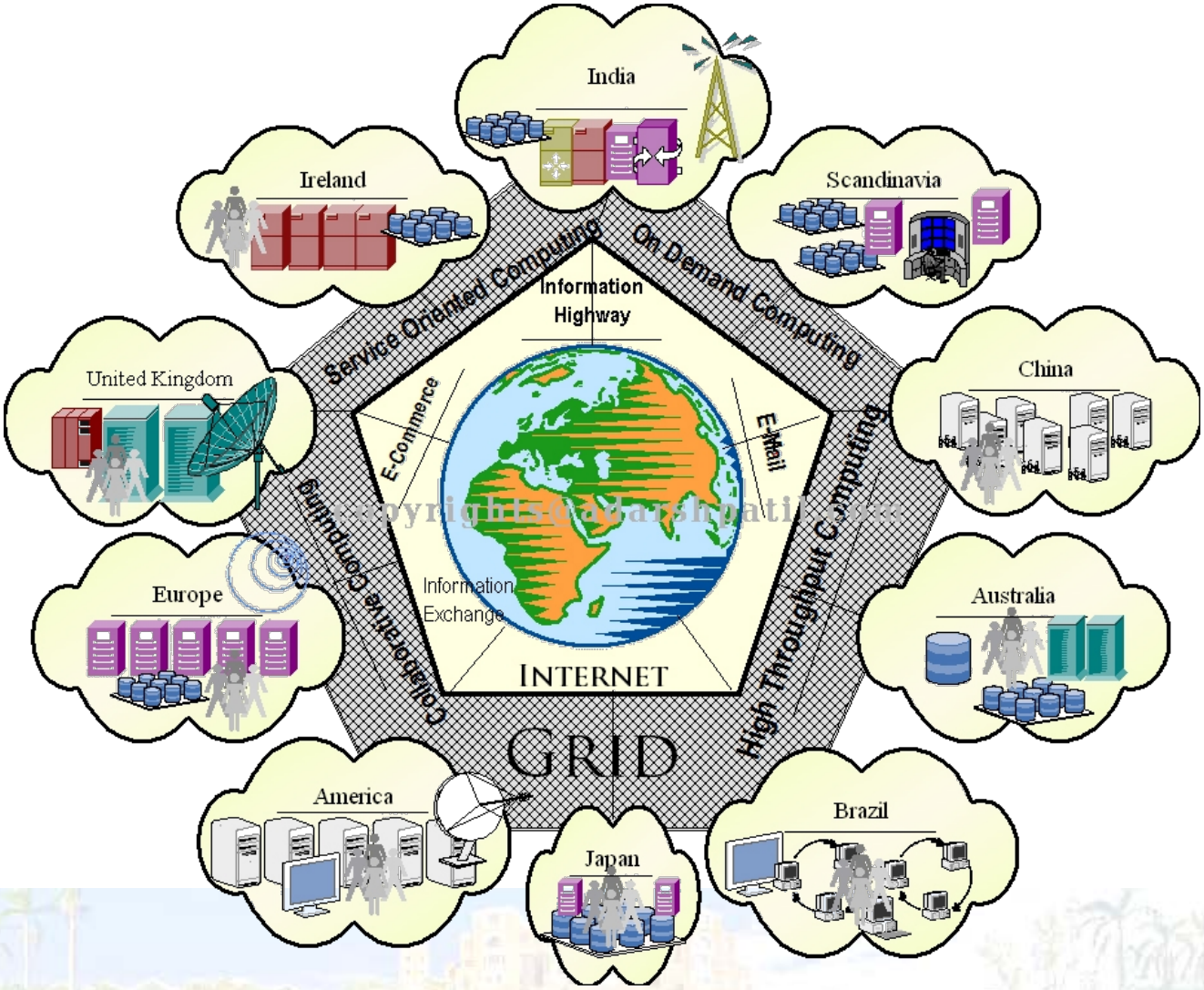
- Non-uniform memory access multi-processor



Cluster computing system

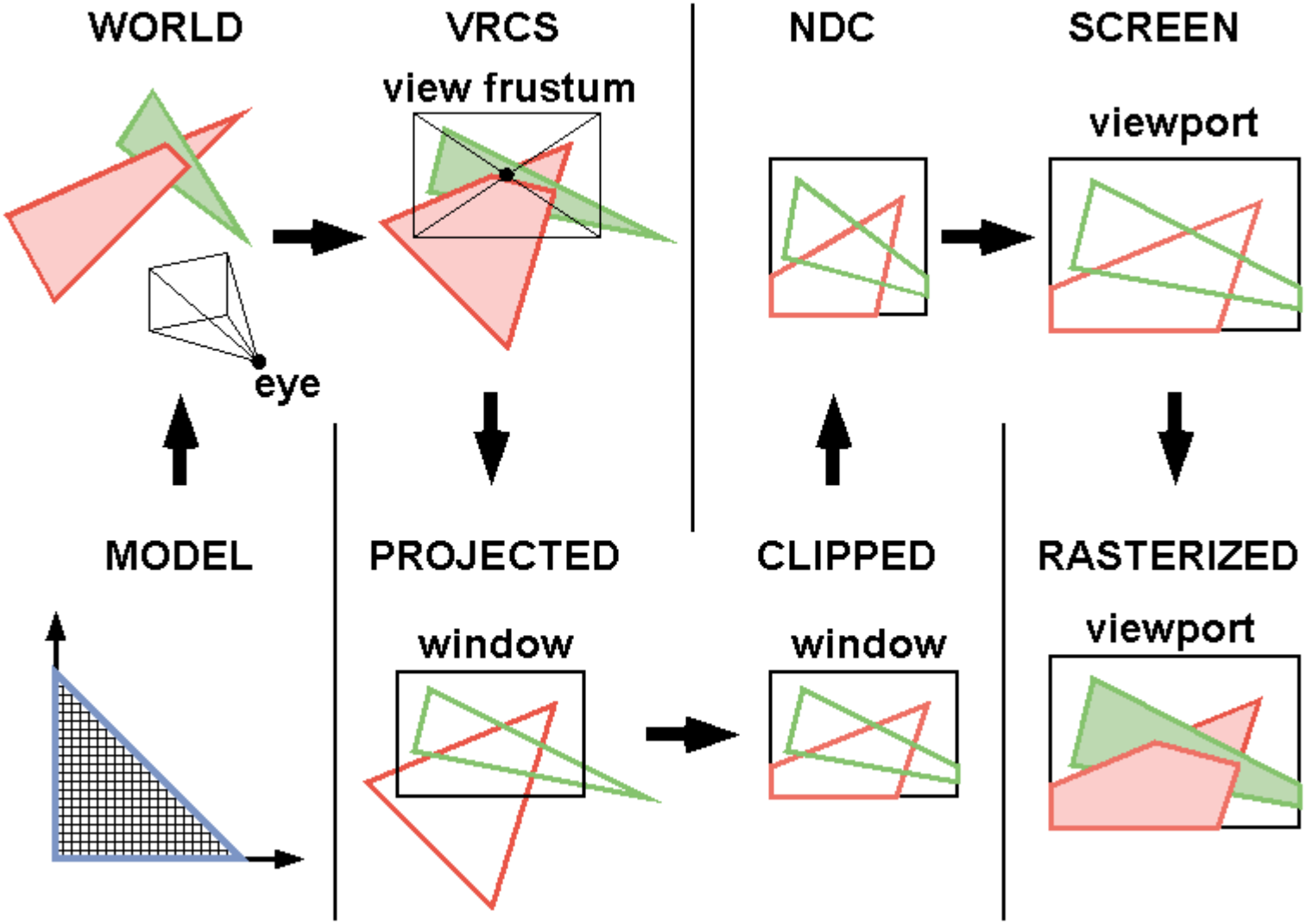


Grid computing system



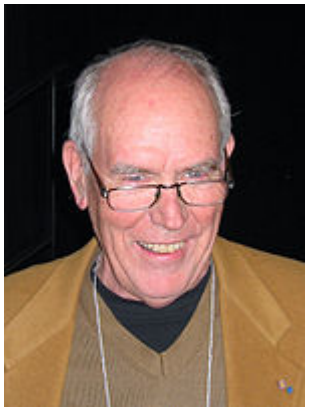
GPU introduction

Computer graphics rendering



History of computer graphics

- 1960, Ivan Sutherland's Sketchpad
  - The beginning of computer graphics
- 1992, OpenGL 1.0
- 1996, Voodoo I
  - The first consumer 3D graphics card
- 1996, DirectX 3.0
  - The first version including Direct3D



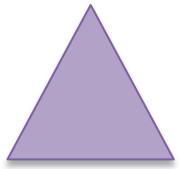
The history of computer graphics

- 2000, DirectX 8.0
  - The first version supporting HLSL
- 2001, GeForce 3 (NV20)
  - The first consumer GPU
- 2004, OpenGL 2.0
  - The first version supporting GLSL
- 2006, GeForce 8 (G80)
  - The first NVIDIA GPU supporting CUDA
- 2008
  - OpenCL (Apple, AMD, IBM, Qualcomm, Intel, …)

Graphics pipeline

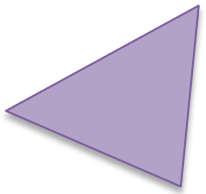
Instructions
States
Data

Input processor



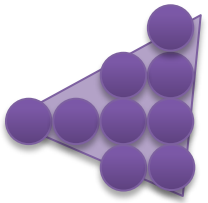
Transforms
Lighting, etc

Do geometry stuff



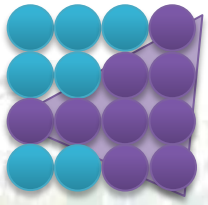
Rasterize
Pixel shading

Do pixel stuff

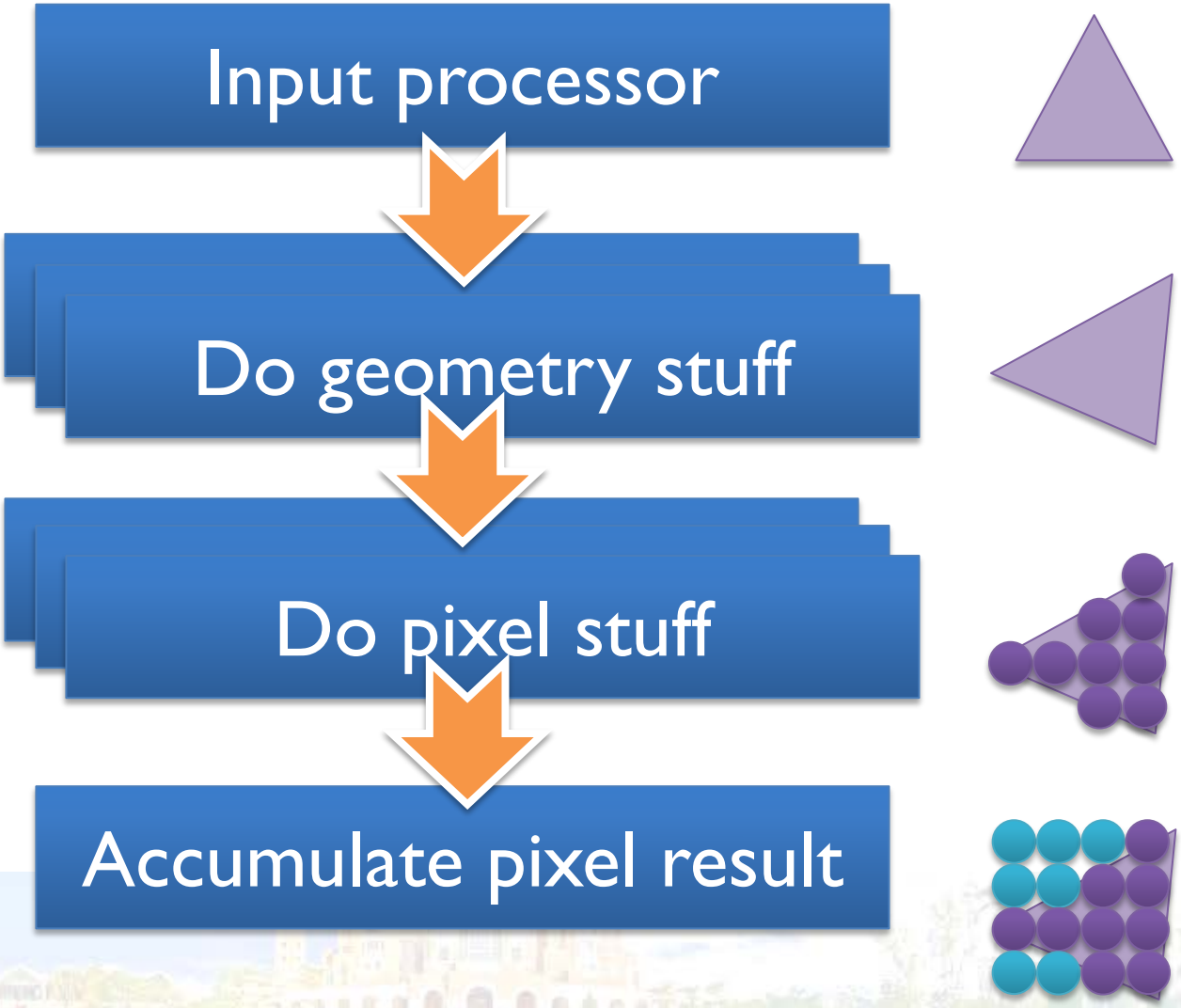


Z-buffer
Transparency

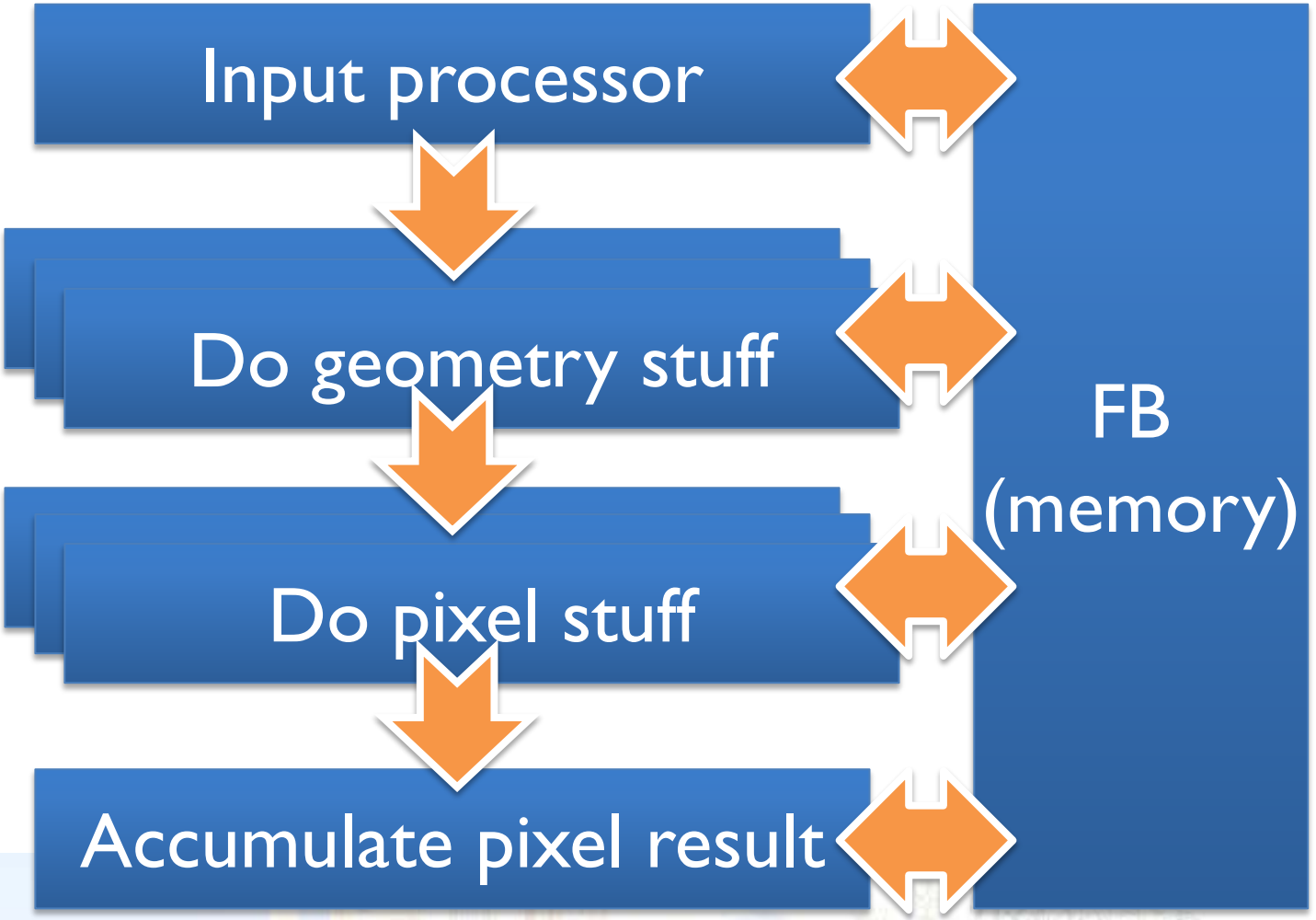
Accumulate pixel result



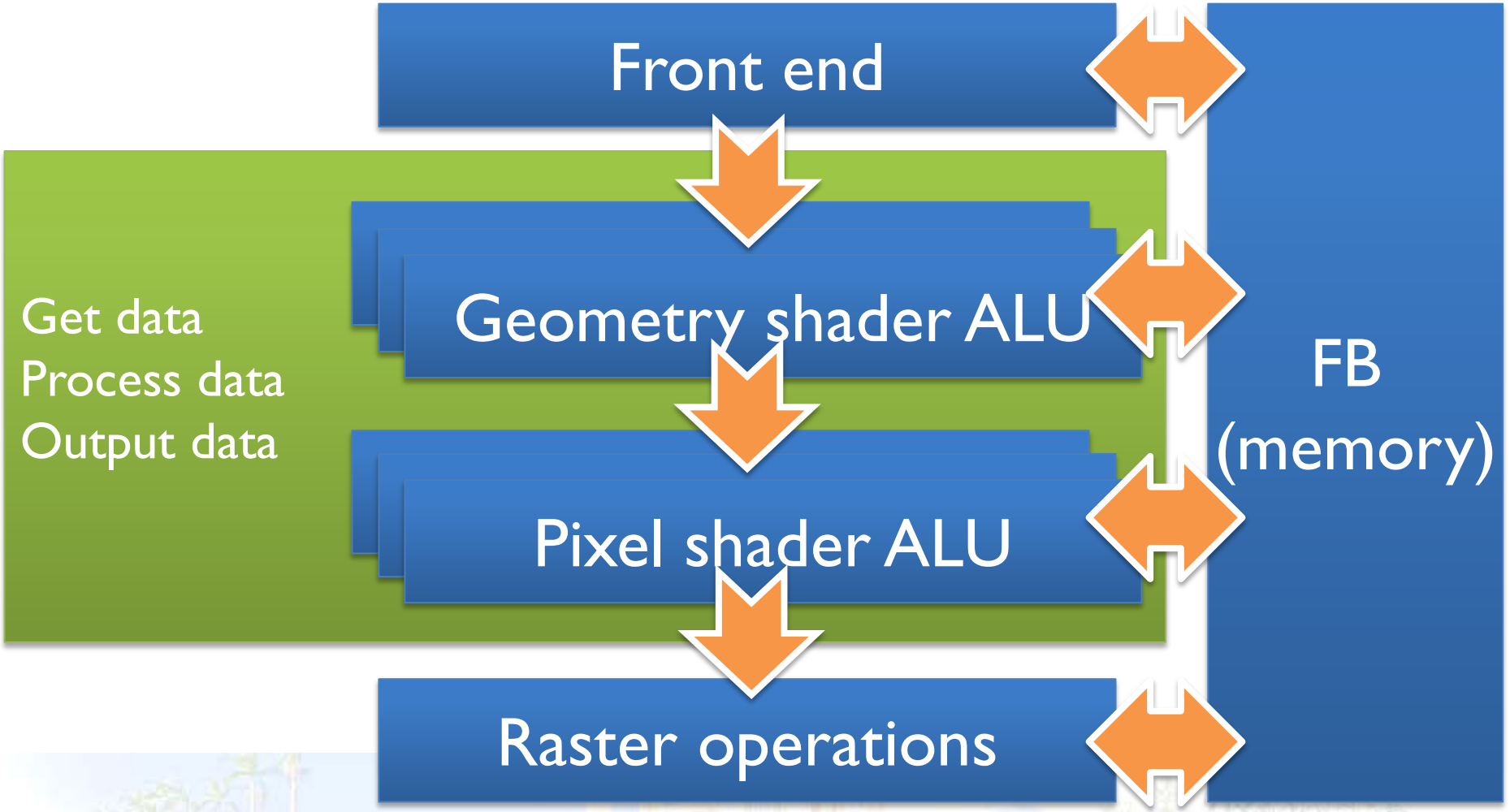
Make it faster



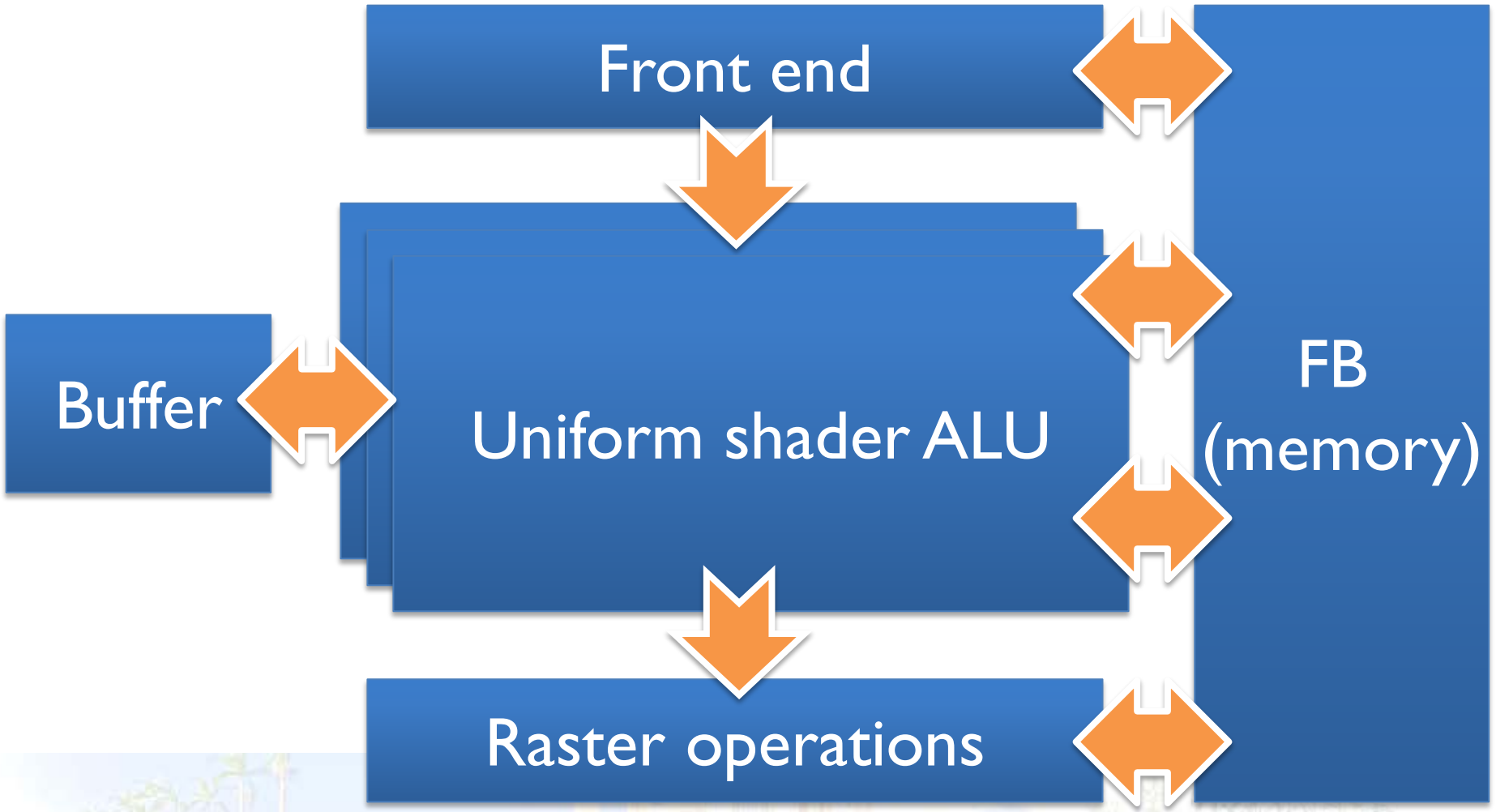
Add framebuffer support



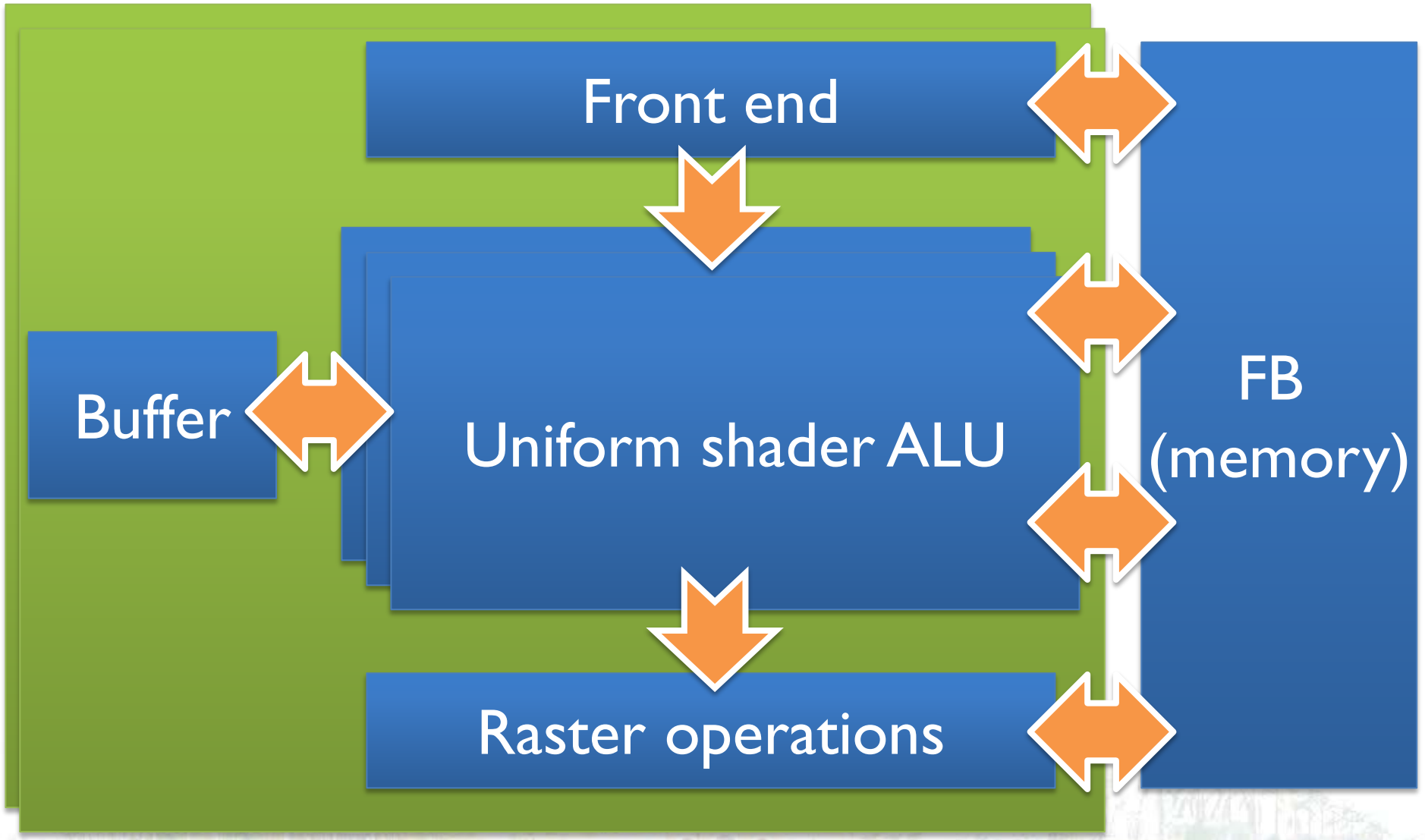
Add programmability



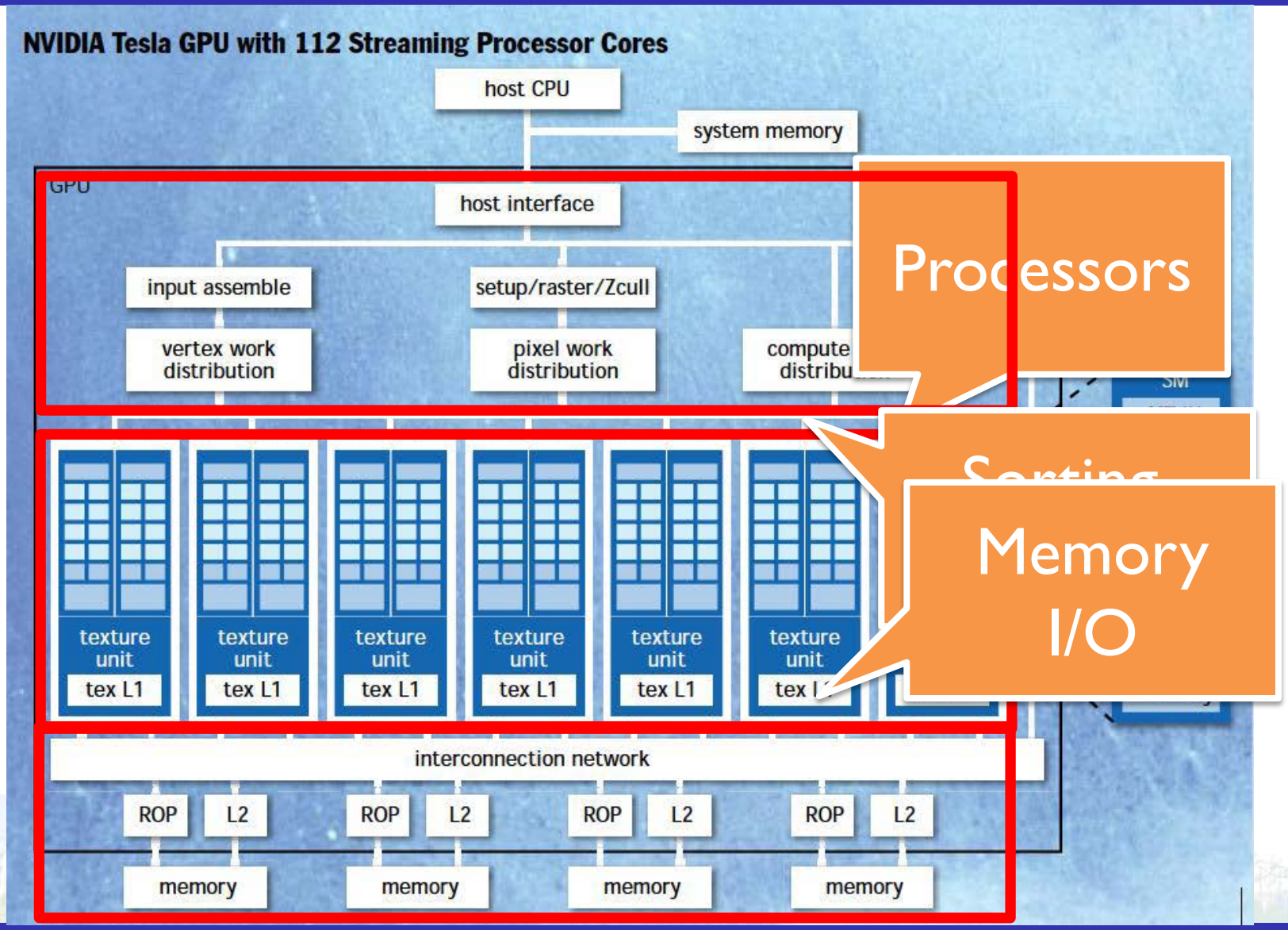
Uniform shader



Scaling it up again



NVIDIA tesla GPU



Interconnection networks

Interconnection networks

Bus

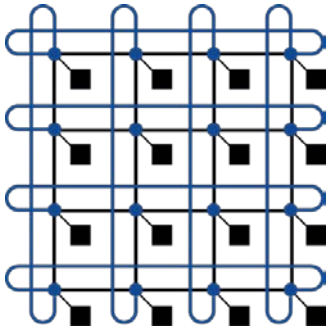
Ring

- Performance metric
  - Network bandwidth, the peek transfer rate of a network (best case)
  - Bisection bandwidth, the bandwidth between two equal parts of a multiprocessor (worse case)

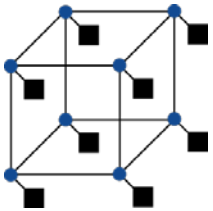
Network topologies

Bus

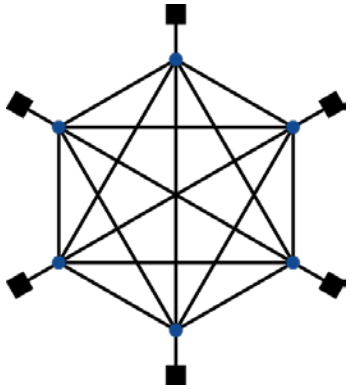
Ring



2D mesh

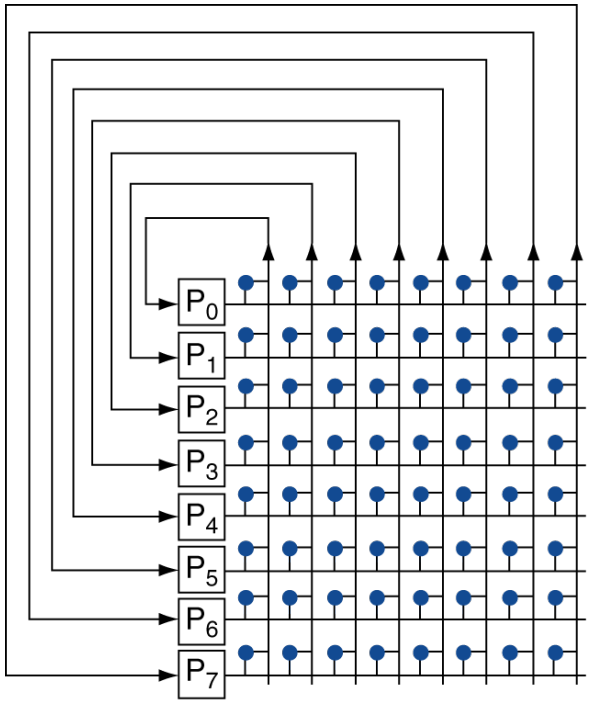


N-cube

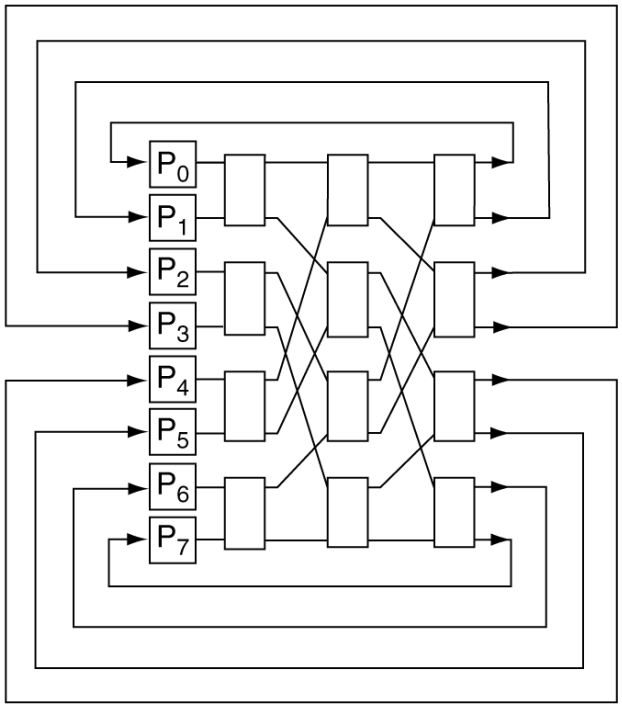


Fully connected

Multistage networks



Crossbar



Omega network

Network characterisitics

- Performance
  - Latency
  - Throughput
    - Link bandwidth
    - Total network bandwidth
    - Bisection bandwidth
  - Congestion delays
- Cost
- Power
- Routability in silicon

Parallel benchmark

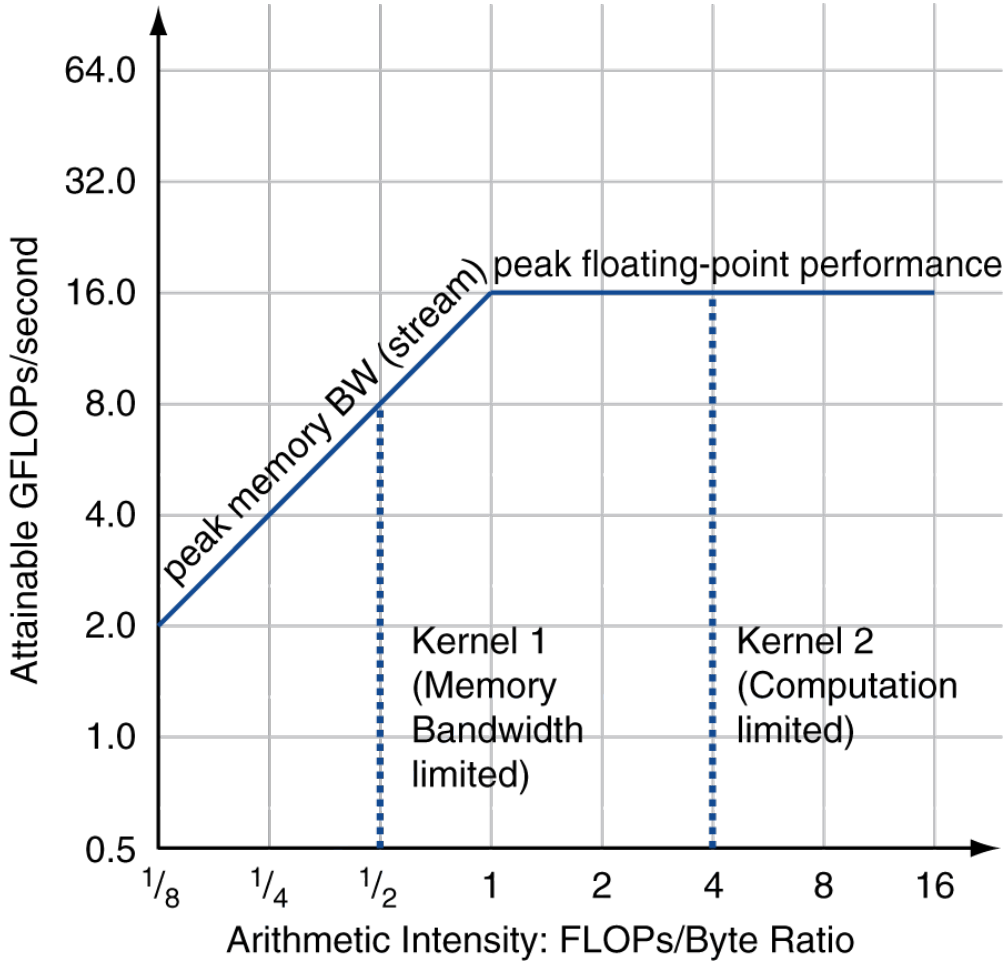
Parallel benchmark

- Linpack: matrix linear algebra
- SPECrate: parallel run of SPEC CPU programs
  - Job-level parallelism
- SPLASH: Stanford Parallel Applications for Shared Memory
  - Mix of kernels and applications, strong scaling
- NAS (NASA Advanced Supercomputing) suite
  - computational fluid dynamics kernels
- PARSEC (Princeton Application Repository for Shared Memory Computers) suite
  - Multithreaded applications using Pthreads and OpenMP

Modeling performance

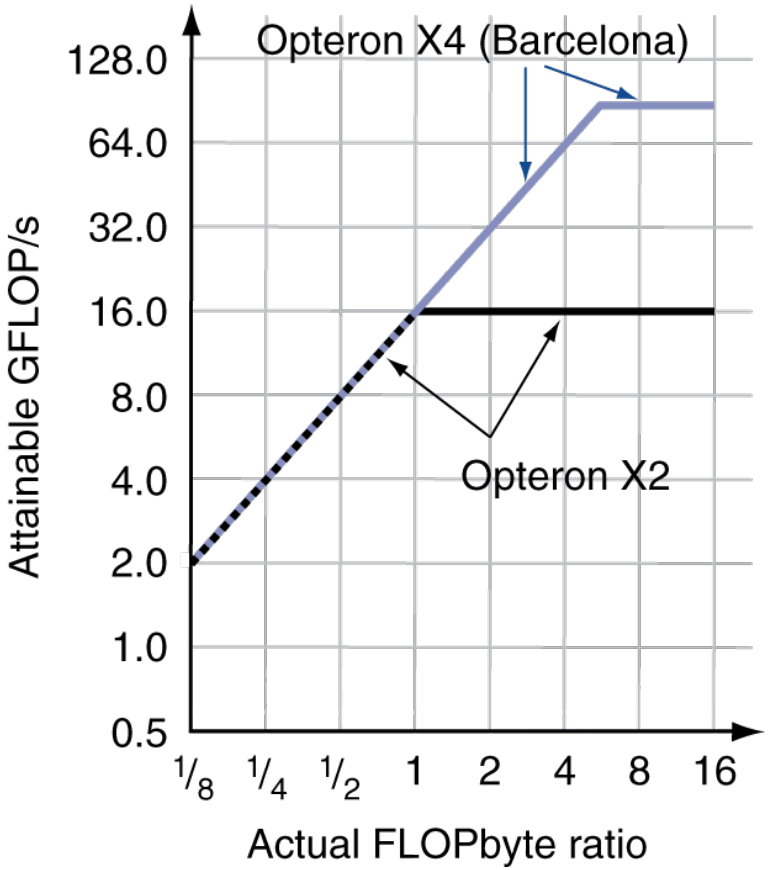
- What is the performance metric of interest?
  - Attainable GFLOPs/second
  - Measured using computational kernels from Berkeley Design Pattern
- Arithmetic intensity
  - FLOPs per byte of memory access
- For a given computer, determine
  - Peak FLOPs
  - Peak memory bytes/second

Roofline diagram

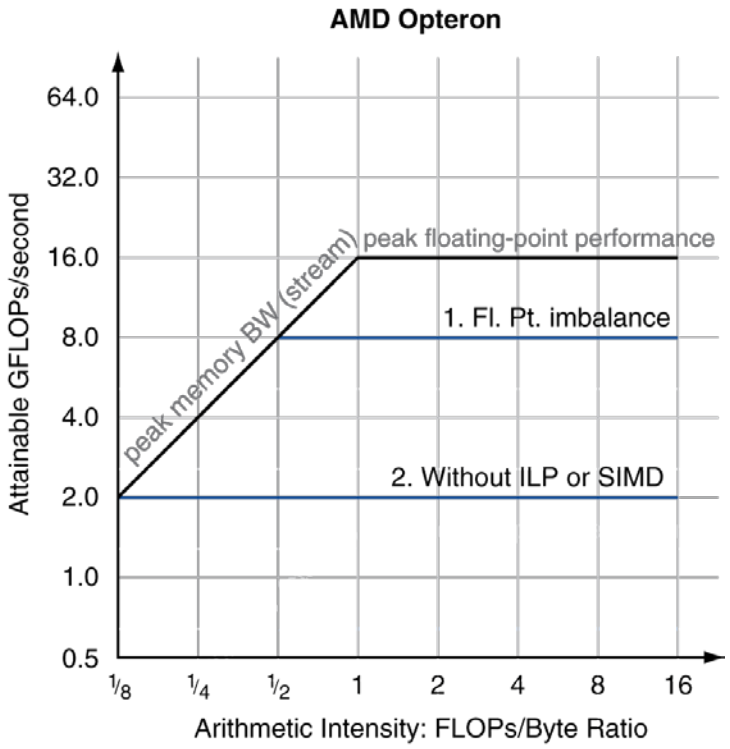


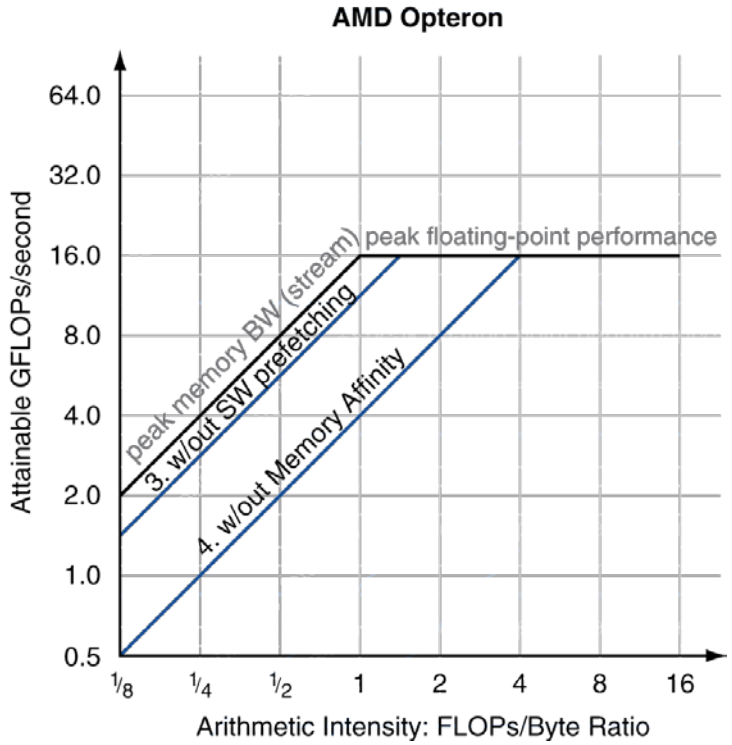
Comparing systems

- Opteron X2 vs. X4
  - 2-core vs. 4-core
  - 2.2GHz vs. 2.3GHz
  - Same memory system



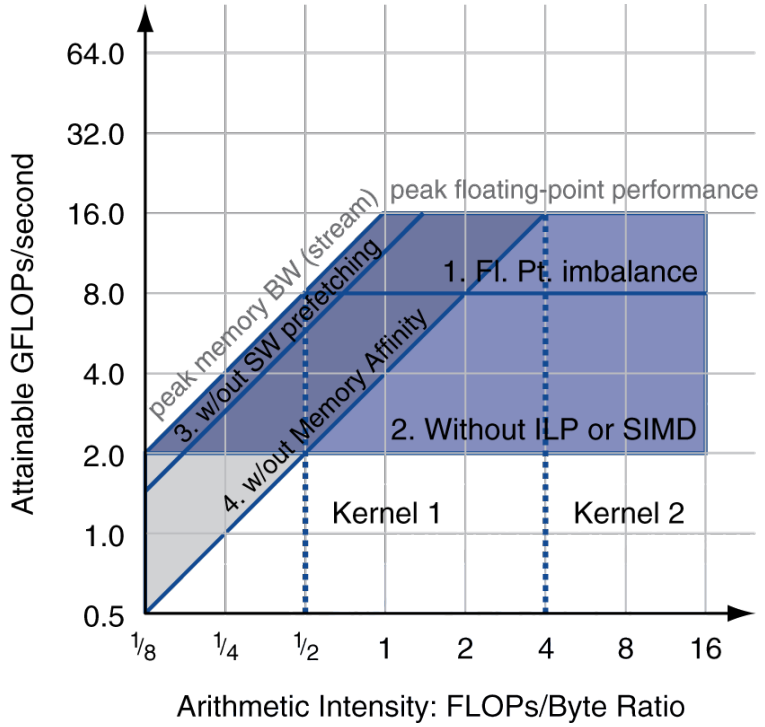
Optimizing performance



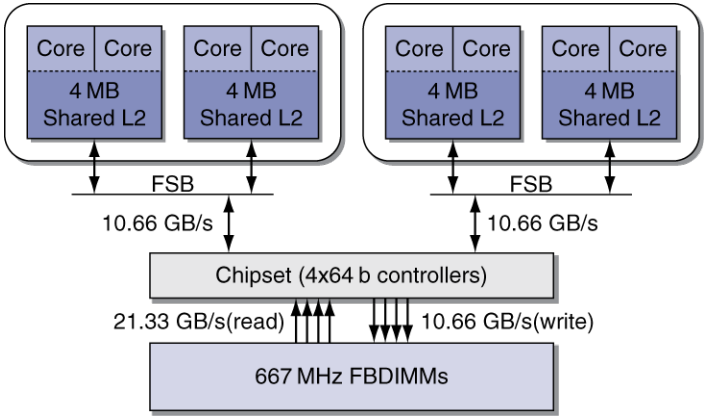


Optimizing performance

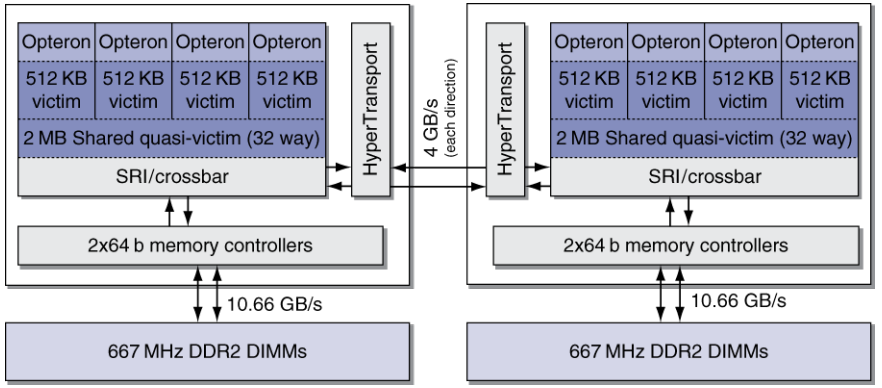
- Choices of optimization depends on arithmetic intensity of code



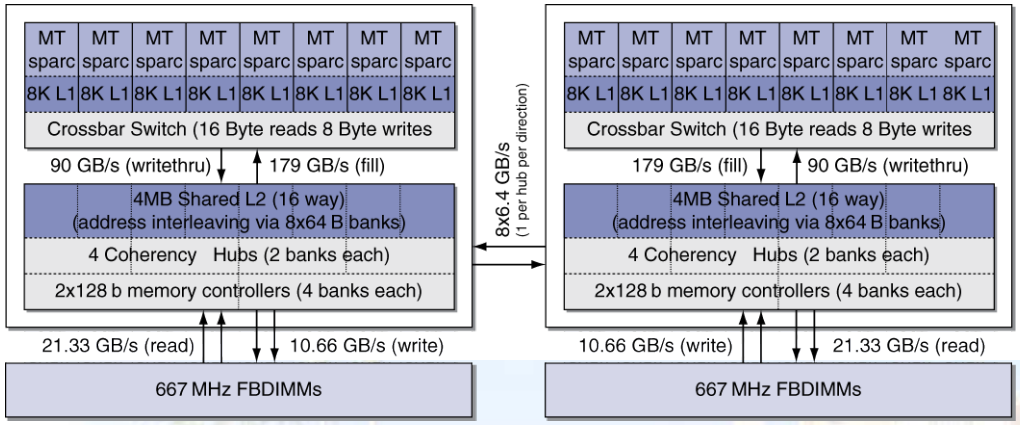
Four example systems



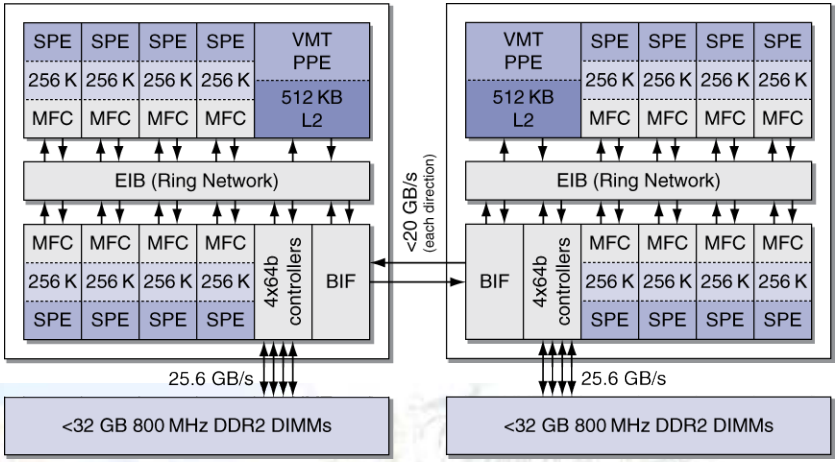
Intel Xeon e5345



AMD Opteron X4 2356

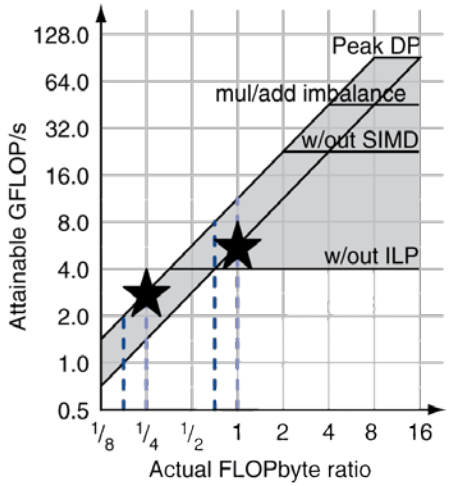


Sun UltraSPARC T2 5140

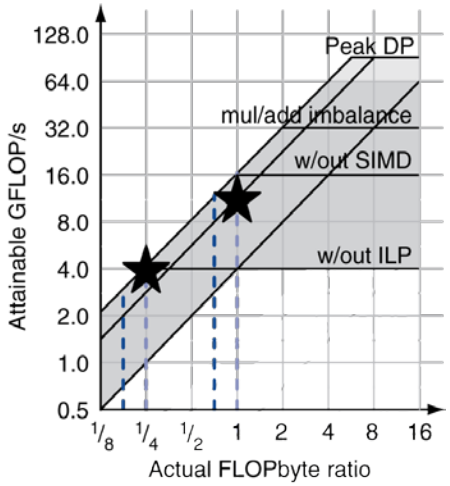


IBM Cell QS20

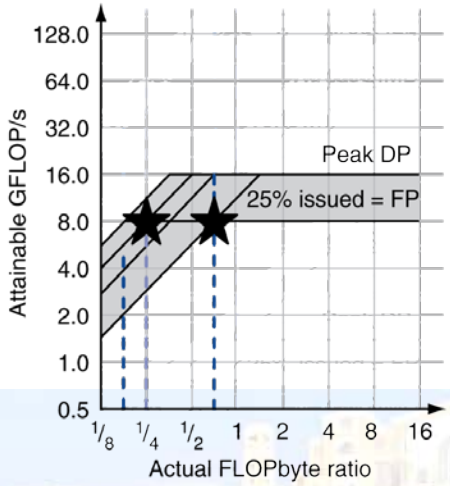
Roofline diagrams



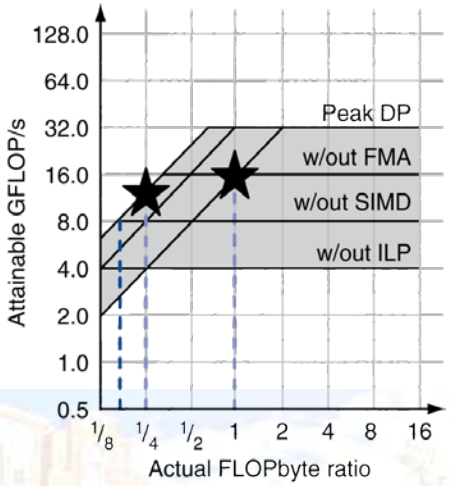
a. Intel Xeon e5345 (Clovertown)



b. AMD Opteron X4 2356 (Barcelona)



c. Sun UltraSPARC T2 5140 (Niagara 2)



d. IBM Cell QS20

Conclusion

- Goal: higher performance by using multiple processors
- Difficulties
  - Developing parallel software
  - Devising appropriate architectures
- Many reasons for optimism
  - Changing software and application environment
  - Chip-level multiprocessors with lower latency, higher bandwidth interconnect
- An ongoing challenge for computer architects!

Parallel programming

Can a program be parallelized?

- Matrix multiplication

```
for (int i = 0; i < M; ++i)
   for (int j = 0; j < N; ++j)
      for (int k = 0; k < K; ++k)
         C[i][j]  = A[i][k] * B[k][j];
```

- Fibonacci sequence

```
A = 0, B = 1
for (int i = 0; i < N; ++i) {
   C = A + B;
   A = B;
   B = C;
}
```

Parallel programming

- Software/algorithm is the key
- Significant performance improvement
  - Otherwise, Just use a faster uniprocessor
- Difficulties
  - Partitioning
  - Coordination
  - Communication overhead

Parallel programming

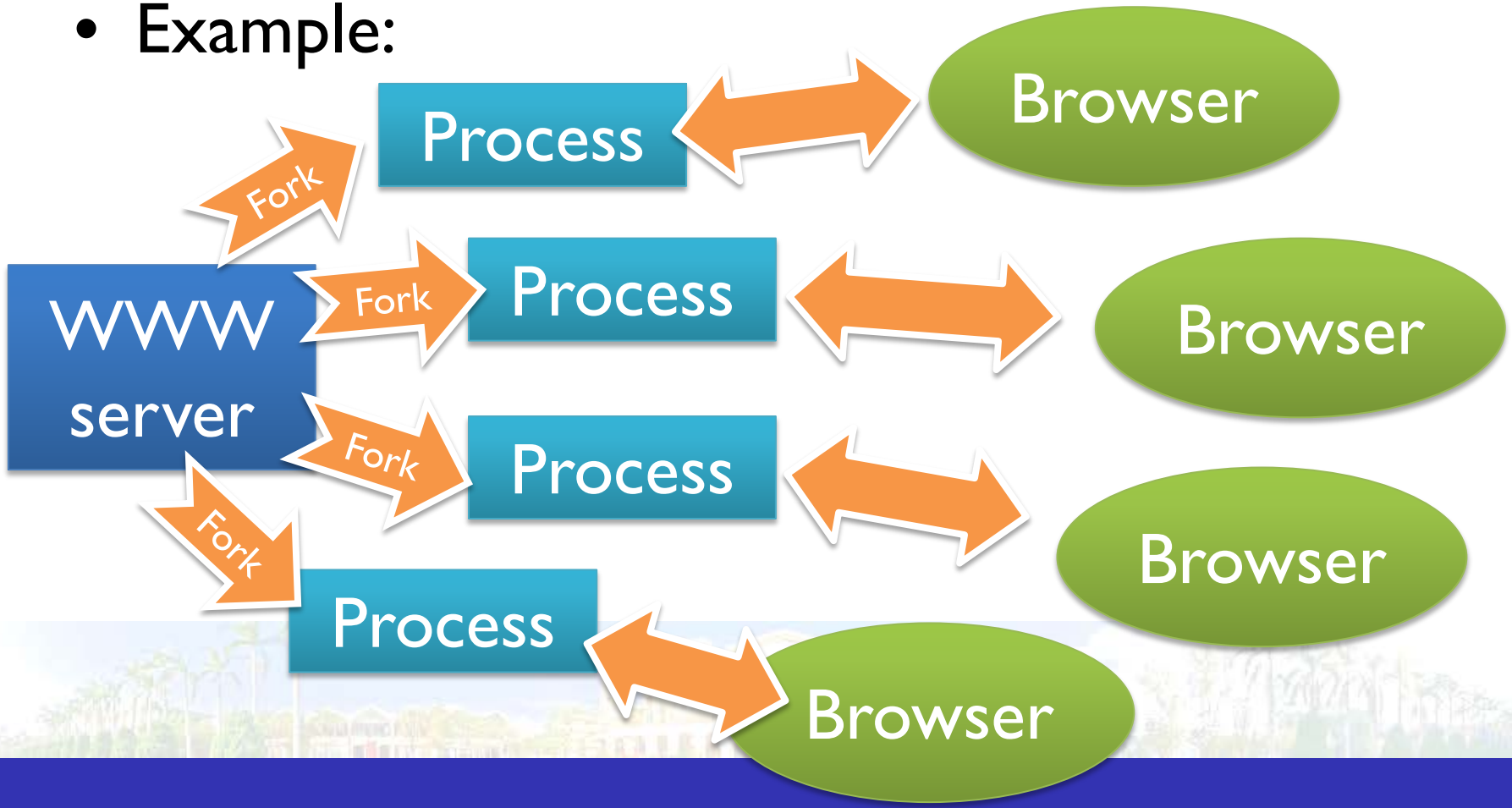
- Job-level parallelism
- Single program runs on multiple processors
- Single program runs on multiple computers

Job-level parallelism

- Operation system does it now
- How to improve the throughput?
  - Processors number vs. jobs number
  - Memory usage
  - I/O statistics
  - Scheduling and priority
  - ...

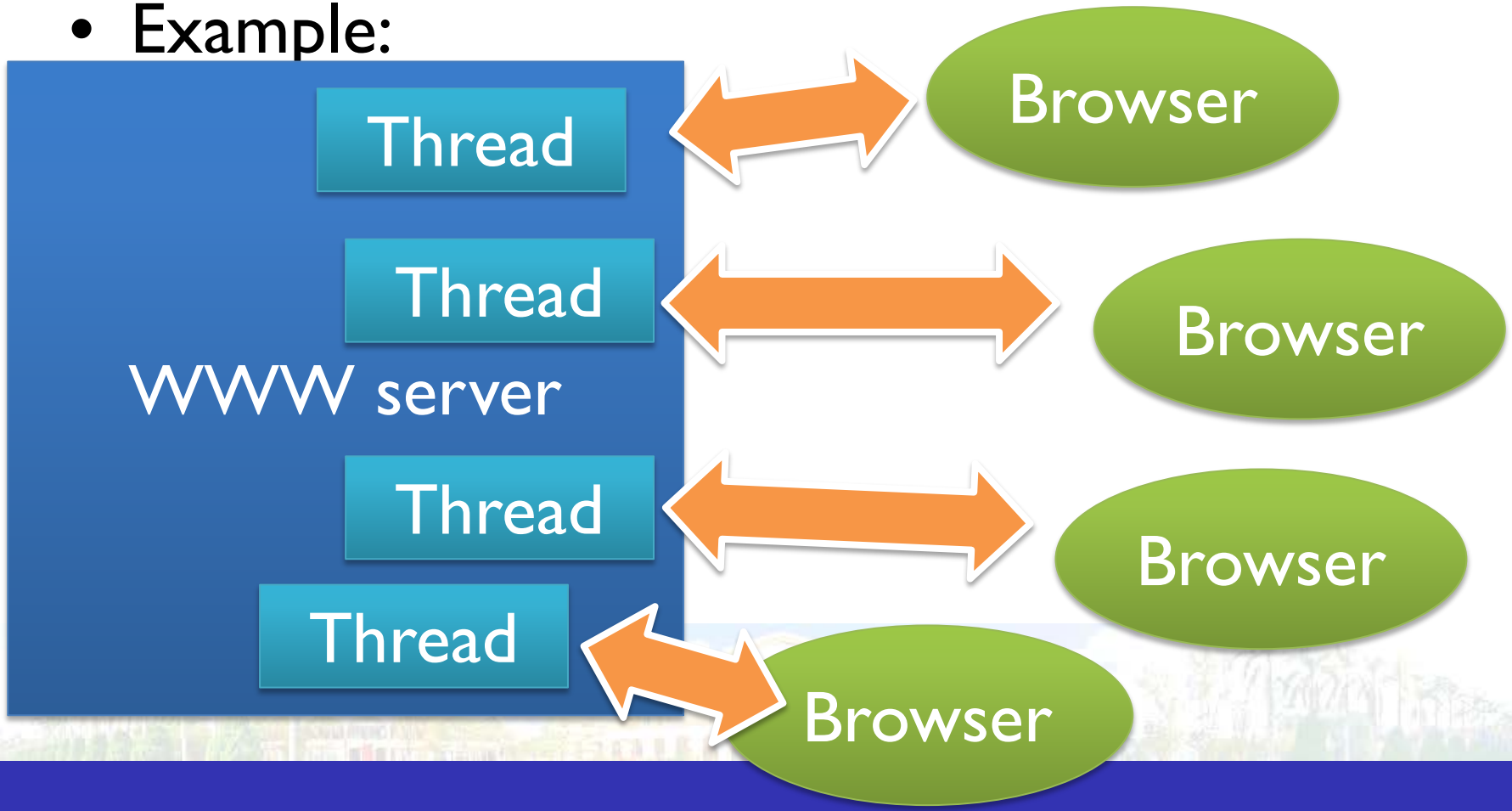
Multi-process program

- Process
  - a running instance of a program
- Example:



Multi-thread program

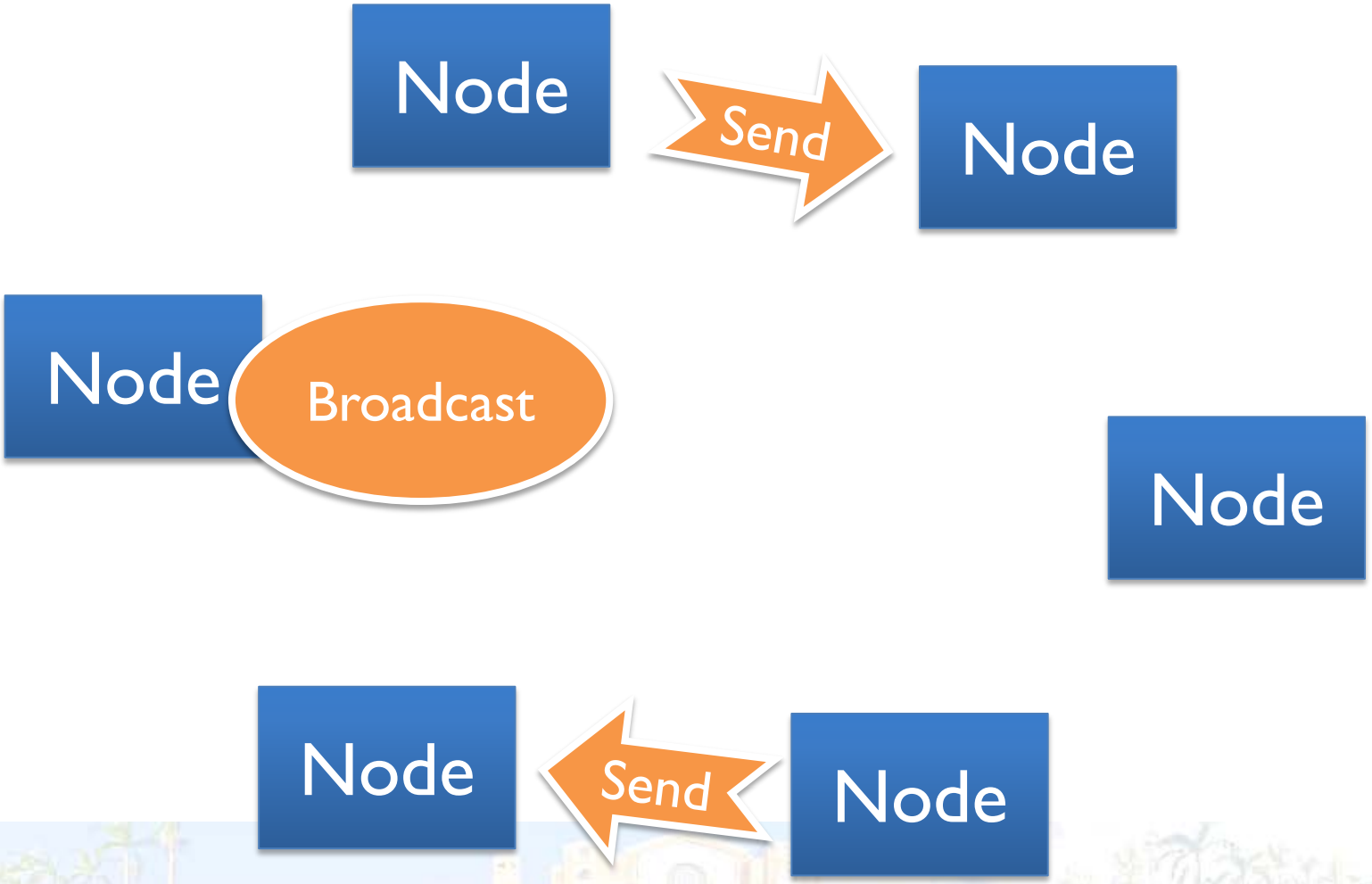
- Thread
  - Light weight process
- Example:



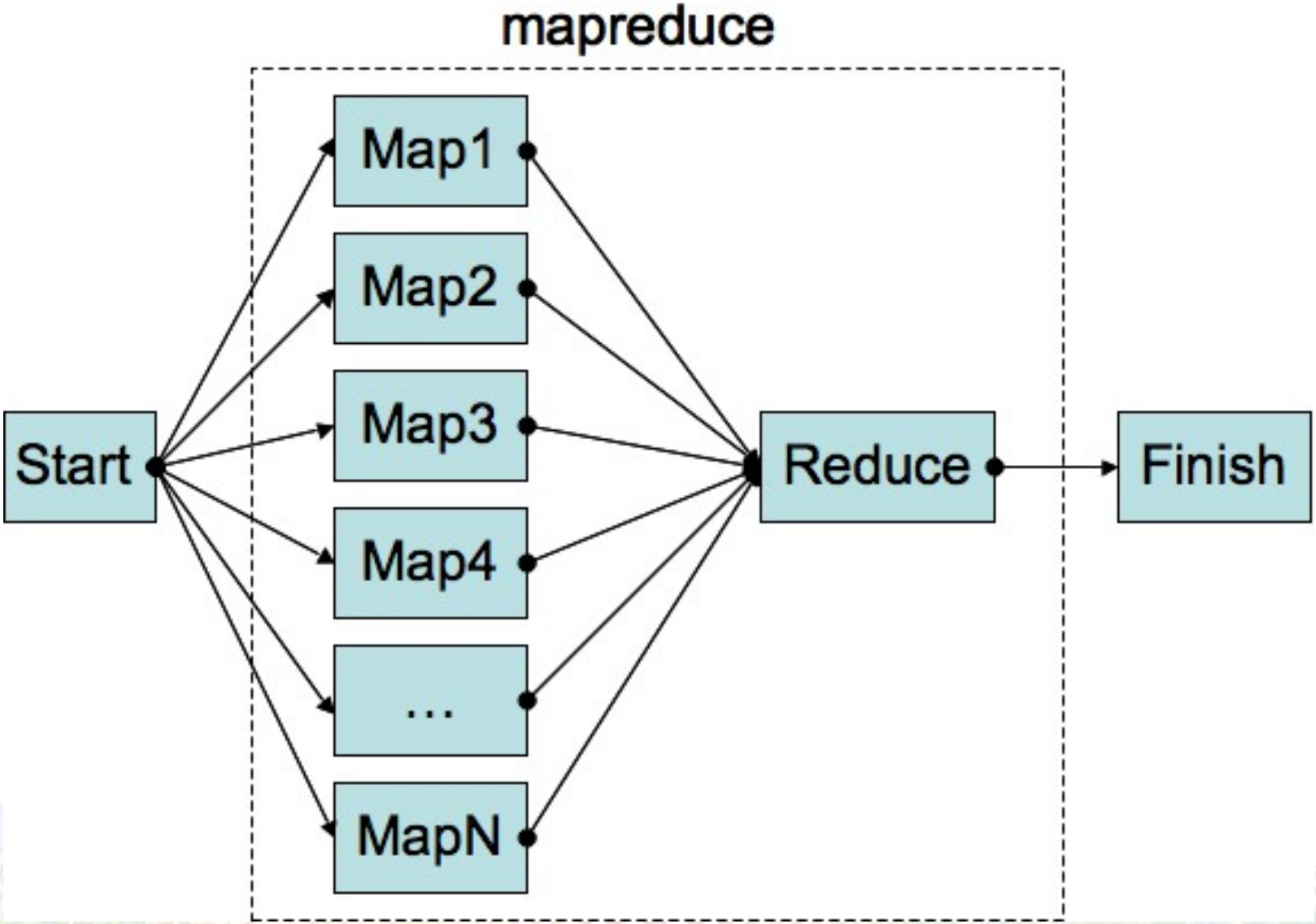
Multi-process vs. multi-thread

- Performance
  - Launch time
  - Context switch
  - Kernel-awareness scheduling
- Communication
  - Inter-process vs. inter-thread communication
- Stability

Message passing interface (MPI)

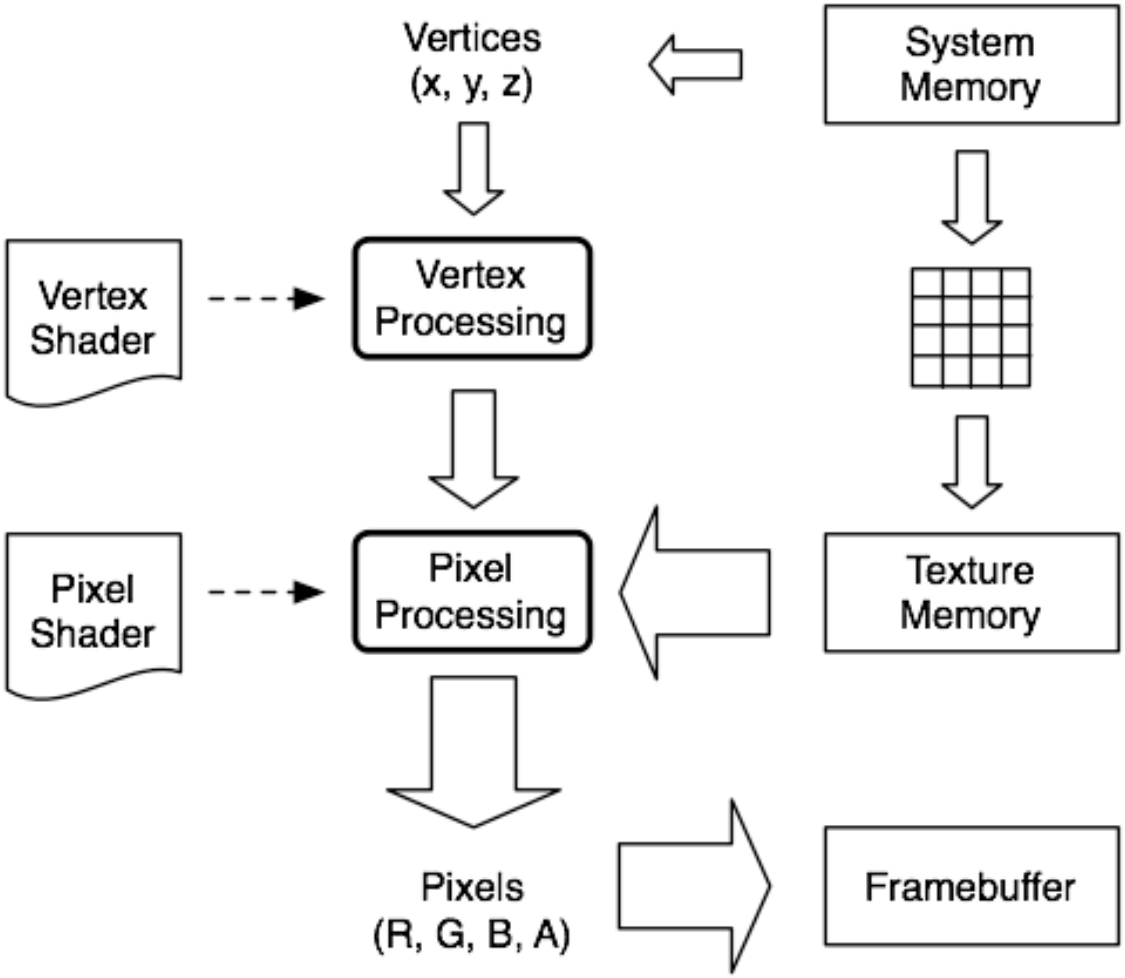


MapReduce/hadoop

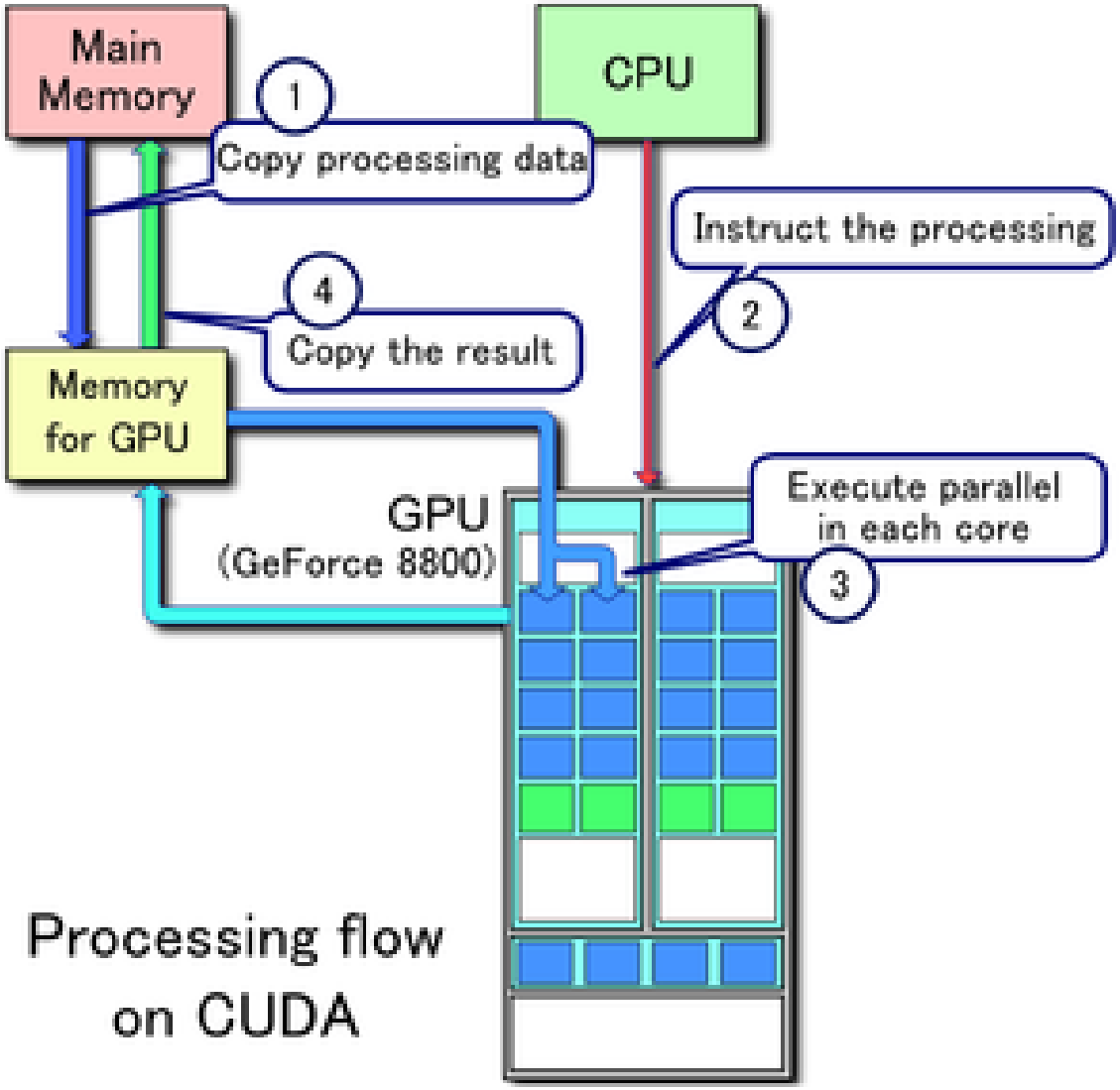


GPU programming

GPGPU



Cuda


Main Memory
CPU
1
Copy processing data
Instruct the processing
2
4
Copy the result
Memory for GPU
GPU
(GeForce 8800)
Execute parallel in each core
3
Processing flow on CUDA

Thanks!