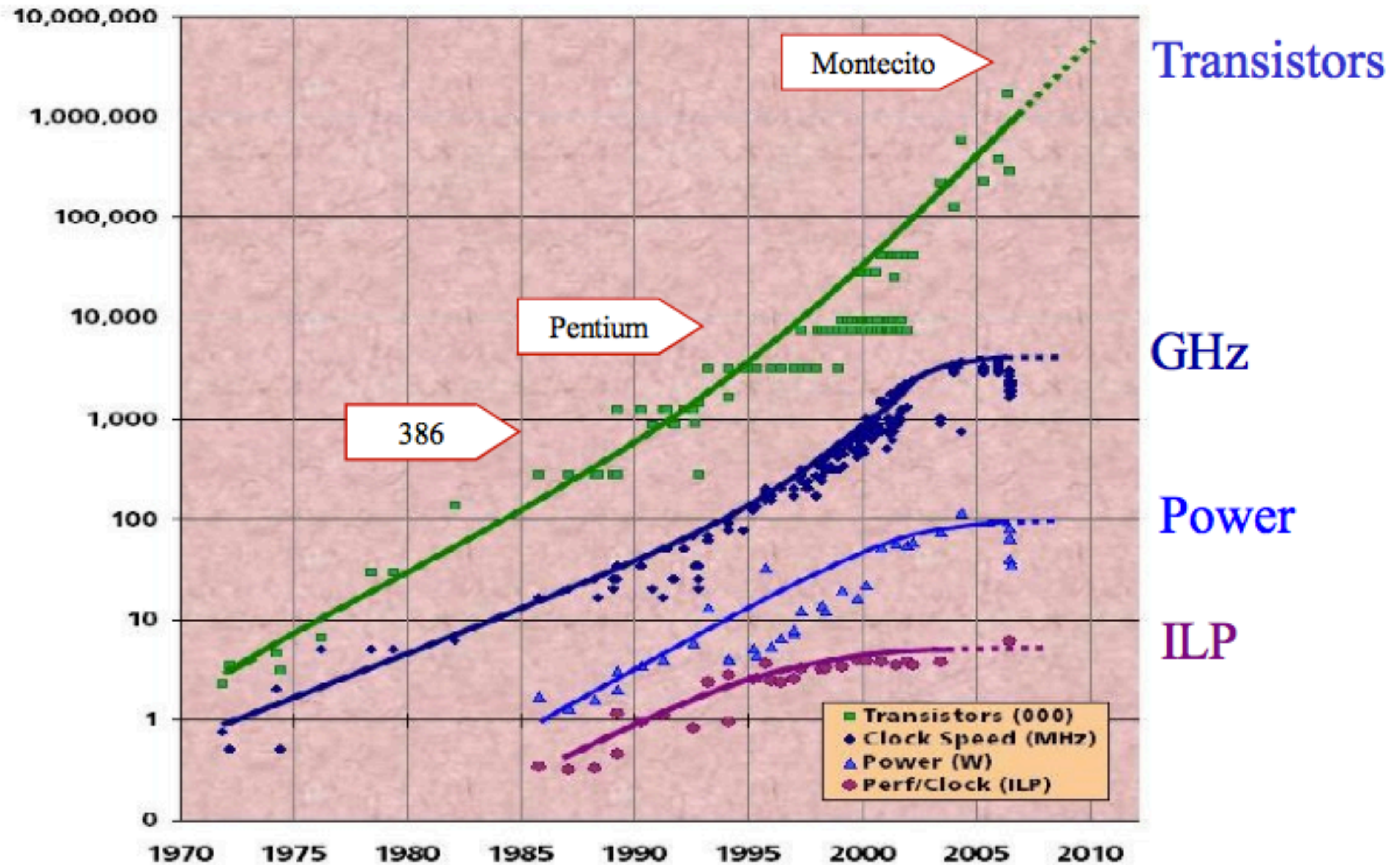
Parallelism, GPU Internals, CUDA

- Why is Parallel Programming Important
- Kinds of Parallelism
- SIMD – Parallel Programming Paradigm
- GPU Internals
- CUDA

Intel Microprocessor Trends



Why is parallel programming important?

Dateline: 1990

My program must run 2X faster.

I can:

- Work hard for several months – Assuming my program is already well-tuned
- Wait 18 months and buy a new computer

Dateline: 2010

My program must run 2X faster.

I can:

- Do nothing. My program may run slower.
- Work hard for several months.
- Or, I can rewrite my program in parallel.
    - Transistor density is still increasing
        - More parallel units on a chip
        - But speed of the units no longer increasing

1 Processor can have

Multiple cores, each can have

- Multiple ALUs, each run
  - One Thread at one time, each runs
    - One instruction at one time.

Parallelism vs. Concurrency

How will rewriting program in parallel help make program faster?

1. Power limits performance

– Cooling ability limits power density
– Determines battery life (mobile phone)
– Running cost and computing capacity of datacenter (Google)
⇒ Power limits performance i.e. speed of units/cores on processor, hence parallel execution on multiple cores make natural sense.

2. Smaller is faster ⇔ smaller is lower power

– Computation is cheaper than memory access
– Small caches have lower access times than large caches
– Cache access is faster than DRAM access
⇒ So it is better to have multiple small processors that execute multiple small programs in parallel than a large processor that executes one large program sequentially.

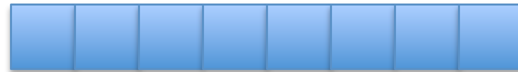
Kinds of Parallelism?

1. **Data Parallelism** (loop level parallelism) = SIMD
   - focuses on distributing the data across different ALUs executing in parallel.

2. **Task Parallelism** (function/control parallelism)
   - focuses on distributing execution processes (threads) across different cores executing in parallel.
   - achieved when each processor core executes a different thread on the same or different data.
   - The threads may execute the same or different code/instruction.

SIMD – Single Instruction Multiple Data

Say you have an array of length N ->



0 1 2 3 4 5 6 7

And lets say you have the following piece of code ->

```
for ( i = 0 to 7 )
      A[i] = A[i] + 1;
```

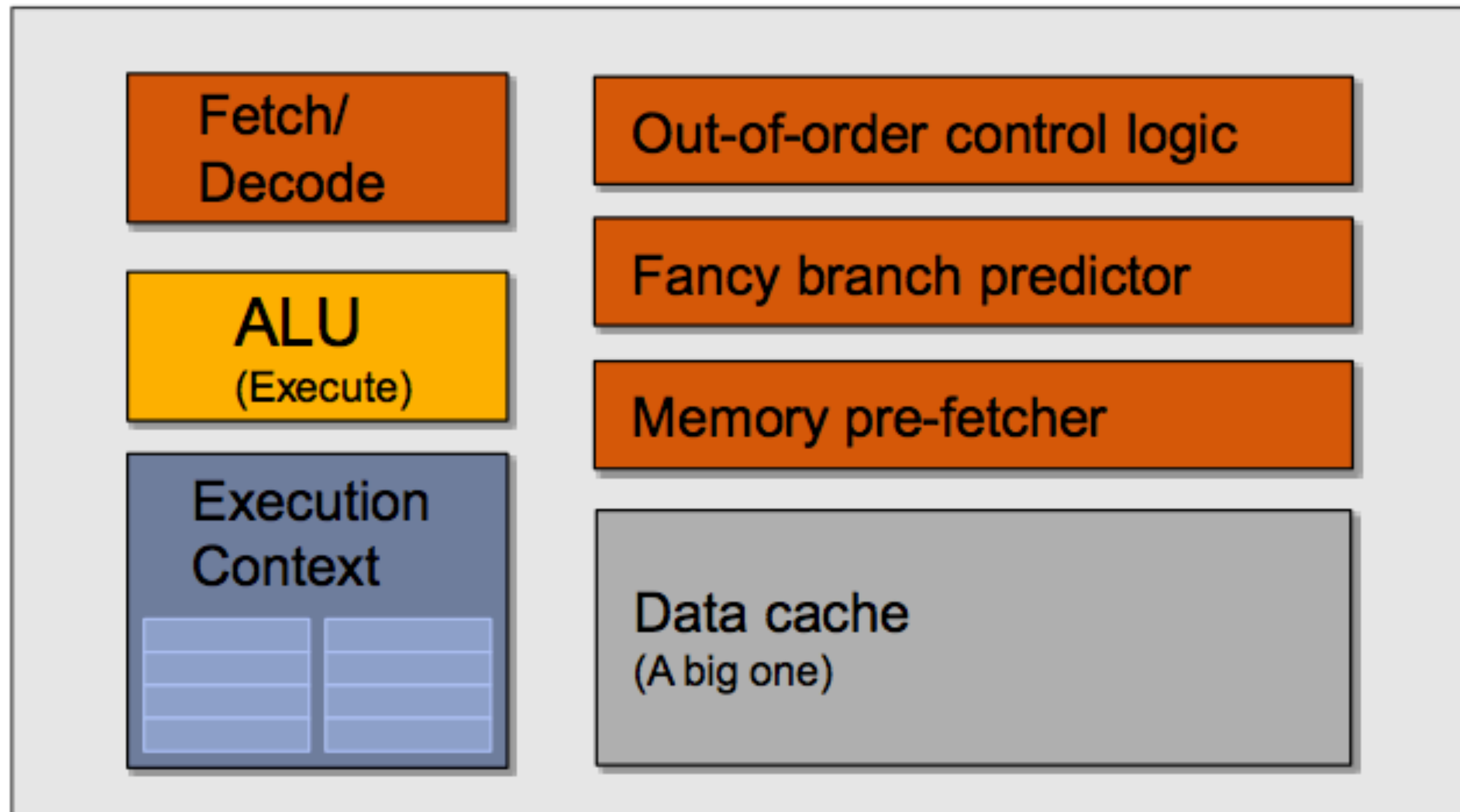
"A[i] = A[i] + 1" is one instruction that needs to run on 8 pieces of data-

```
A[0] = A[0] + 1;
A[1] = A[1] + 1;
A[2] = A[2] + 1;
A[3] = A[3] + 1;
A[4] = A[4] + 1;
A[5] = A[5] + 1;
A[6] = A[6] + 1;
A[7] = A[7] + 1;
```

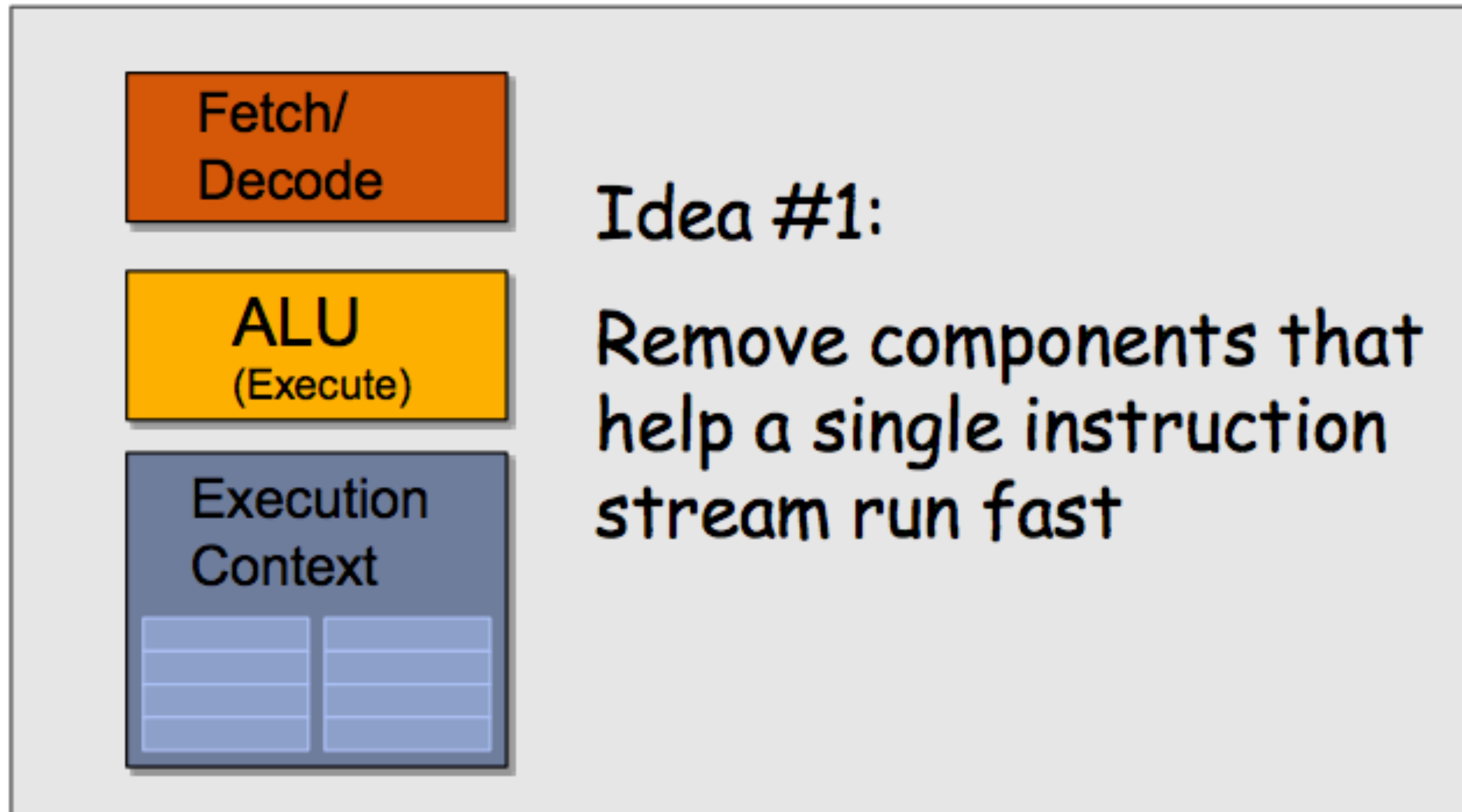
On a SIMD platform (e.g. GPU), 1 processor-core will contain 8 ALUs that can run 1 thread each, and hence execute all 8 statements in parallel in one time step.

GPU Internals

CPU-"style" cores



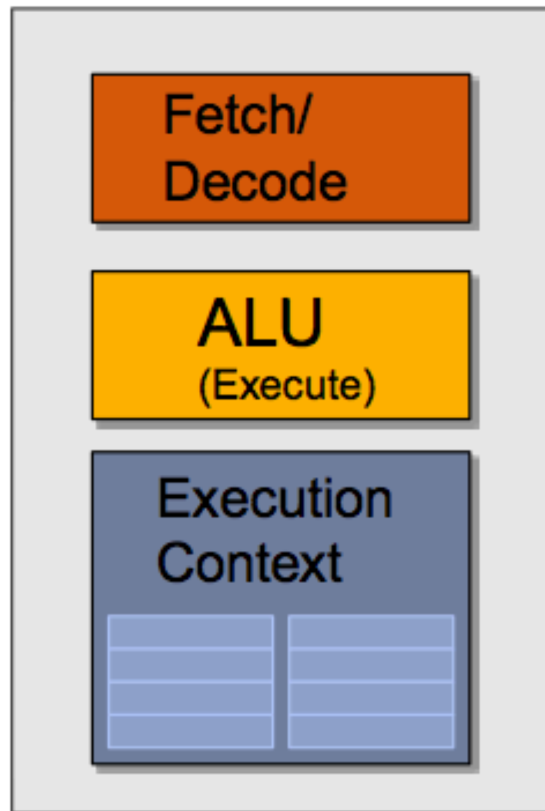
Slimming down

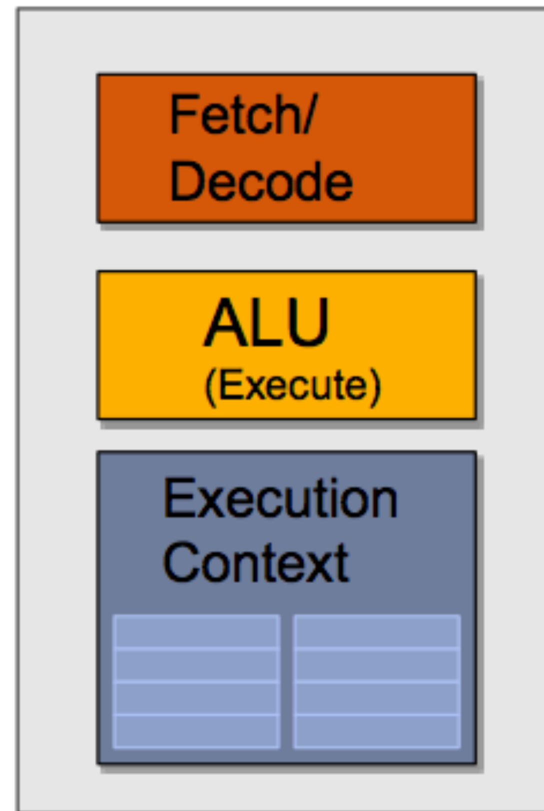


Two cores (two threads in parallel)

Thread 1

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

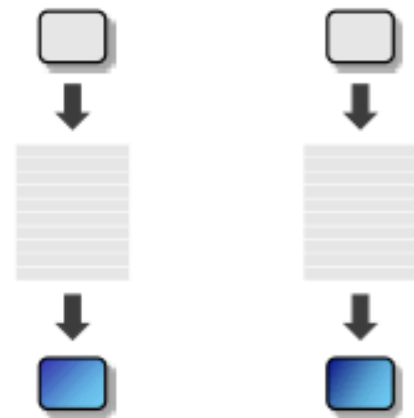


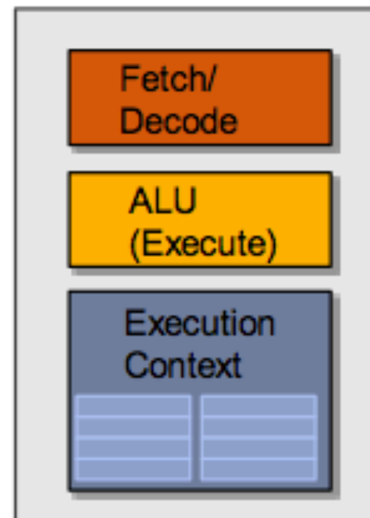


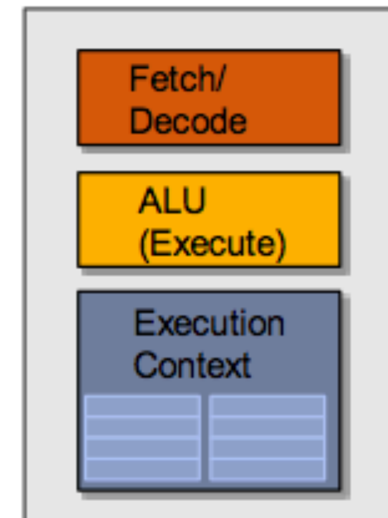
Thread 2

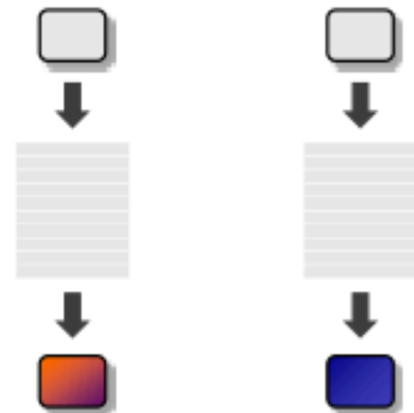
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

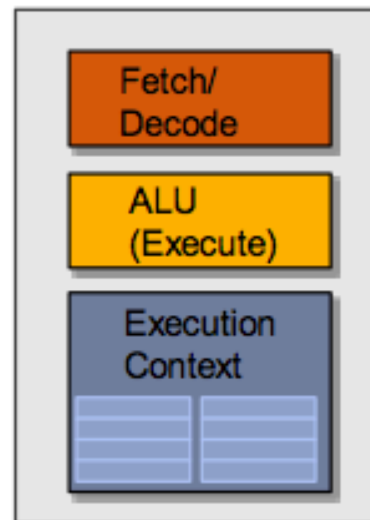
Four cores (four threads in parallel)

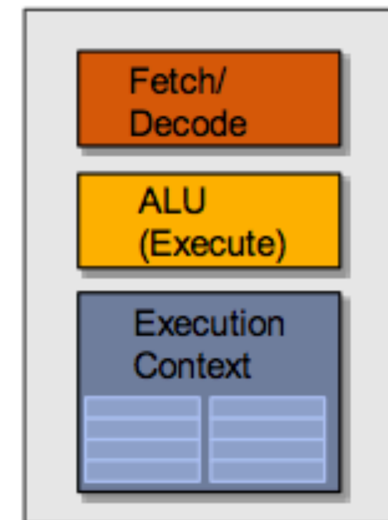




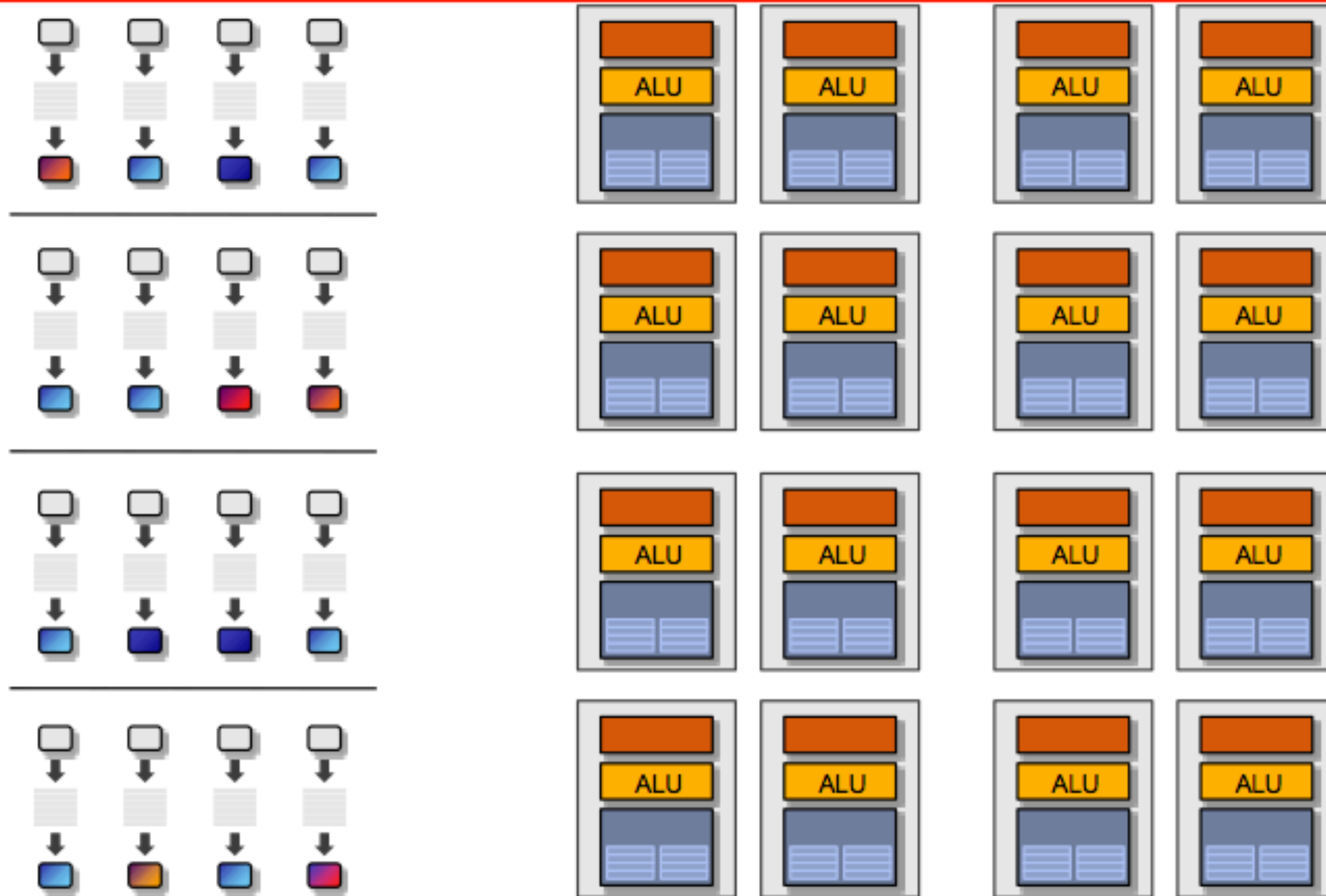








Sixteen cores (sixteen threads in parallel)



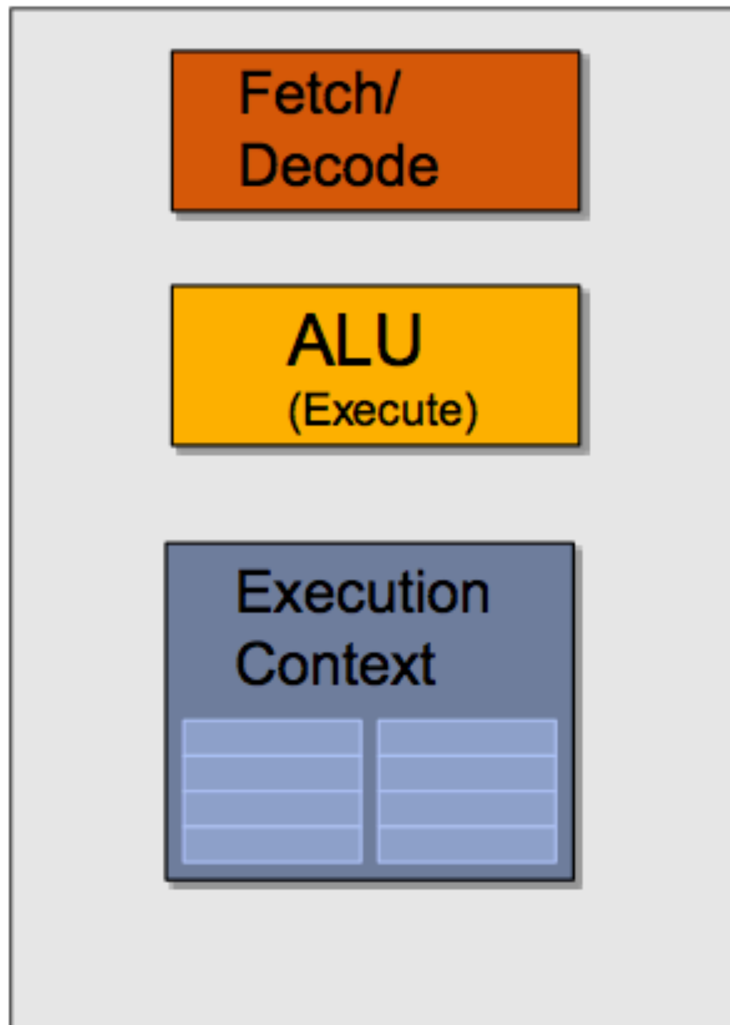
16 cores = 16 simultaneous instruction streams

Instruction stream sharing

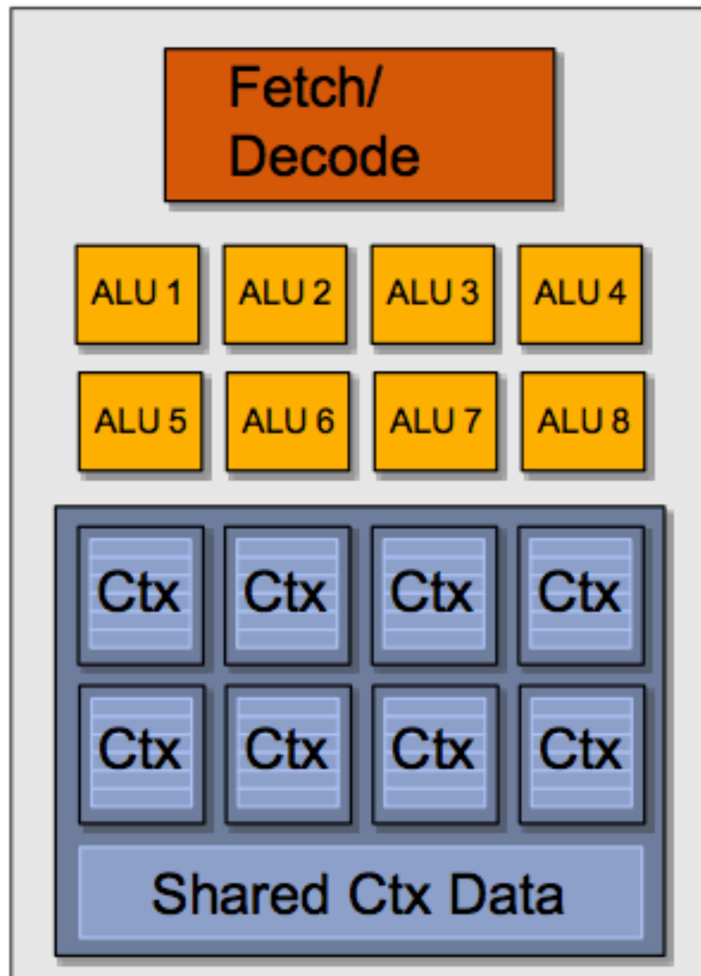
But… many threads should be able to share an instruction stream!

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Recall: simple processing core



Add ALUs

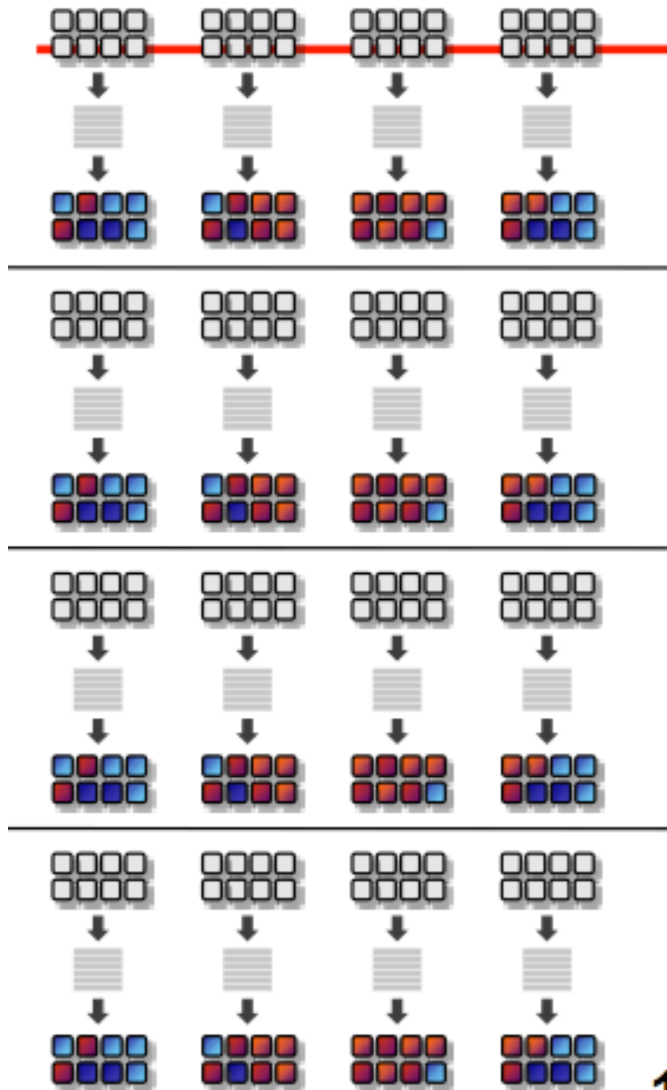


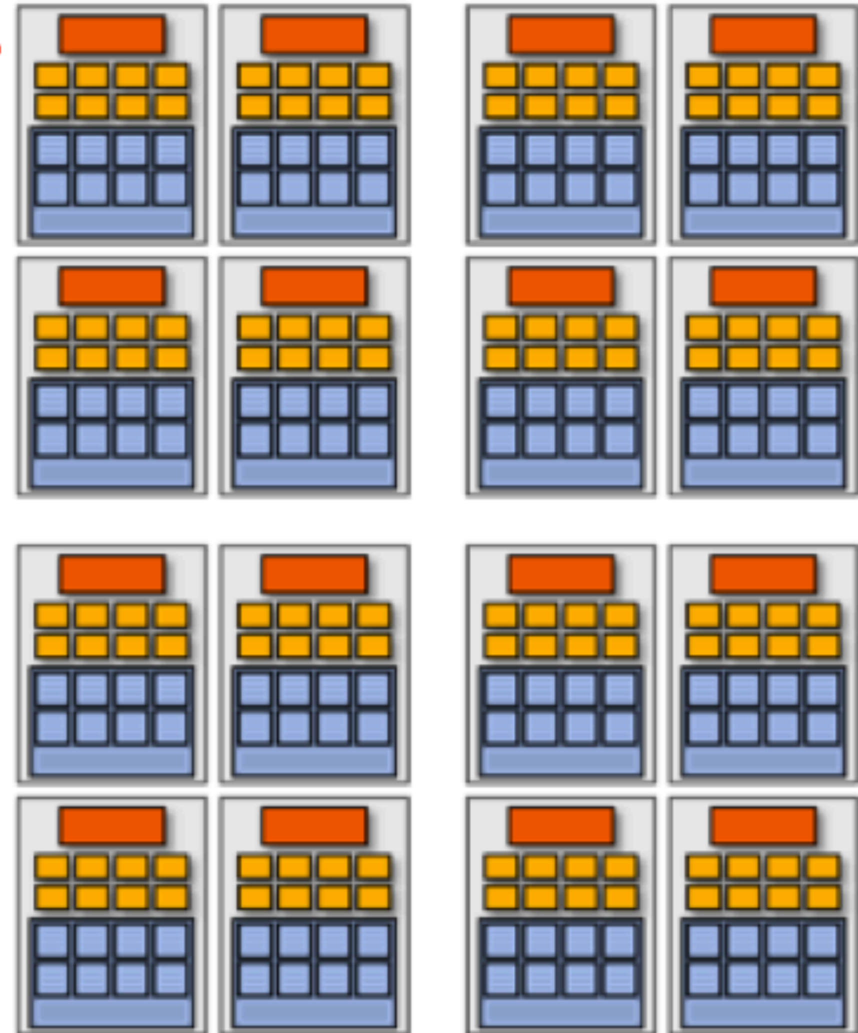
Idea #2:

Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD or Vector processing

128 threads in parallel



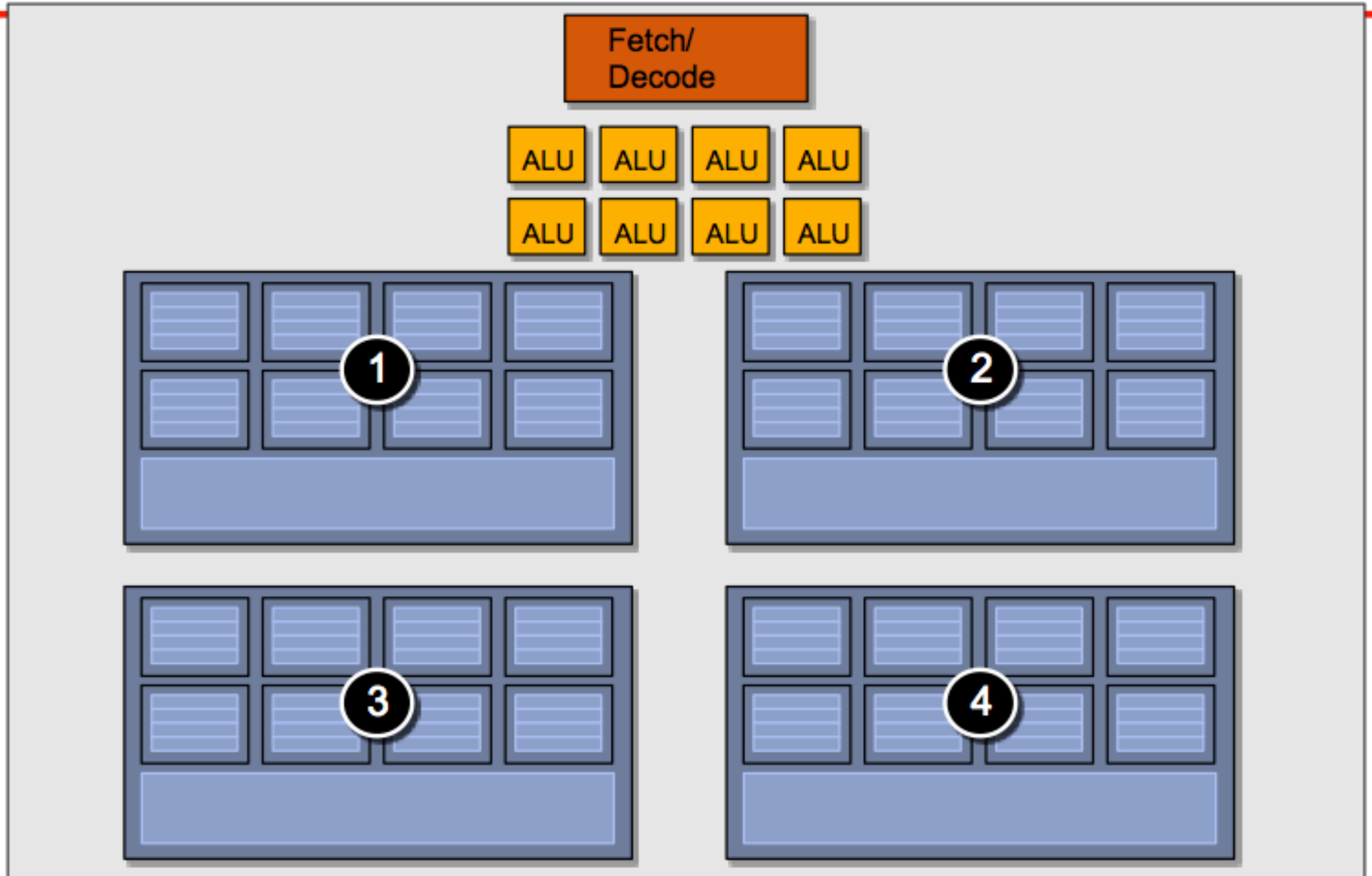


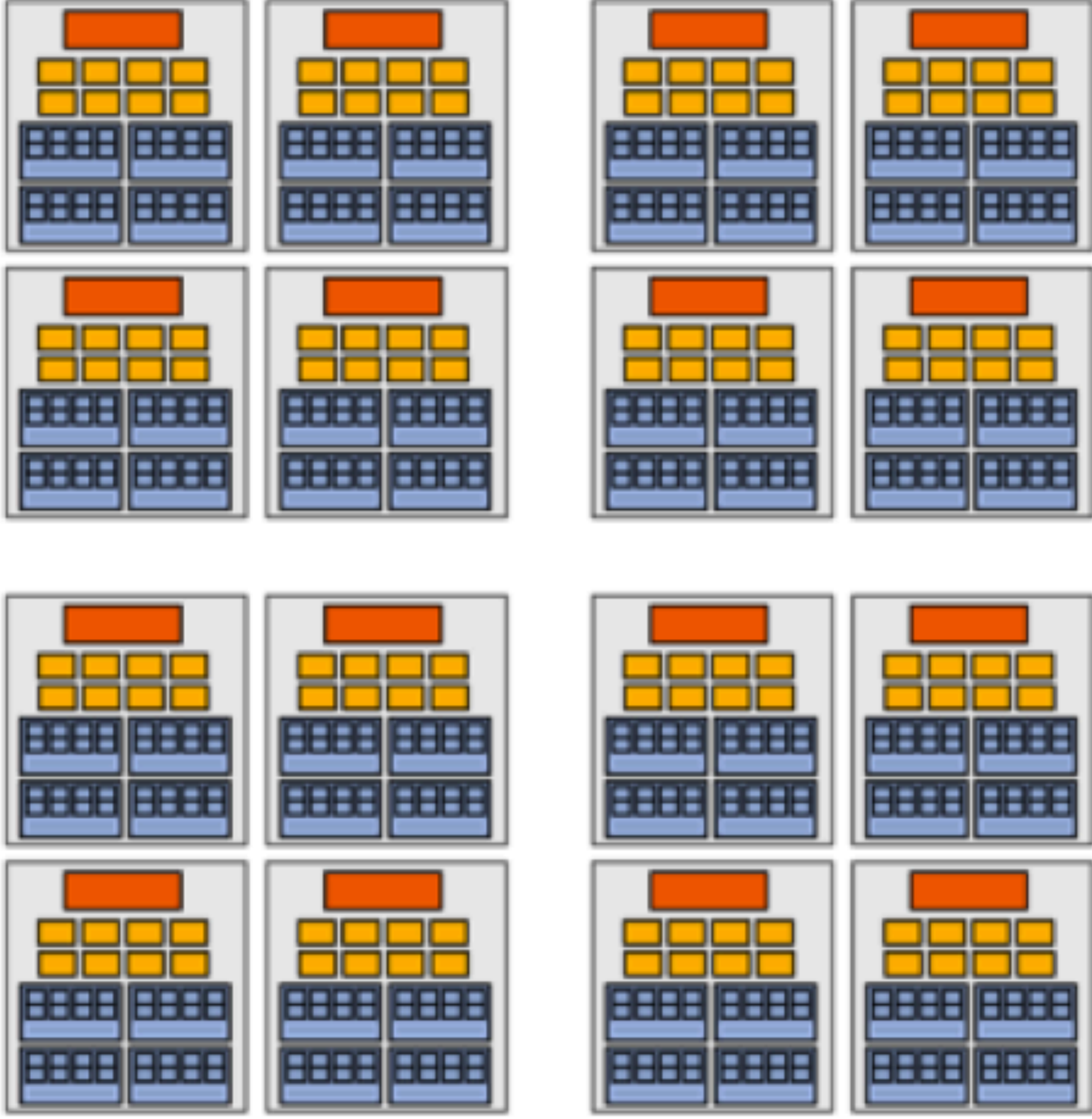
16 cores = 128 ALUs

= 16 simultaneous instruction streams

Four large contexts

(low latency hiding ability)





NVIDIA GeForce GTX 285

- Generic speak:
  - 30 cores
  - 8 SIMD functional units per core
  - 768 threads/core ⇒ 23, 040 threads/chip

- NVIDIA-speak:
  - 240 stream processors
  - “SIMT execution”

Summary: three key ideas

1. Use many "slimmed down cores" to run in parallel

2. Pack cores full of ALUs (by sharing instruction stream across groups of threads)
   - Option 1: Explicit SIMD vector instructions
   - Option 2: Implicit sharing managed by hardware

3. Avoid latency stalls by interleaving execution of many groups of threads
   - When one group stalls, work on another group

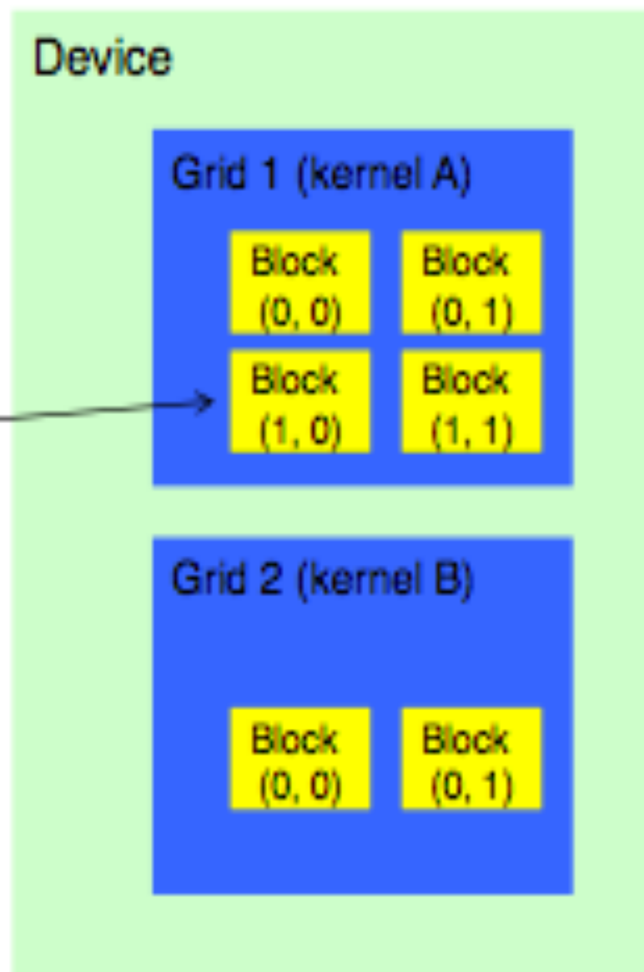
CUDA

CUDA Kernels

- Kernels are executed by an array of threads
- All threads run the same code (SPMD)
  - Single program multiple data
- Each thread has an ID
  - Compute memory addresses
  - Make branch decisions

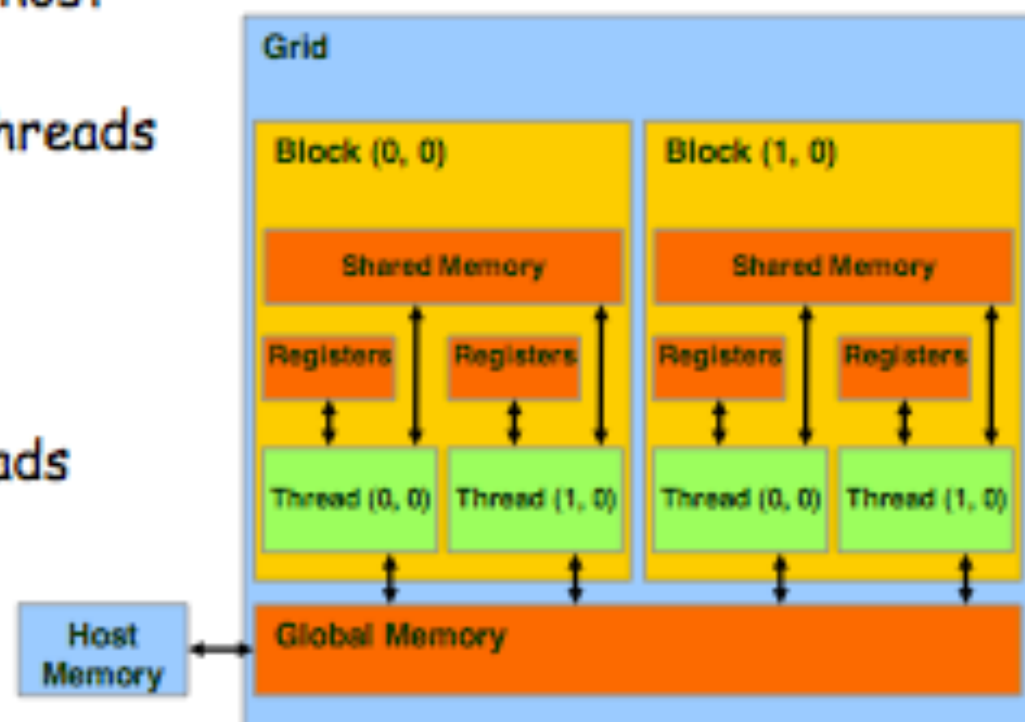
Grids, Blocks and Threads

- Each kernel runs as grid of threads on whole device
- Grids are composed of blocks
  - Block ID: 1D or 2D
  - Blocks are executed in any order
- Blocks are composed of threads
  - Thread ID: 1D, 2D, or 3D
  - Max 512 threads per block



CUDA Memory Model

- Global memory
  - Data transfer between host and device
  - Contents visible to all threads
  - Long latency access
- Shared Memory
  - Only accessible to threads within a block
    - Atomics
    - Barriers
  - user managed
    - Not a cache



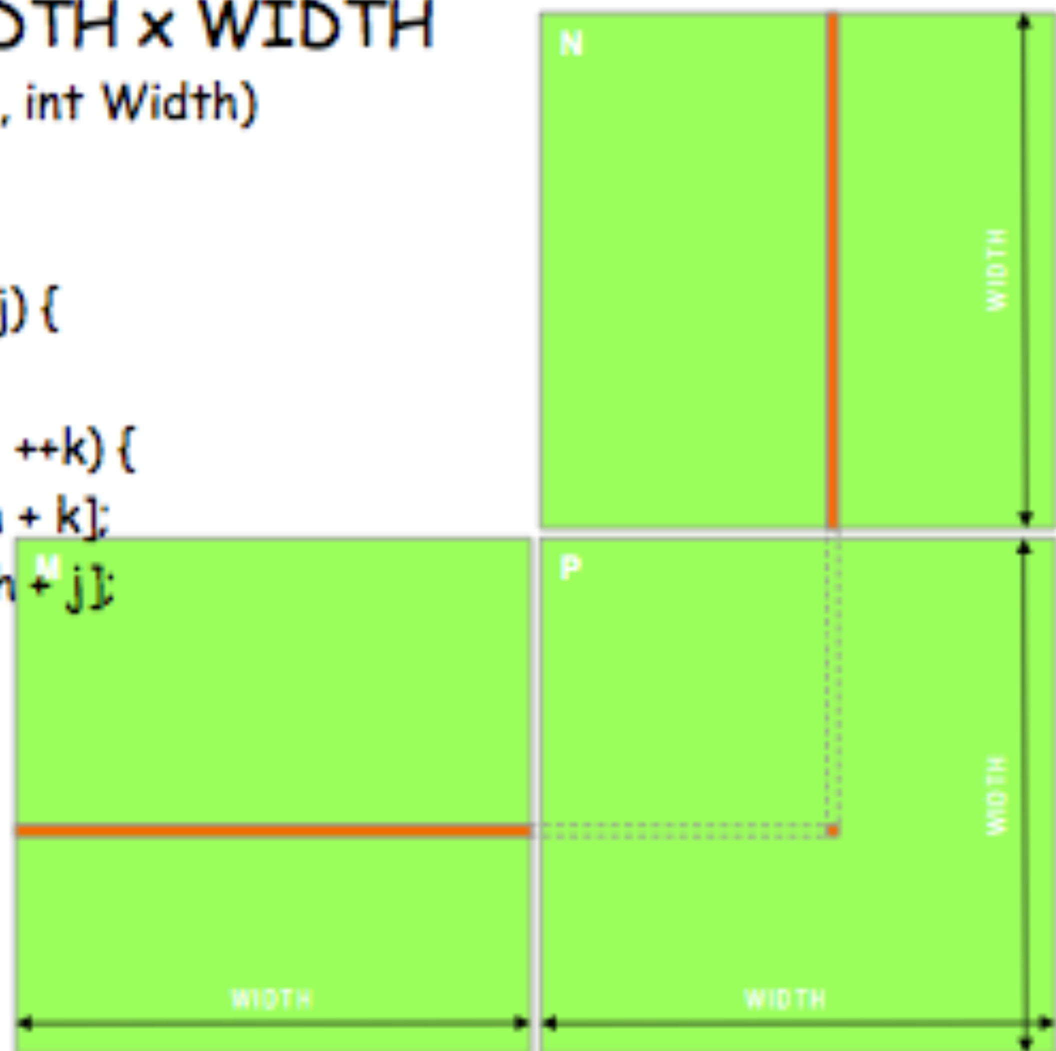
Canonical CUDA Program

1. Allocate device global memory
2. Transfer input data from host to device
3. Invoke kernel (s)
4. Transfer output data from device to host
5. Deallocate device global memory

Matrix Multiplication Example

- P = M * N of size WIDTH x WIDTH

```
MM(float* M, float* N, float* P, int Width)
{
  for (int i = 0; i < Width; ++i)
    for (int j = 0; j < Width; ++j) {
      double sum = 0;
      for (int k = 0; k < Width; ++k) {
        double a = M[i * width + k];
        double b = N[k * width + j];
        sum += a * b;
      }
      P[i * Width + j] = sum;
    }
}
```

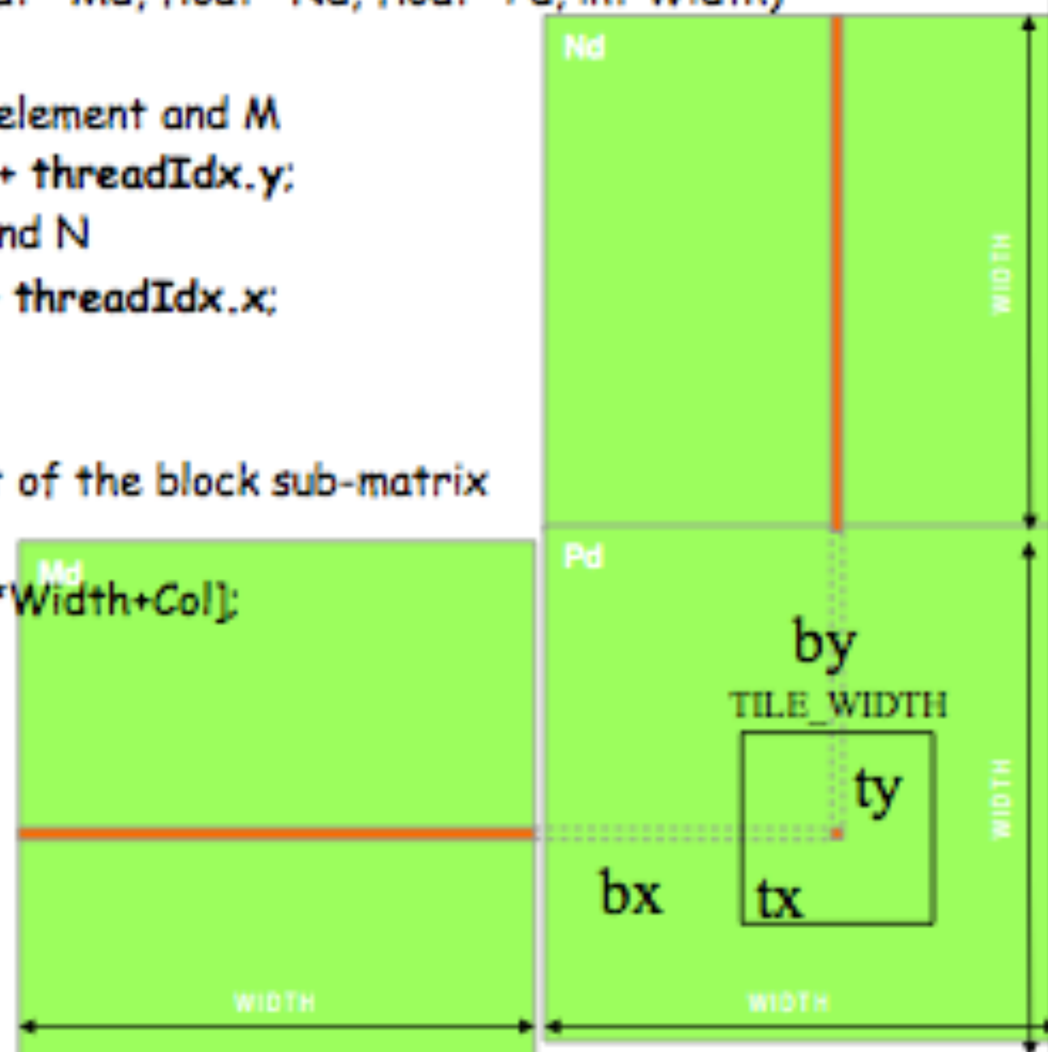


Allocate and Transfer Data

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;
  ...
// 1. Allocate M, N, P to device global memory
    cudaMalloc(&Md, size);
    cudaMalloc(&Nd, size);
    cudaMalloc(&Pd, size);

// 2. Load M and N
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
```

Matrix Multiply Kernel

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column index of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

float Pvalue = 0;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

Pd[Row*Width+Col] = Pvalue;
}
```



Kernel Invocation, Data Transfer, Deallocation

```
    // Setup the execution configuration
    dim3 dimGrid(Width  / TILE_WIDTH, Width /  TILE_WIDTH);
    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);

    // 3.  Launch the device computation threads!
   MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);

    // 4.  Read P from the device
     cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

      // 5. Free device matrices
     cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
}
```