

# TBB : Pros and cons

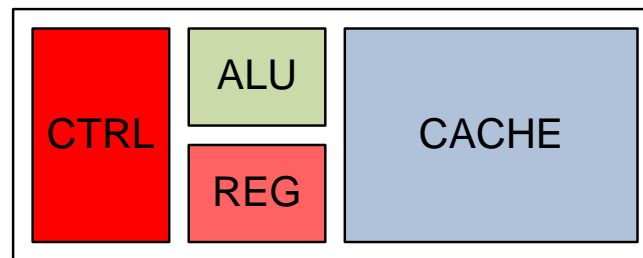
- ✓ Very flexible in terms of parallelism, with multiple patterns
  - Data-parallel, pipelined, fork-join, arbitrary task graphs
- ✓ Makes it possible to exploit many cores with little effort
  - Quite easy to scale up to 64+ cores
  - Can get close to linear speed-up
- ✓ Integrates very well with existing codebases
  
- ✗ Runs on quite heavy-weight multi-core CPUs
  - All the power and area wastage of super-scalar processors
- ✗ Have to be careful to **agglomerate** parallel tasks
  - Very small tasks have a lot of overhead
  - Must agglomerate potentially parallel tasks into a single serial task

# Reducing the Cost of Work

- What limits the execution rate in a “normal” CPU?
  - Communication: time taken to read/write data to memory
  - ALU Speed: speed with which calculations can be performed
  - Clock Rate: how often can instructions be issued
- Technology scaling: increase clock rate / increase area
  - Clock scaling is very limited; too much power consumption
  - But we have increasing amounts of area
  - What to do with it?
- Limitation has become instruction issue rate
  - *How can we execute  $n$  operations in fewer than  $n$  cycles?*

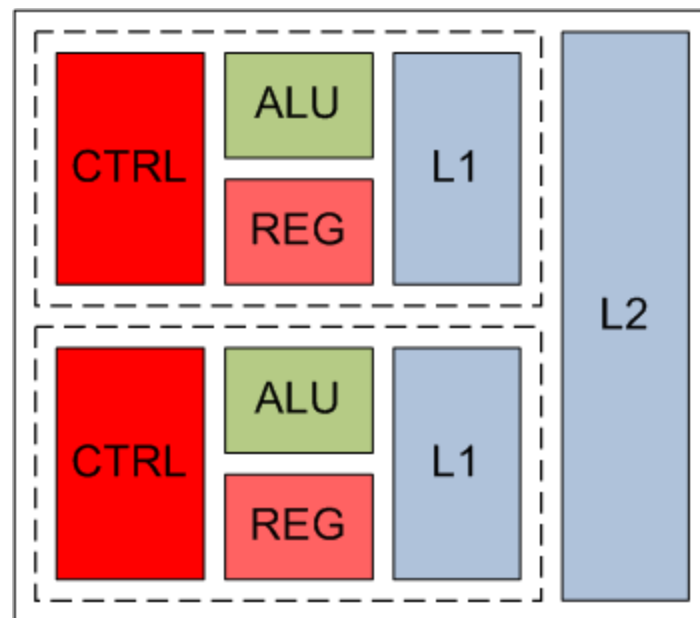
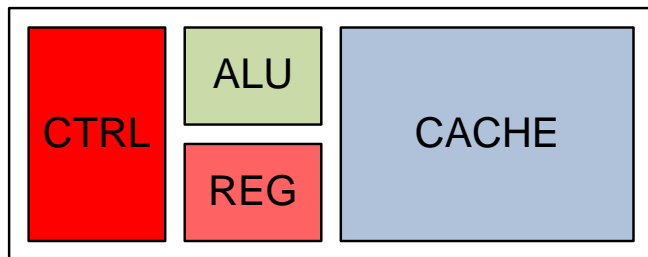
# Using Area to Increase Instructions/Sec

- Assume current technology is able to support a basic CPU
  - Control logic: power consuming; almost pure overhead
  - ALU: uses power, but gets useful work done
  - Registers: uses some power, but not too bad
  - Cache: consumes a lot of area, but less power
- What do we do with twice the area?



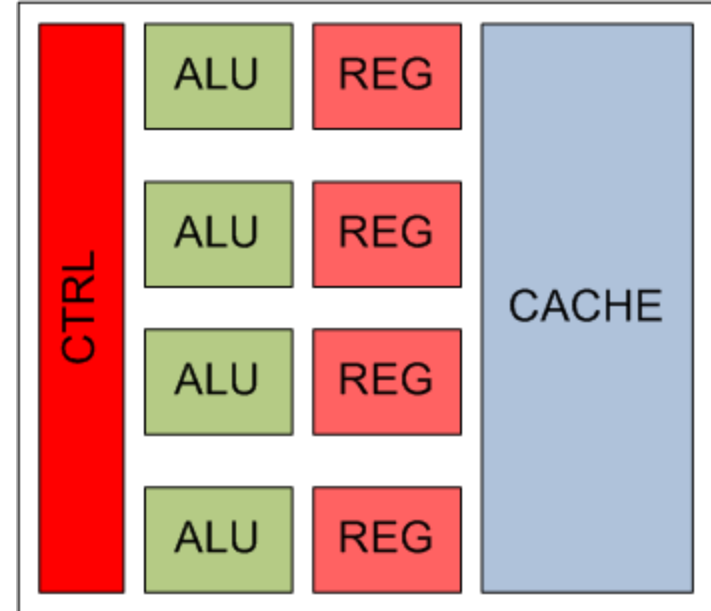
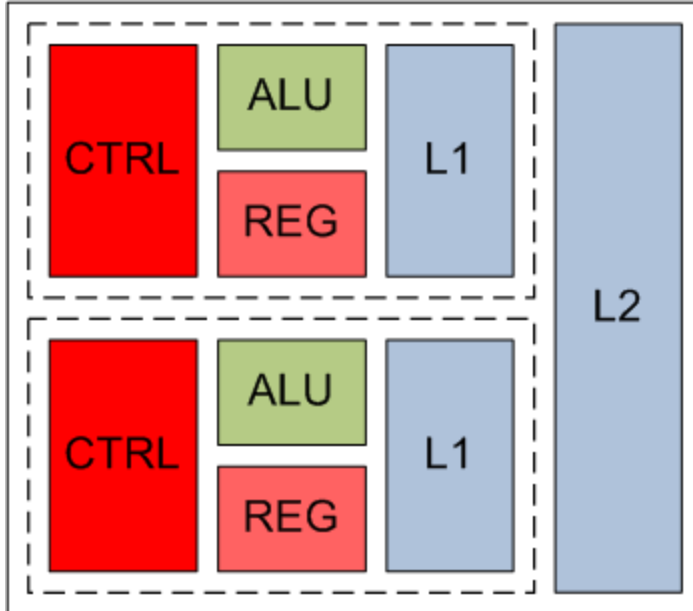
# Using Area to Increase Instructions/Sec

- Default answer: twice the area, let's create two CPUs!
  - Some advantages: two CPUs share larger L2 cache
- Efficiency is about the same
  - Either Operations/Cycle/Area or Operations/Joule



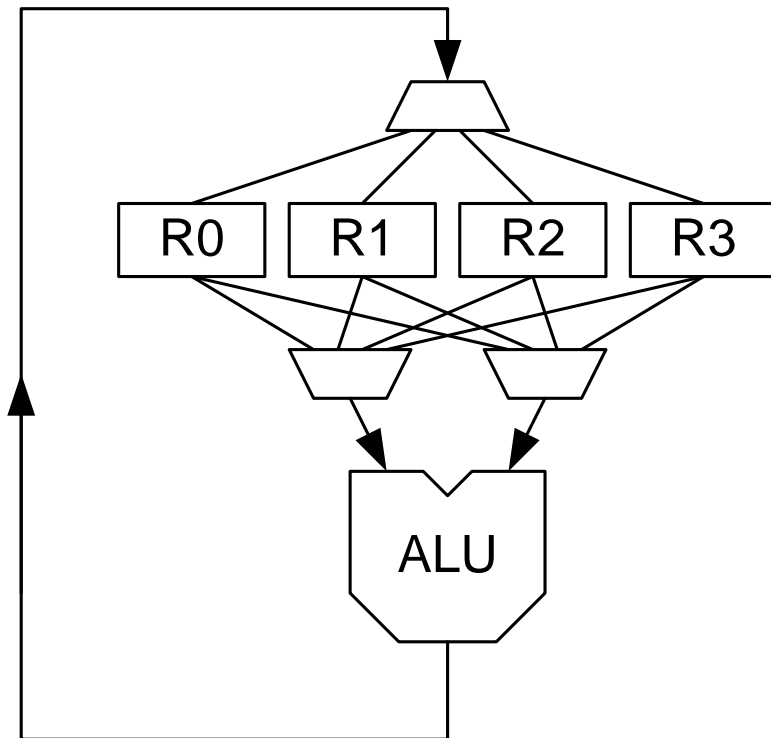
# SIMD and VLIW

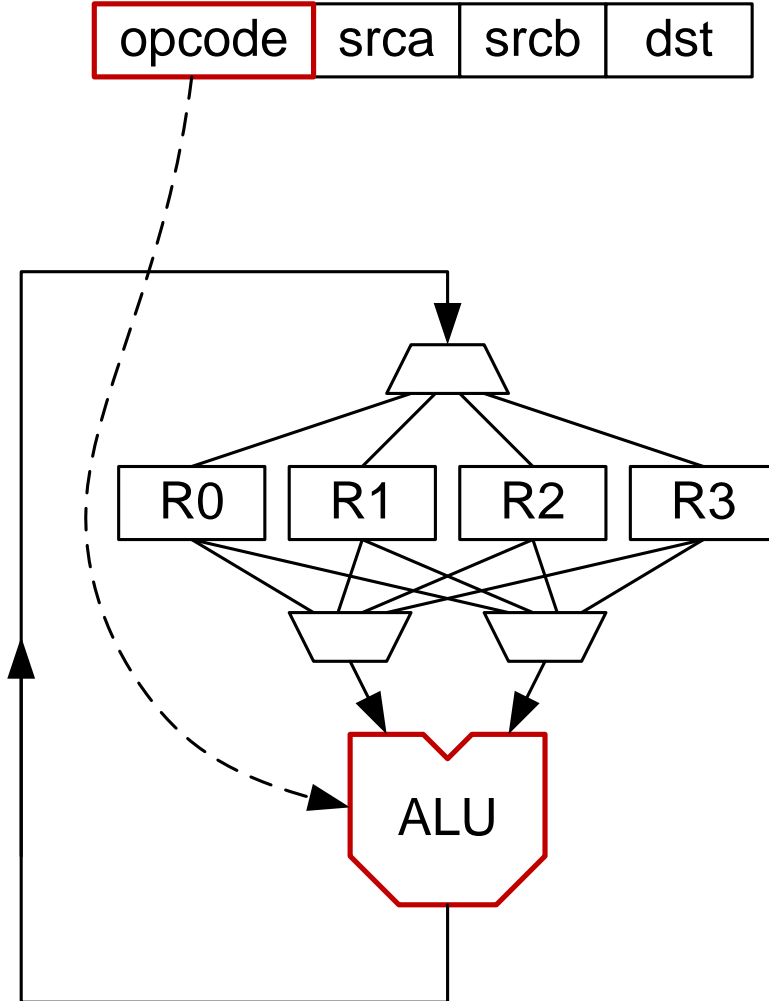
- Extend processor with more complex instructions
  - Each instruction issued can perform multiple operations
- Proportion of ALU (good) to CTRL (bad) is increased
  - Can we ensure ALUs will actually be occupied?



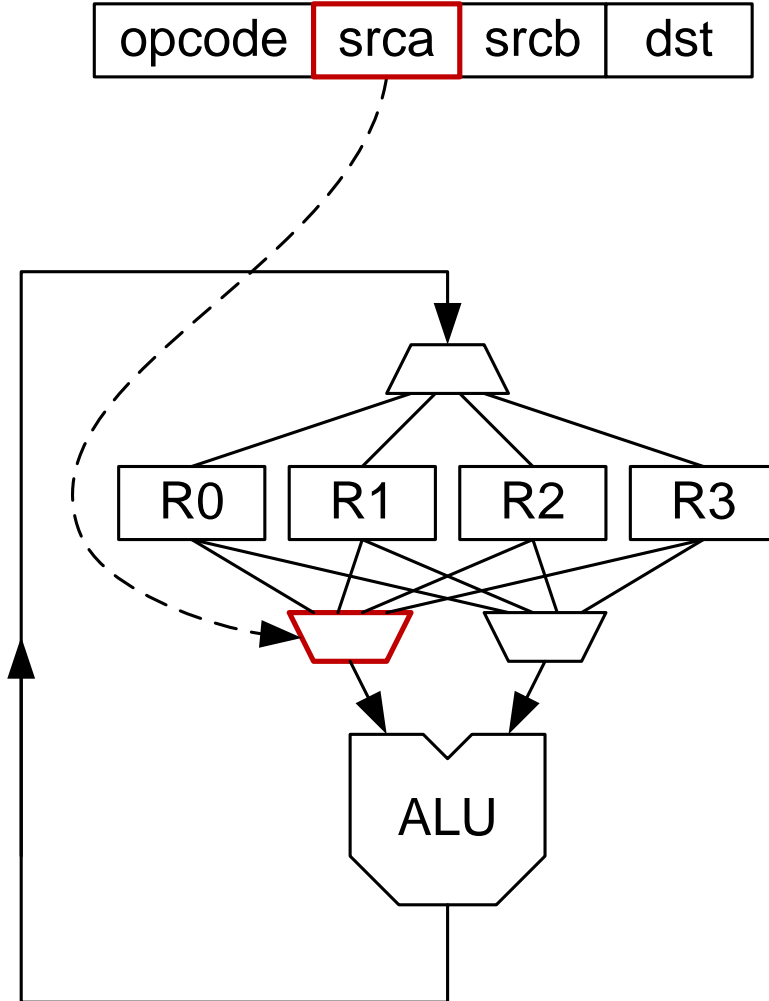


- Classic scalar CPU
  - One set of registers
  - Single ALU



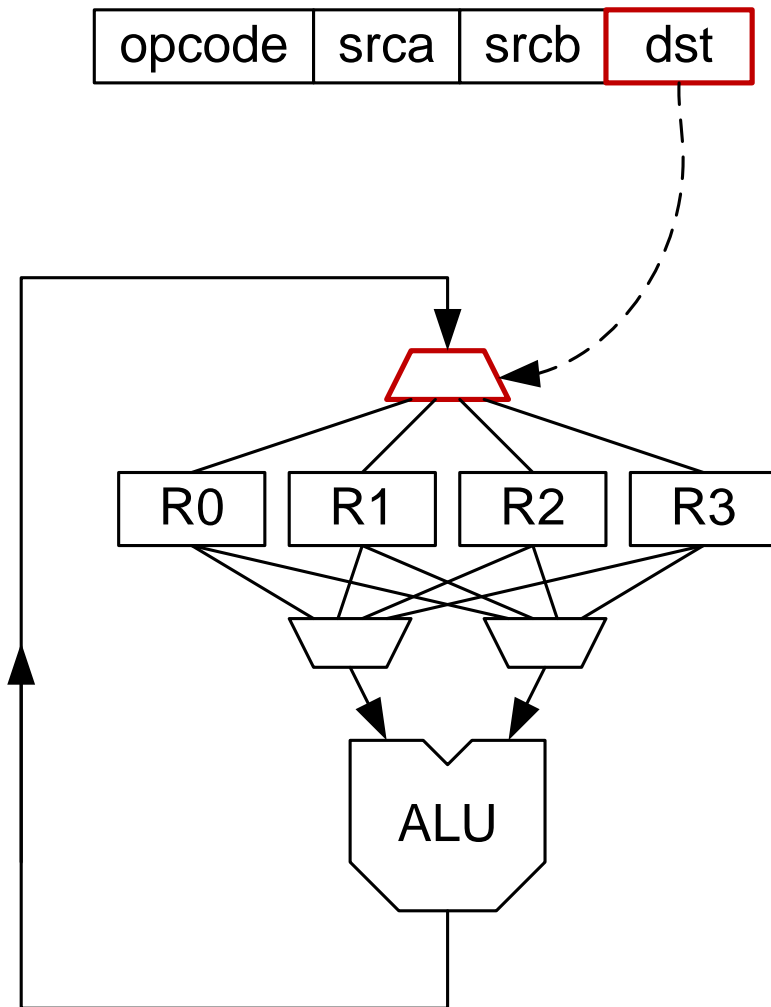


- Classic scalar CPU
  - One set of registers
  - Single ALU
- Instruction controls data-flow
  - Opcode: what should ALU do?



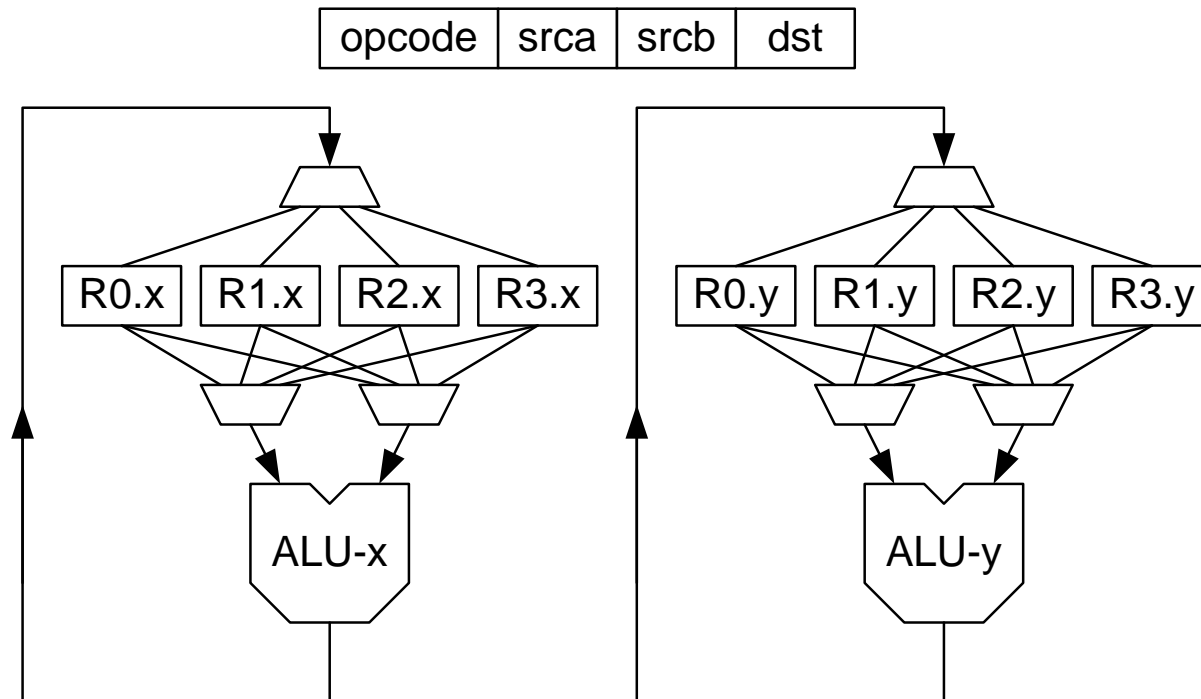
- Classic scalar CPU
  - One set of registers
  - Single ALU
- Instruction controls data-flow
  - Opcode: what should ALU do?
  - Sources: where is data from?



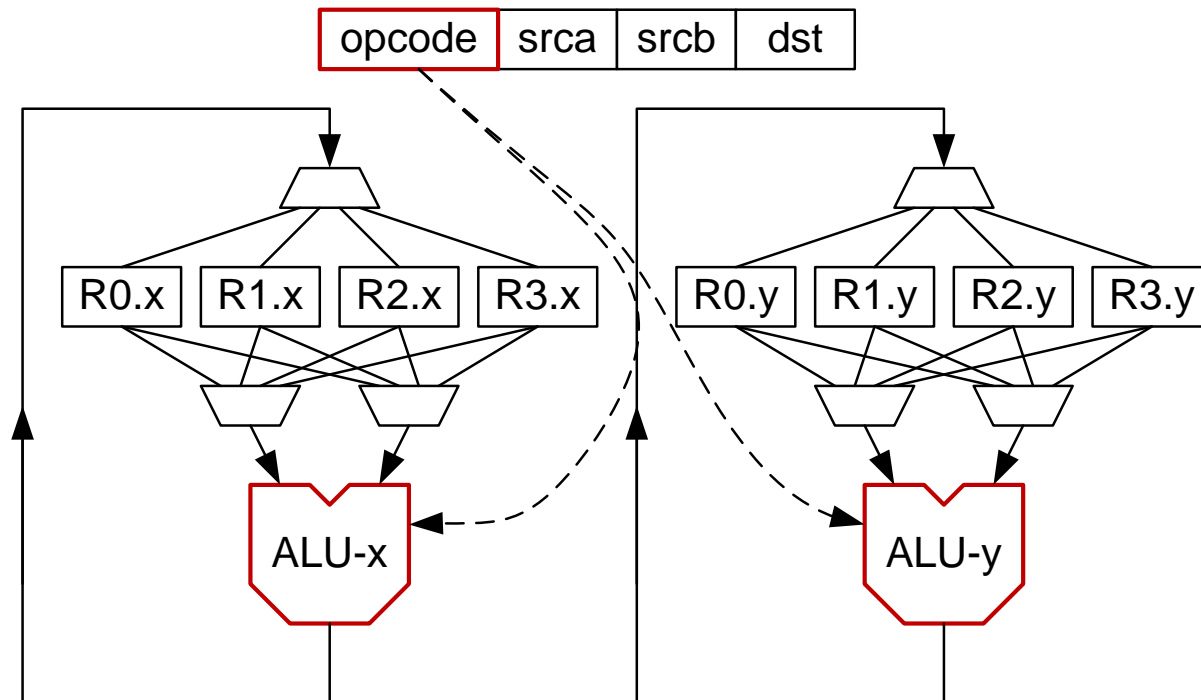


- Classic scalar CPU
  - One set of registers
  - Single ALU
- Instruction controls data-flow
  - Opcode: what should ALU do?
  - Sources: where is data from?
  - Destination: where does data go?

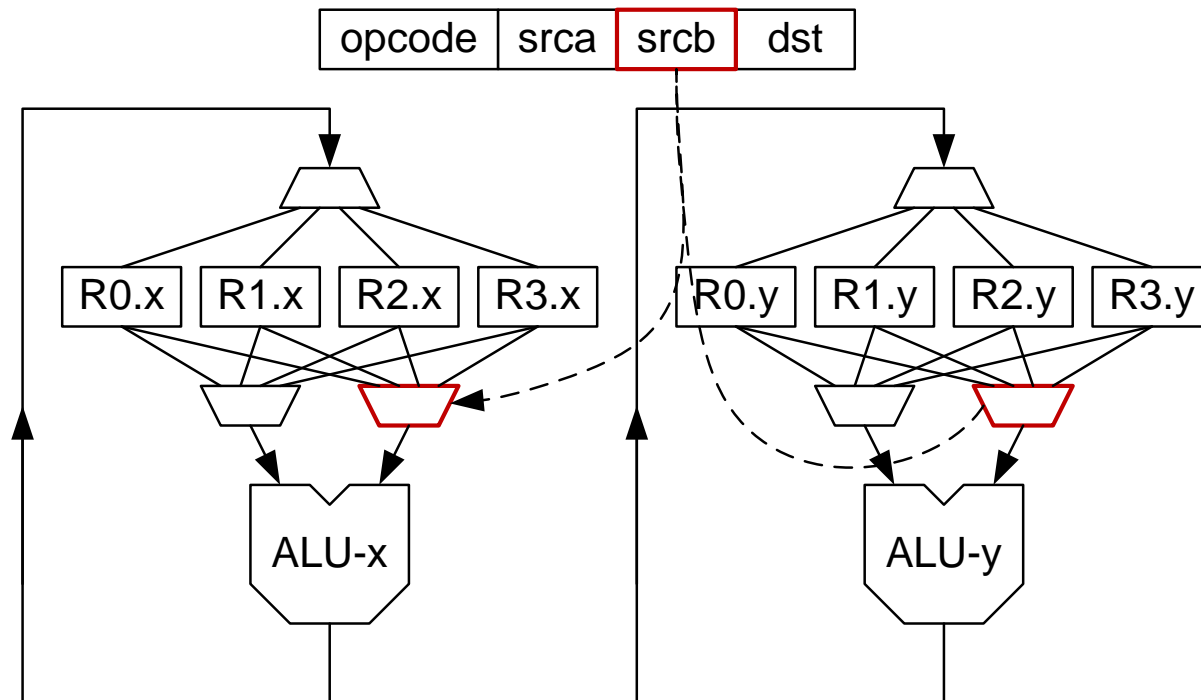
- SIMD: Each data-path (lane) is isolated from the others
  - One set of registers for each SIMD lane
  - One ALU for calculation in each lane



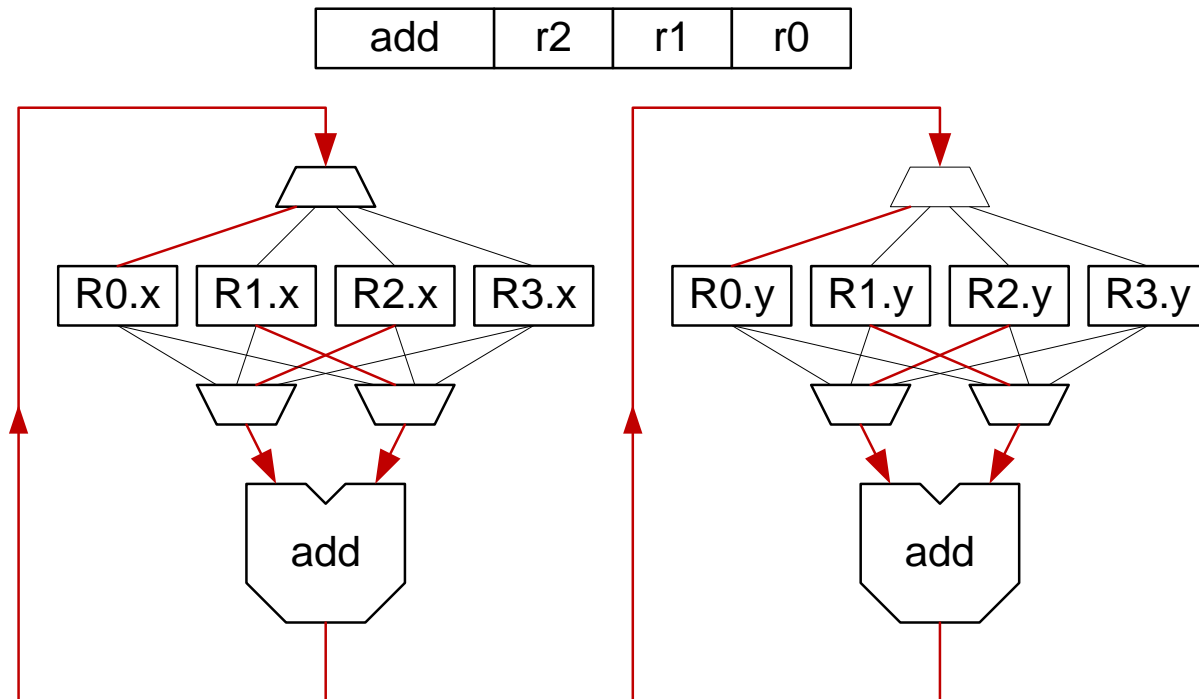
- SIMD: Each data-path (lane) is isolated from the others
  - One set of registers for each SIMD lane
  - One ALU for calculation in each lane
- Same instruction is applied to all lanes



- SIMD: Each data-path (lane) is isolated from the others
  - One set of registers for each SIMD lane
  - One ALU for calculation in each lane
- Same instruction is applied to all lanes

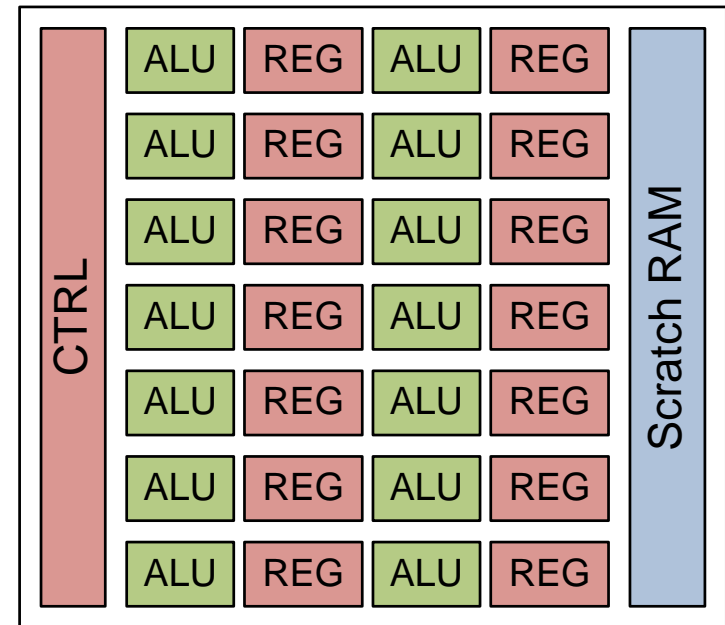
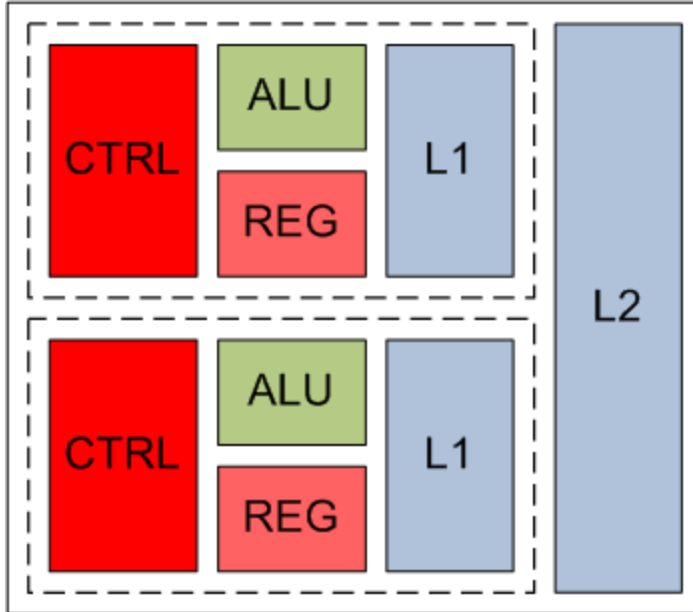


- SIMD: Each data-path (lane) is isolated from the others
  - One set of registers for each SIMD lane
  - One ALU for calculation in each lane
- Same instruction is applied to all lanes



# GPU : Fill the chip with ALUs!

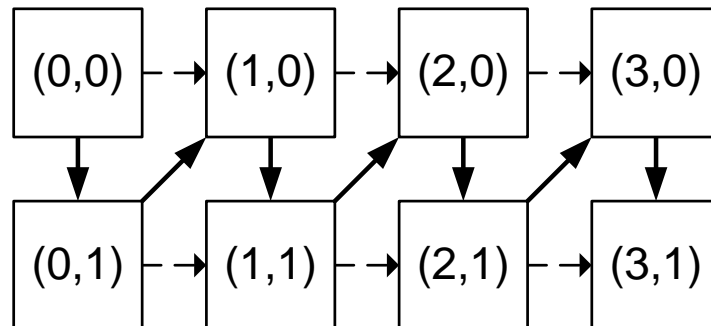
- ALUs are the good part; let's have as many as possible
  - Need to have simple ALUs so they can be small
  - Sacrifice cache area, move to simple scratch-pad RAM



# Recap: Iteration Spaces : Sequential

- Iteration space over a 2D set of points
- Loop control-flow forces execution order

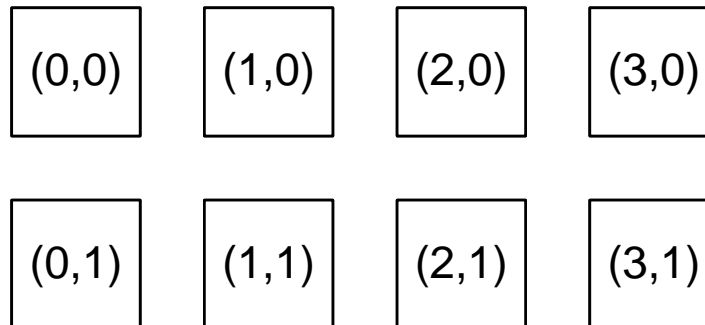
```
for x in [0,4)
  for y in [0,2)
    f(x,y);
```



# Iteration Spaces : Nested Parallel

- Two parallel for loops – remove control-flow dependency

```
par_for x in [0,4)
  par_for y in [0,2)
    f(x,y);
```

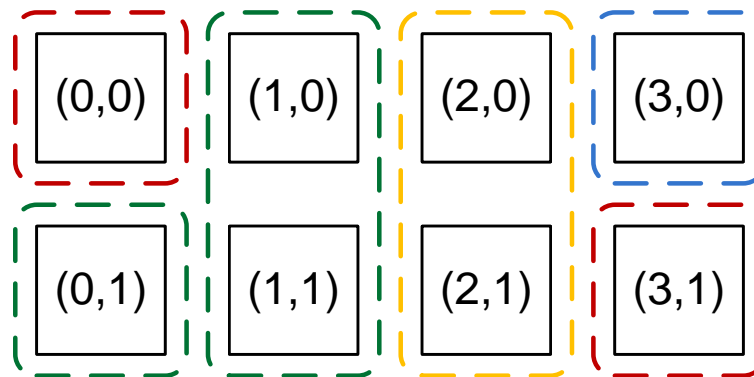




# Iteration Spaces : Nested Parallel

- Two parallel for loops – remove control-flow dependency
- Nesting order controls how TBB can split loops
  - If y is inner loop, will tend to agglomerate vertically

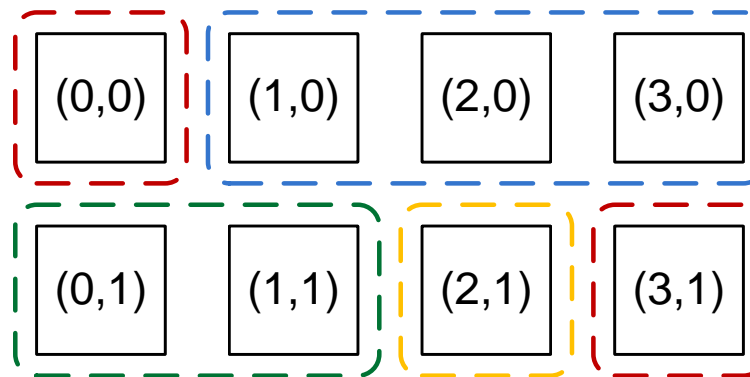

```
par_for x in [0,4)
  par_for y in [0,2)
    f(x,y);
```



# Iteration Spaces : Nested Parallel

- Two parallel for loops – remove control-flow dependency
- Nesting order controls how TBB can split loops
  - If x is inner loop, will tend to agglomerate horizontally

```
par_for y in [0,2)  
  par_for x in [0,4)  
    f(x,y);
```



# Iteration Spaces : Parallel 2D

- Single parallel for loop describing entire iteration space

```
par_for_2d (x,y) in [0,4) × [0,2)  
  f(x,y);
```

(0,0)

(1,0)

(2,0)

(3,0)

(0,1)

(1,1)

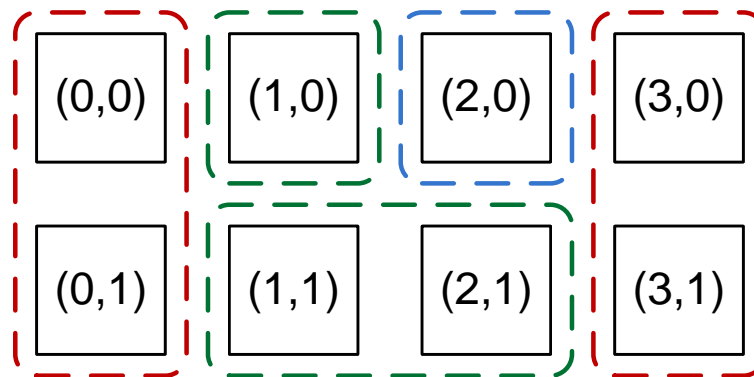
(2,1)

(3,1)

# Iteration Spaces : Parallel 2D

- Single parallel for loop describing entire iteration space
- Scheduler is free to group tasks horizontally or vertically
  - Consider both loops at once, rather than one loop then another

```
par_for_2d (x,y) in [0,4) × [0,2)  
  f(x,y);
```



# Iteration Spaces : GPU Work-Items

- Each point in iteration space corresponds to a GPU work-item

```
cl::Kernel f;  
cl::NDRange size(4,2);  
cl::enqueueKernel(f, size);
```

(0,0)

(1,0)

(2,0)

(3,0)

(0,1)

(1,1)

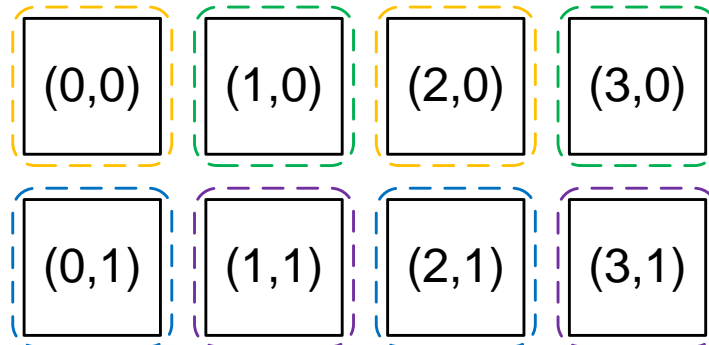
(2,1)

(3,1)

# Iteration Spaces : GPU Work-Items

- Each point in iteration space corresponds to a GPU work-item
- No adaptive agglomeration needed for the GPU
  - Work-items and work-groups provide architectural agglomeration

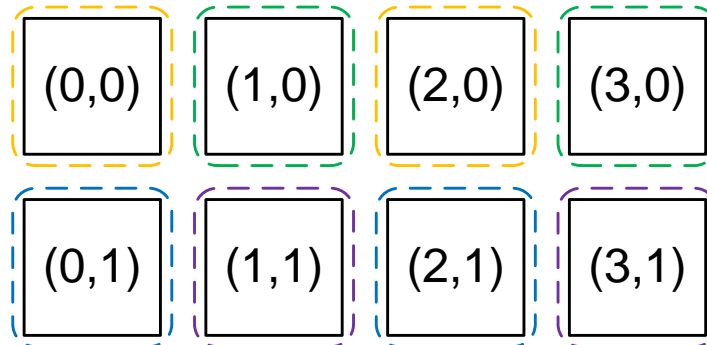
```
cl::Kernel f;  
cl::NDRange size(4,2);  
cl::enqueueKernel(f, size);
```



# Iteration Spaces : GPU Work-Items

- Each point in iteration space corresponds to a GPU work-item
- No adaptive agglomeration needed for the GPU
- Co-ordinates are built into hardware registers: not parameters

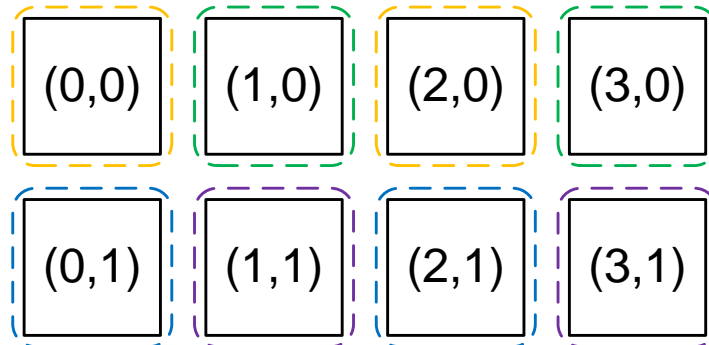
```
__kernel f()  
{  
    uint x=get_global_id(0);  
    uint y=get_global_id(1);  
}
```



# Iteration Spaces : GPU Hierarchy

- Multiple work-items are organised into a ***local work-group***
- Batches of work-items issued by GPU as ***warps***
  - A warp is some sub-set of a local work-group

```
cl::Kernel f;  
cl::NDRange gSize(4,2);  
cl::NDRange lSize(2,2);  
cl::enqueueKernel(f,gSize,lSize);
```

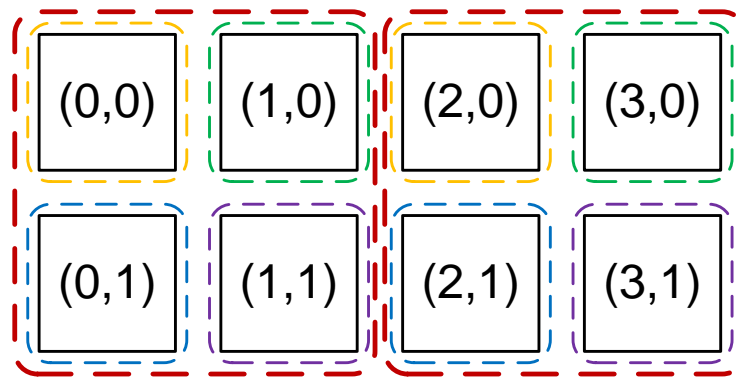




# Iteration Spaces : GPU Hierarchy

- Multiple work-items are organised into a ***local work-group***
- Each local work-group executes in one processor on GPU
- Many processors on a GPU: *need multiple local groups*

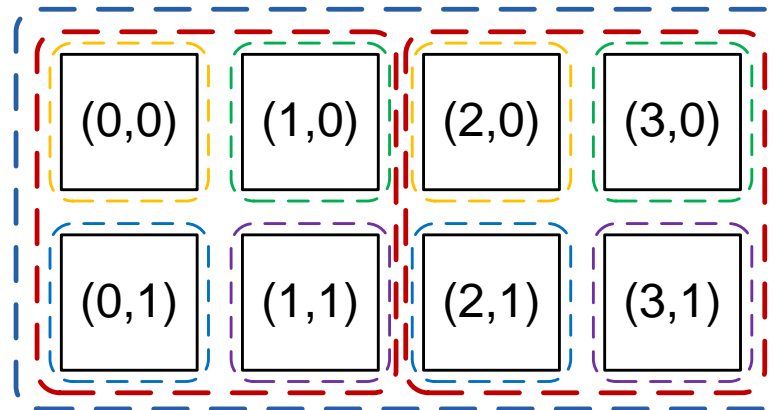
```
cl::Kernel f;  
cl::NDRange gSize(4,2);  
cl::NDRange lSize(2,2);  
cl::enqueueKernel(f,gSize,lSize);
```



# Iteration Spaces : GPU Hierarchy

- Multiple blocks of threads are organised into **grids** of blocks
- Use grid and block index to get global index of thread

```
cl::Kernel f;  
cl::NDRange gSize(4,2);  
cl::NDRange lSize(2,2);  
cl::enqueueKernel(f,gSize,lSize);
```



# The Memory Hierarchy

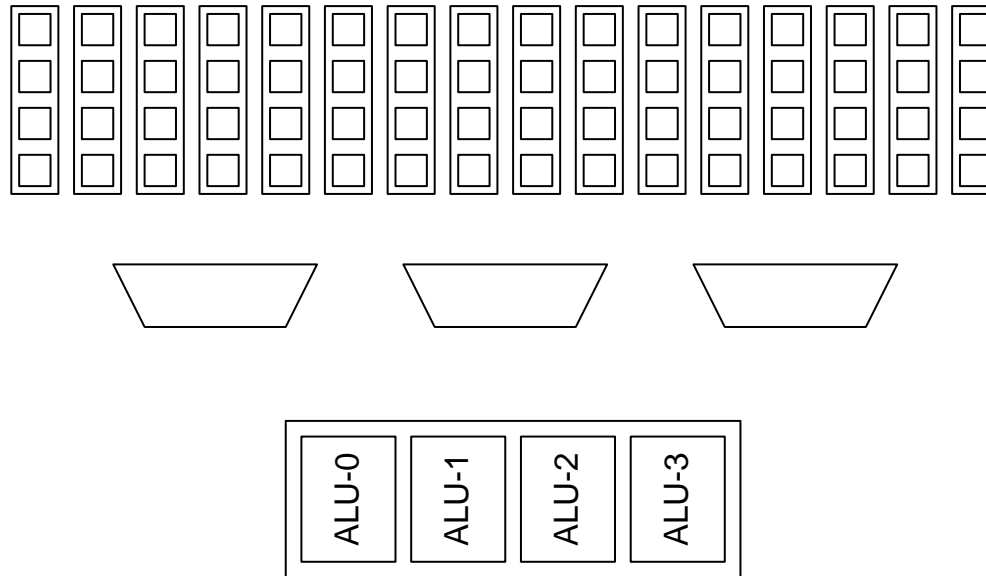
- Three types of read-write storage available within the GPU
  - **Private** (registers) : Only accessible to one thread (work-item)
  - **Local** (scratchpad) : Accessible to threads in a local work-group
  - **Global** (off-chip DRAM): Accessible to any thread on the GPU
- Two types of read-only storage available
  - **Constant memory** : Caches constants, e.g. Pi, lookup tables
  - **Texture memory** : Supports interpolated table lookups
  - Both are cached views of global memory

# Storage: Speed versus size

- **Registers** : very fast to access, fairly abundant on GPUs
  - SIMD registers : dedicated path between registers and ALUs
- **Local memory** : fast to access, not very large though
  - Fast path between registers and shared memory
  - Can move data between threads within the same block
- **Global memory** : big, shared by everyone
  - Very high latency – hundreds of clock cycles
  - Relatively high bandwidth if accesses are ordered carefully
  - If many threads access global memory some will stall
- Use shared memory as a **scratchpad** memory
  - Manually stage data from global into shared memory
  - Same effect as a cache, but less transparent and more efficient

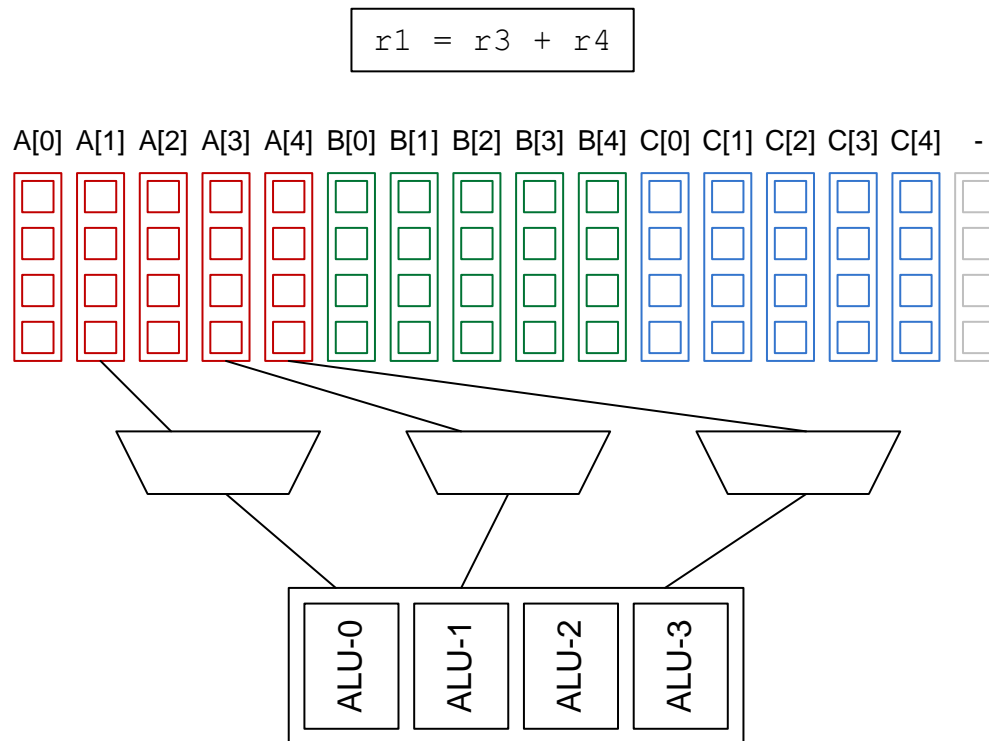
# Register Allocation

- GPU has register file with fixed number of lanes per register



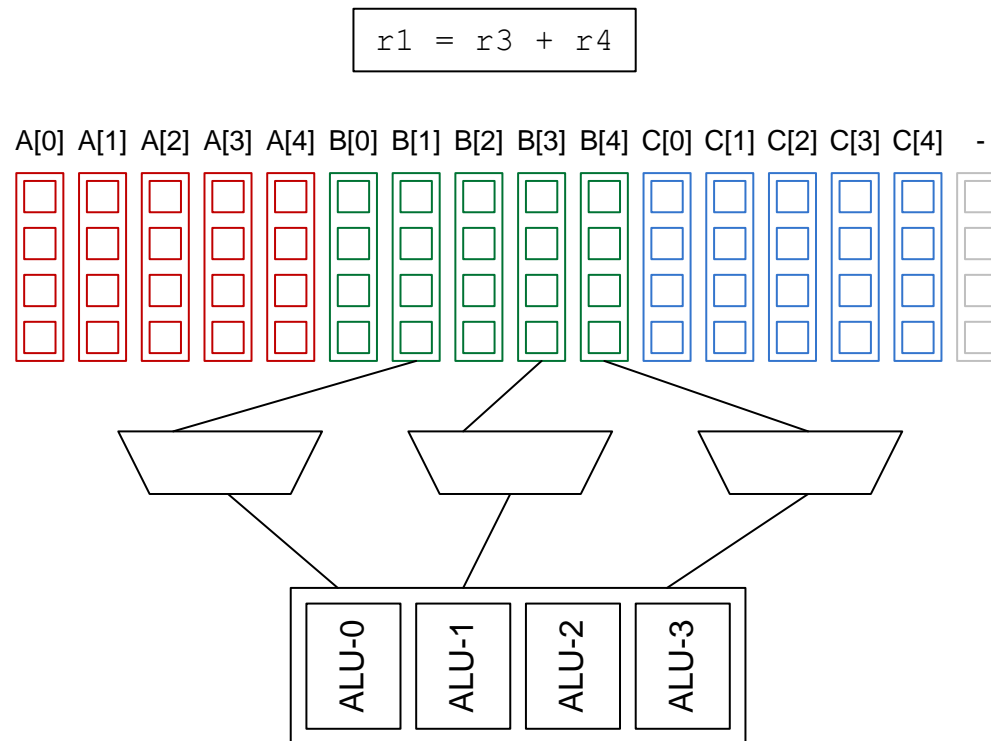
# Register Allocation

- GPU has register file with fixed number of lanes per register
- At run-time sets of registers are dedicated to a warp



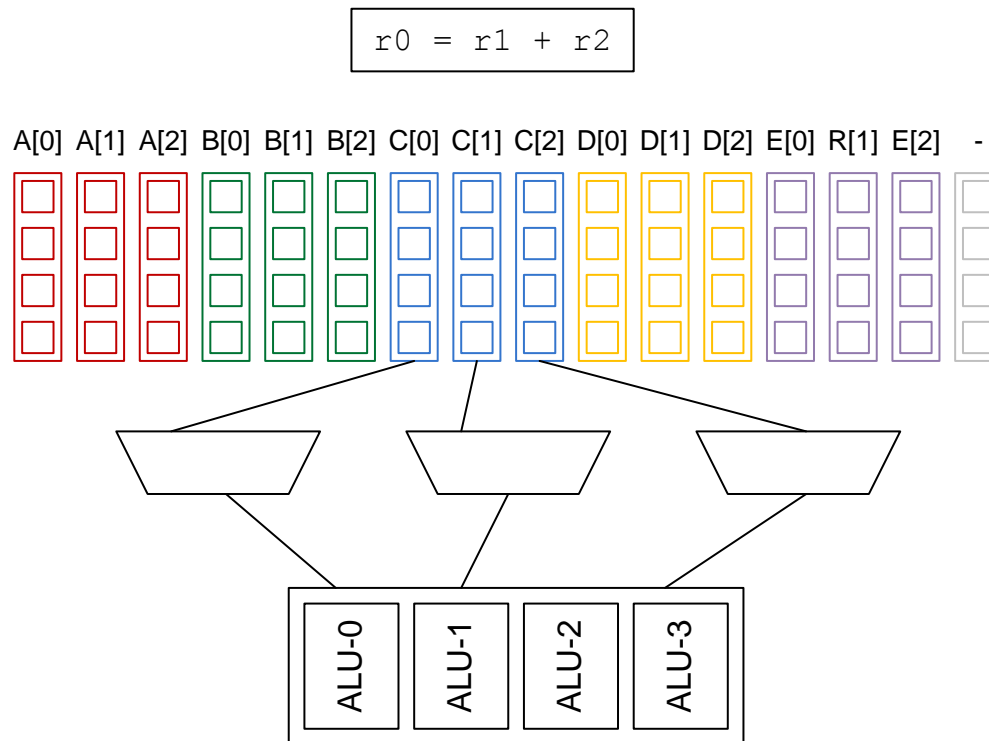
# Register Allocation

- Instruction arguments are mapped to registers at run-time



# Register Allocation

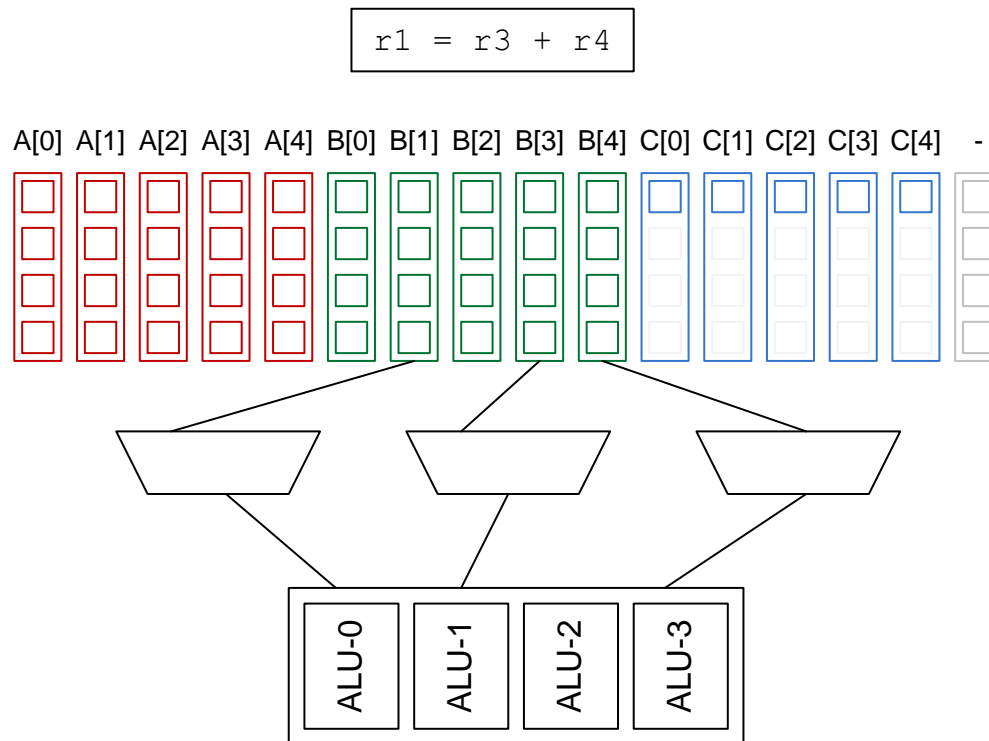
- Instruction arguments are mapped to registers at run-time
- If each thread uses fewer registers, larger blocks are possible





# Register Allocation

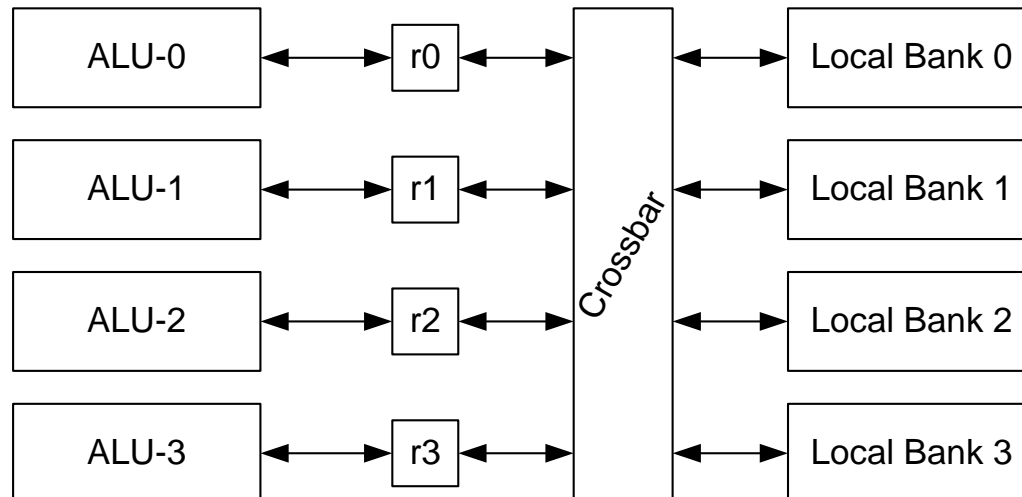
- Block size does not have to be a multiple of warp size
- Can have a warp with one active thread – not very efficient



# Shared memory banks

- ALUs are connected to one register lane
- Shared memory has one bank per lane via a crossbar

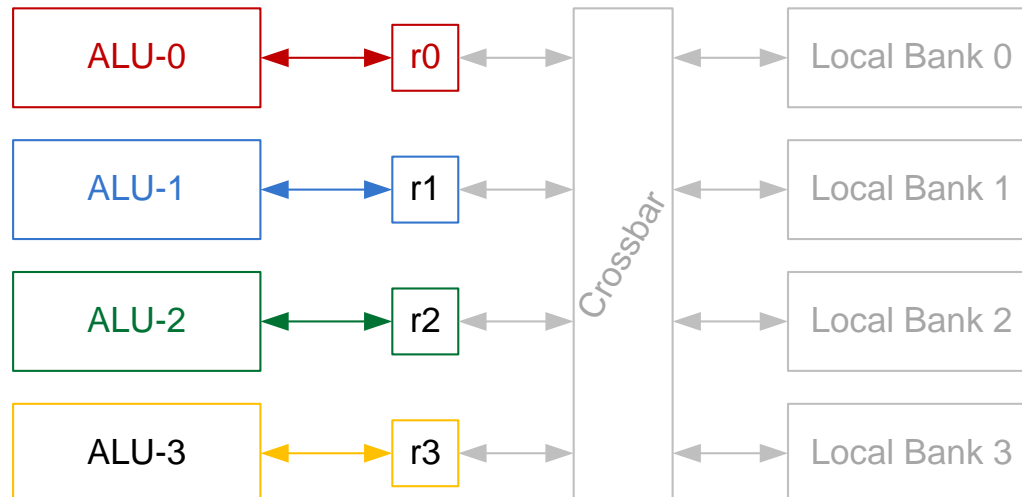
```
__global__ void MyKernel()  
{  
    i=i+1;  
    v=myMem[i];  
}
```



# Shared memory banks

- ALUs are connected to one register lane
- Shared memory has one bank per lane via a crossbar

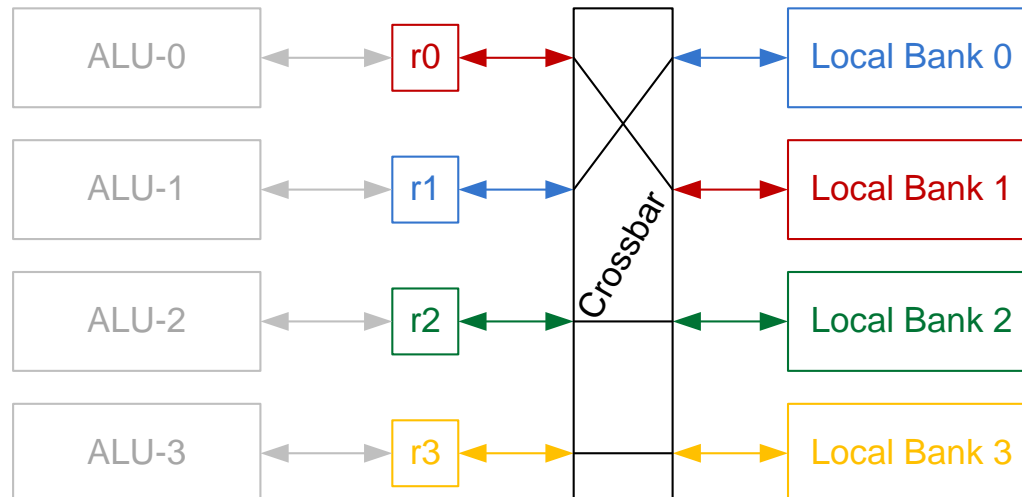
```
__global__ void MyKernel()  
{  
    i=i+1;  
    v=myMem[i];  
}
```



# Shared memory banks

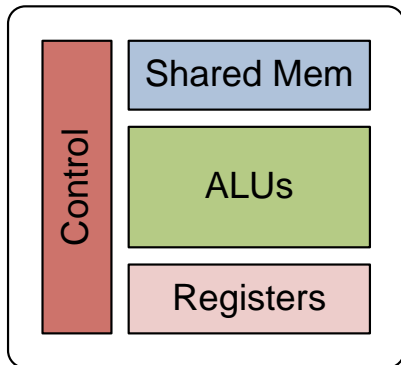
- ALUs are connected to one register lane
- Shared memory has multiple banks connected via crossbar
- Crossbar: fast comms. between threads *in the same local group*

```
__global__ void MyKernel()  
{  
    i=i+1;  
    v=myMem[i];  
}
```



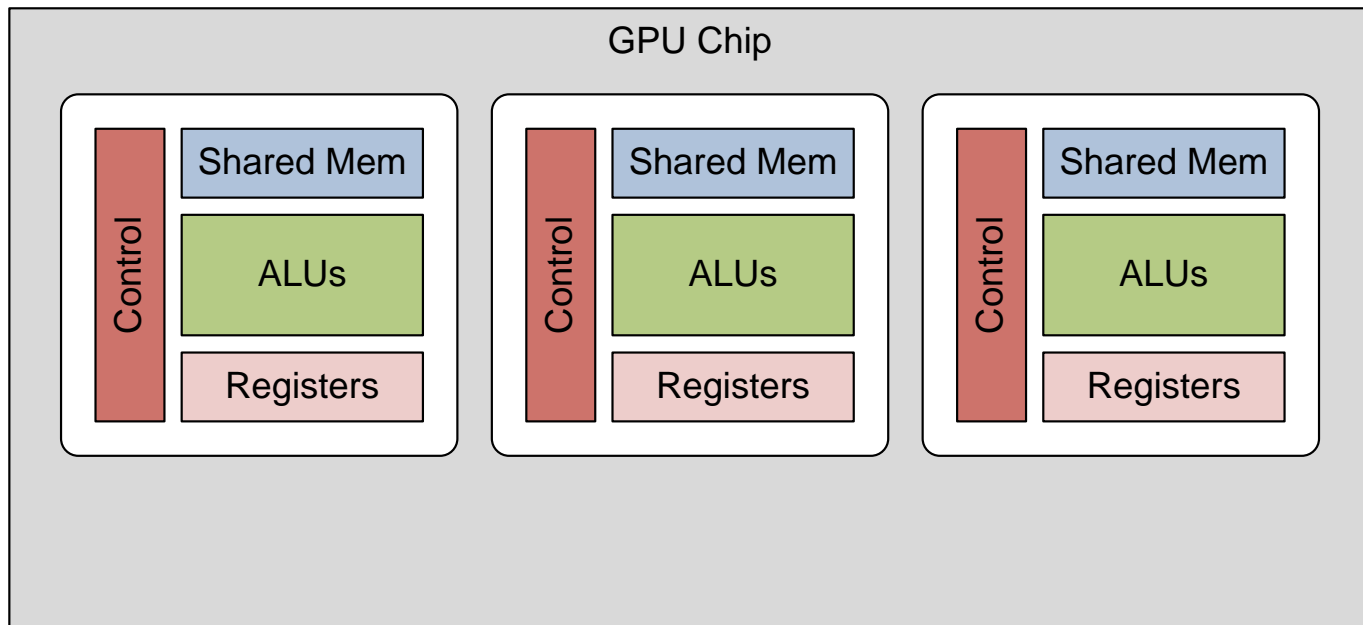
# Putting it all together

- Individual processor within GPU
- Schedules instructions which are applied to warps of threads
- Contains all warps executing within a given block



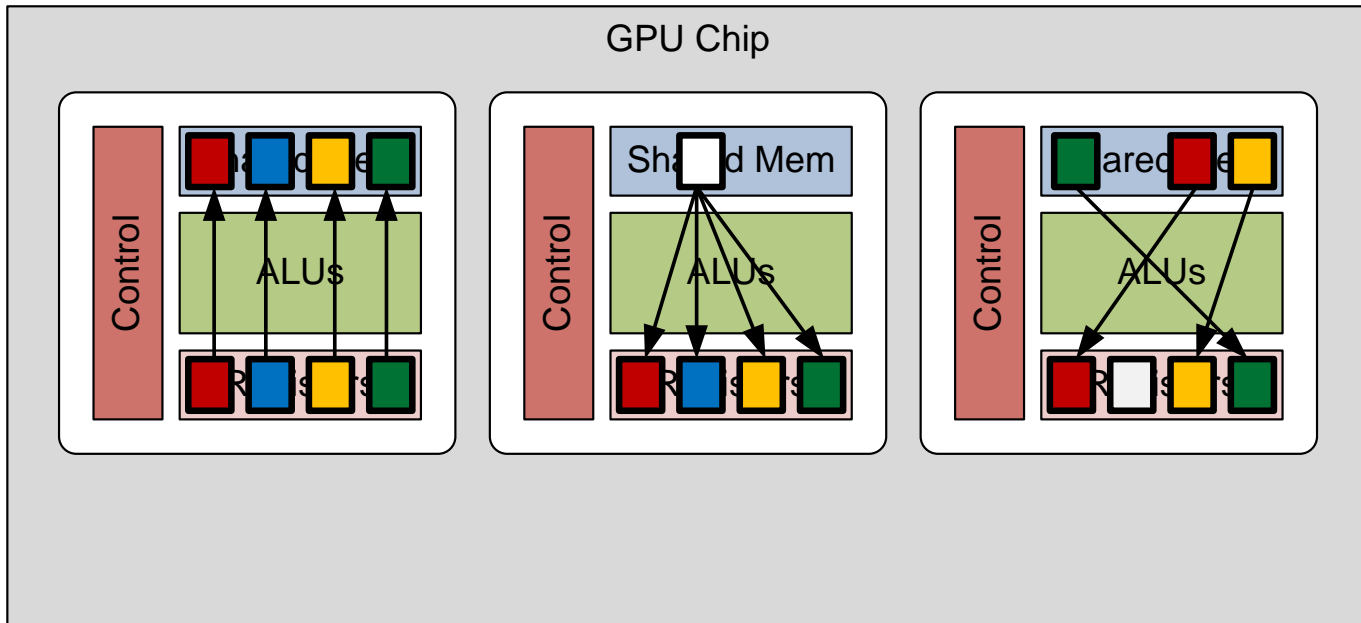
# Putting it all together

- GPU chip contains multiple processors working in parallel
- All threads from a single block are in one processor
- Blocks from a single grid can be executed on many processors



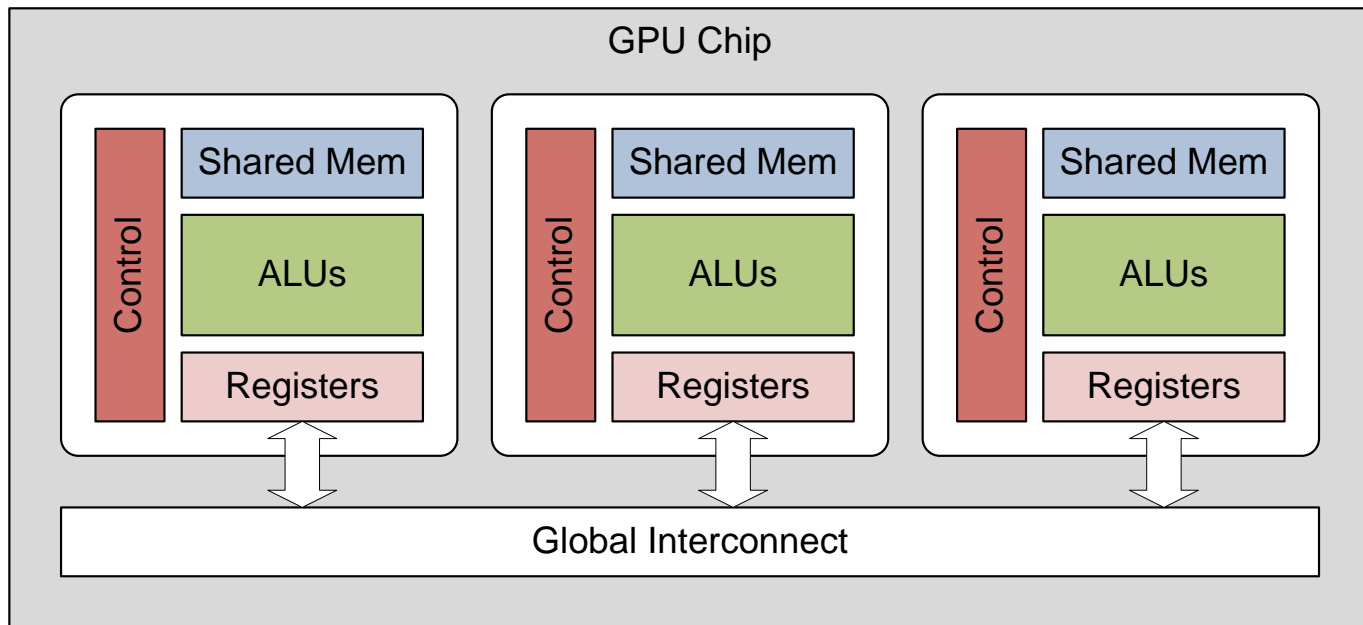
# Putting it all together

- Threads within processor can communicate via shared memory
- Cannot read and write shared memory from other processors



# Putting it all together

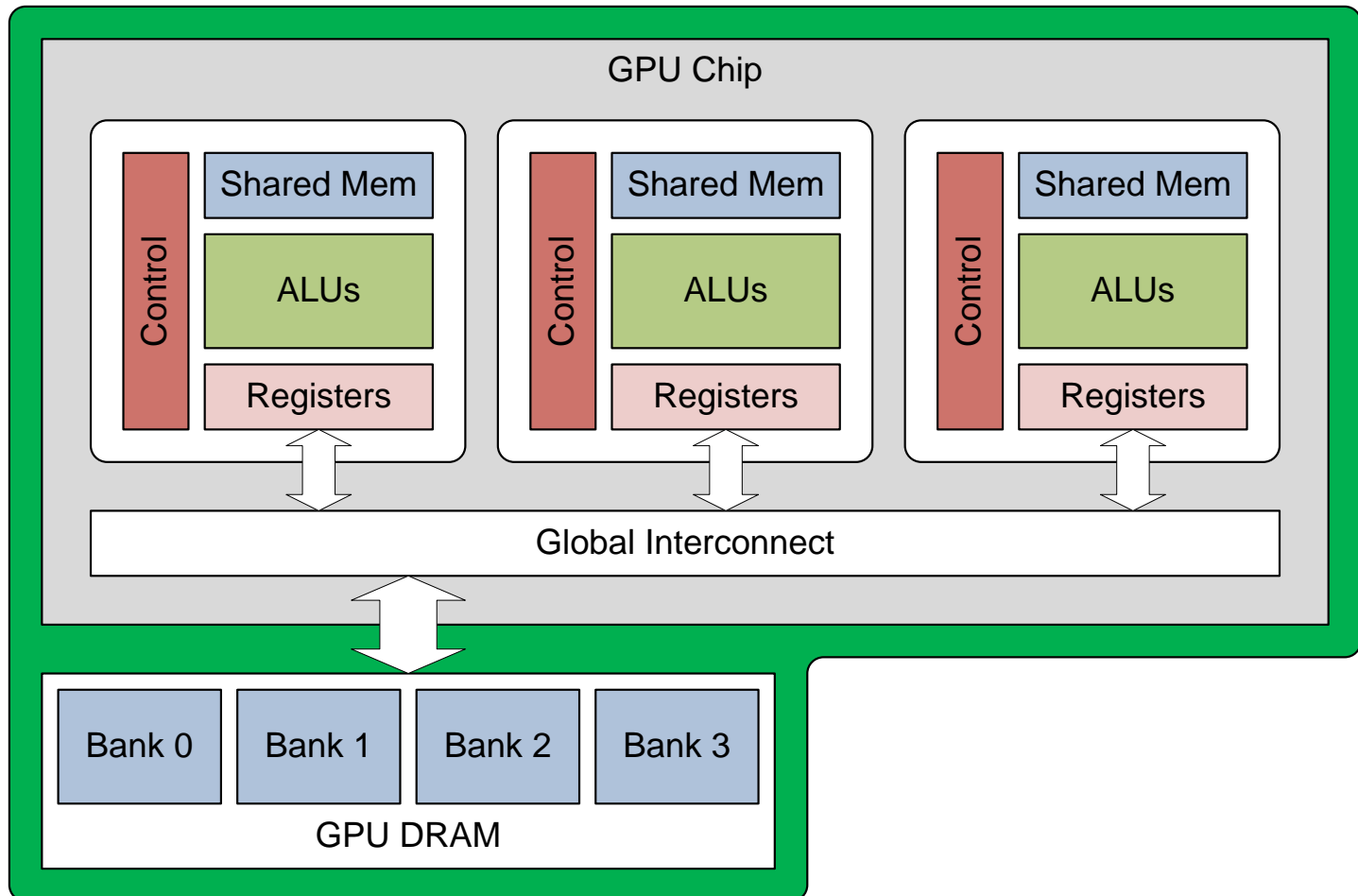
- Processors within GPU share a chip-wide global interconnect
- Used for things like managing blocks, instruction fetch, etc.





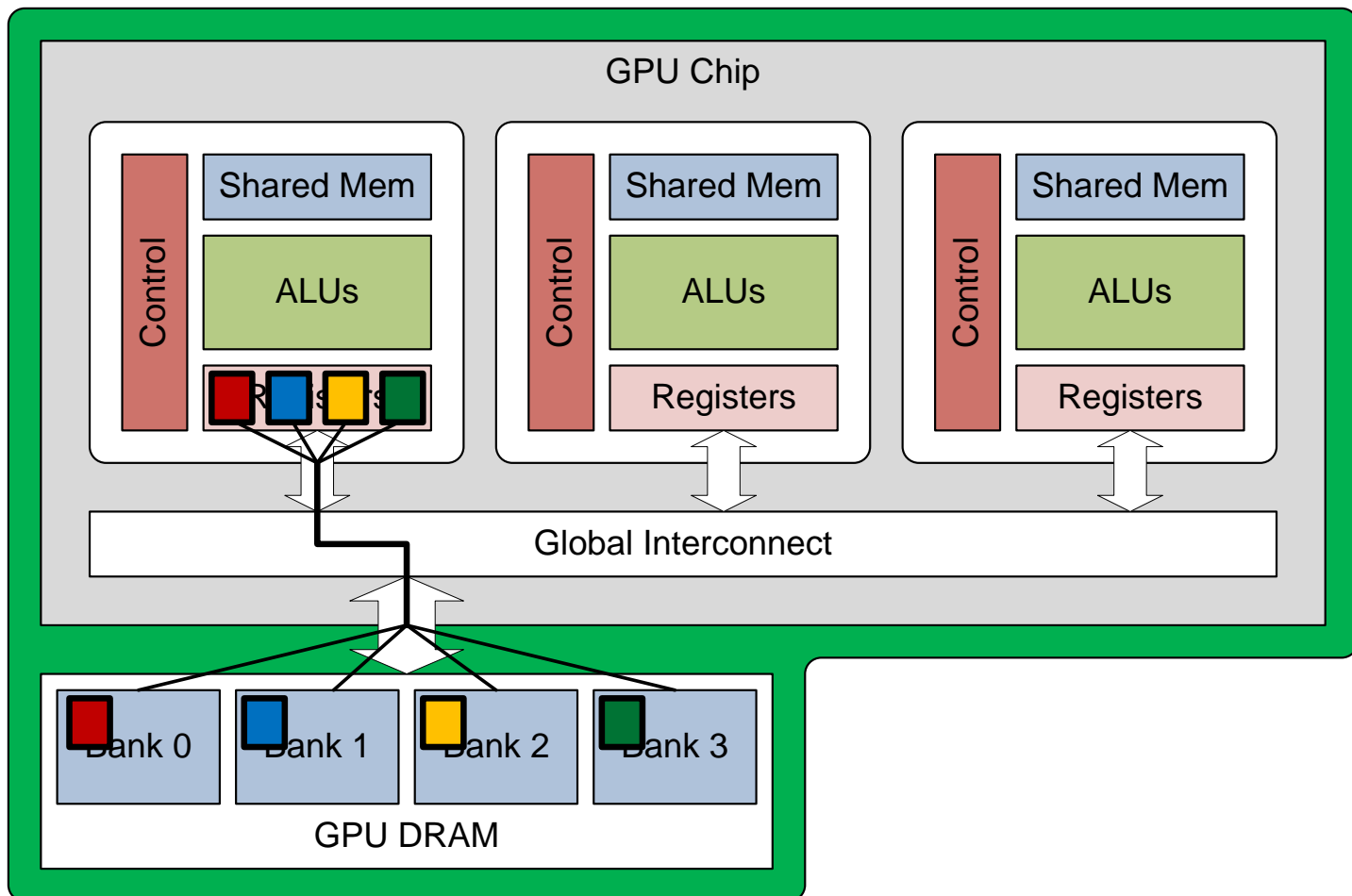
# Putting it all together

- Processors within GPU share a chip-wide global interconnect
- Used for things like managing blocks, instruction fetch, etc.
- Also connects GPU processors to **off-chip** global memory



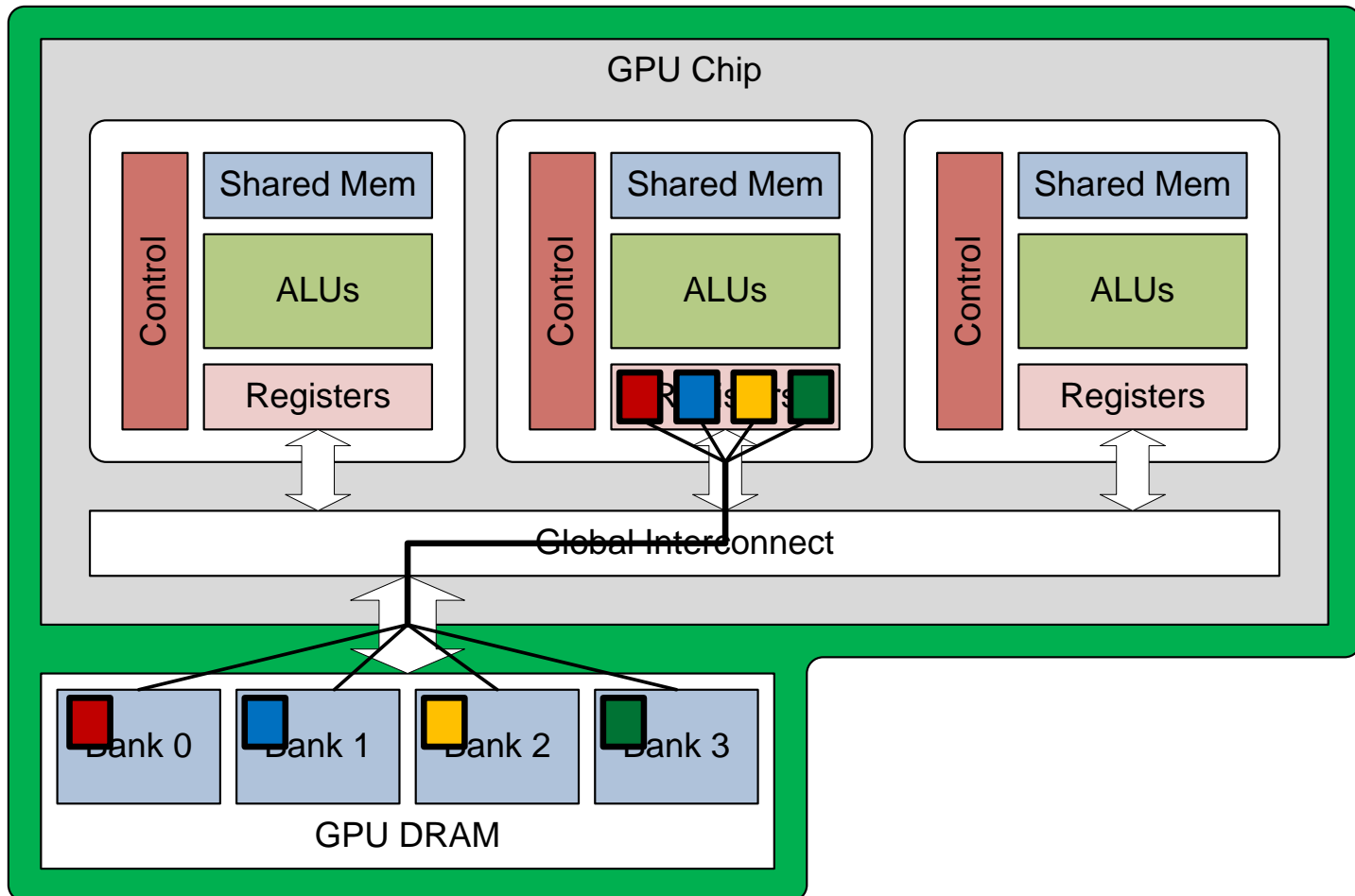
# Putting it all together

- Global DRAM contains multiple parallel memory banks
- To maximise bandwidth all threads should access different banks



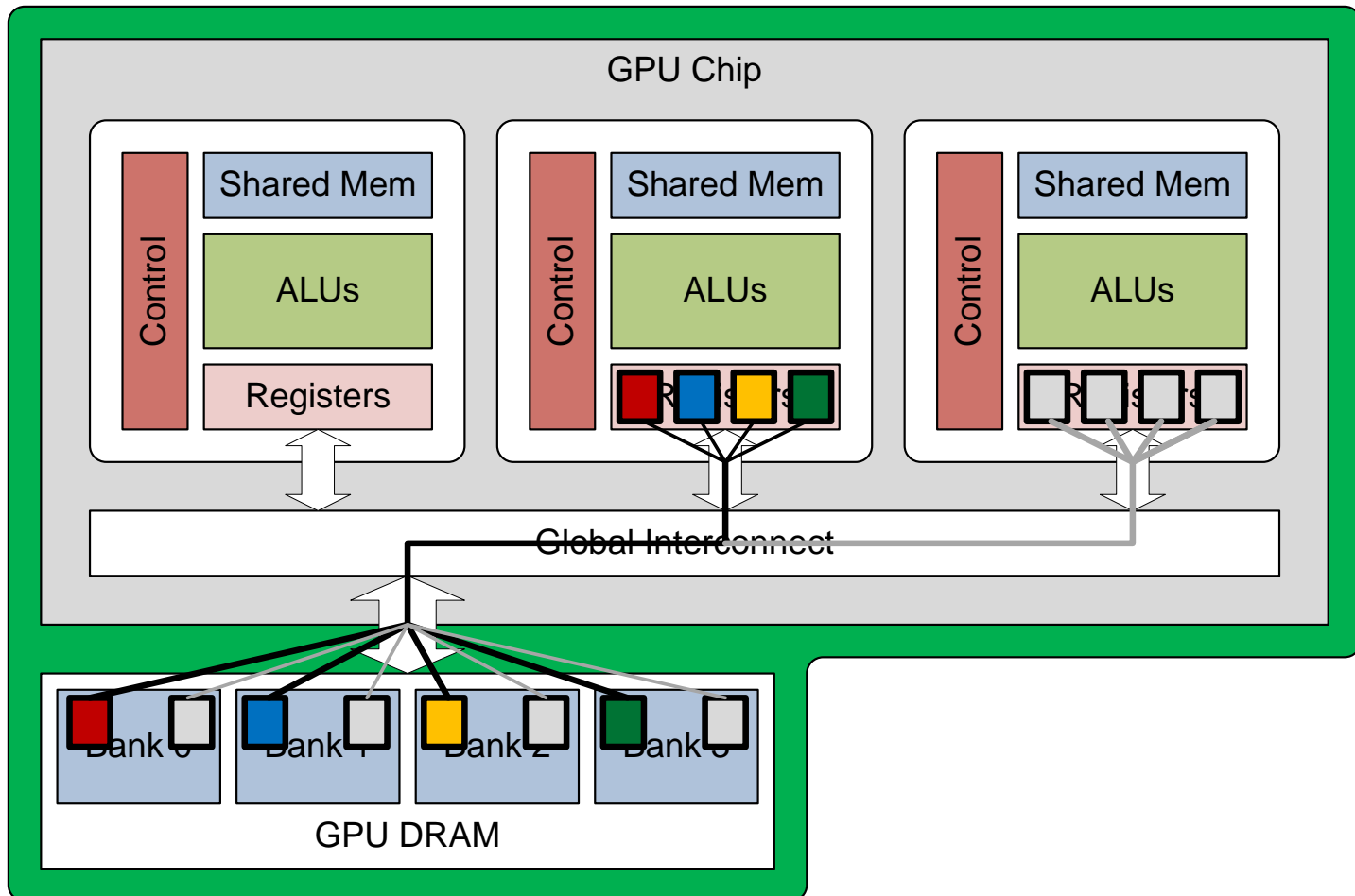
# Putting it all together

- Global DRAM contains multiple parallel memory banks
- To maximise bandwidth all threads should access different banks
- Allows all threads within GPU to communicate



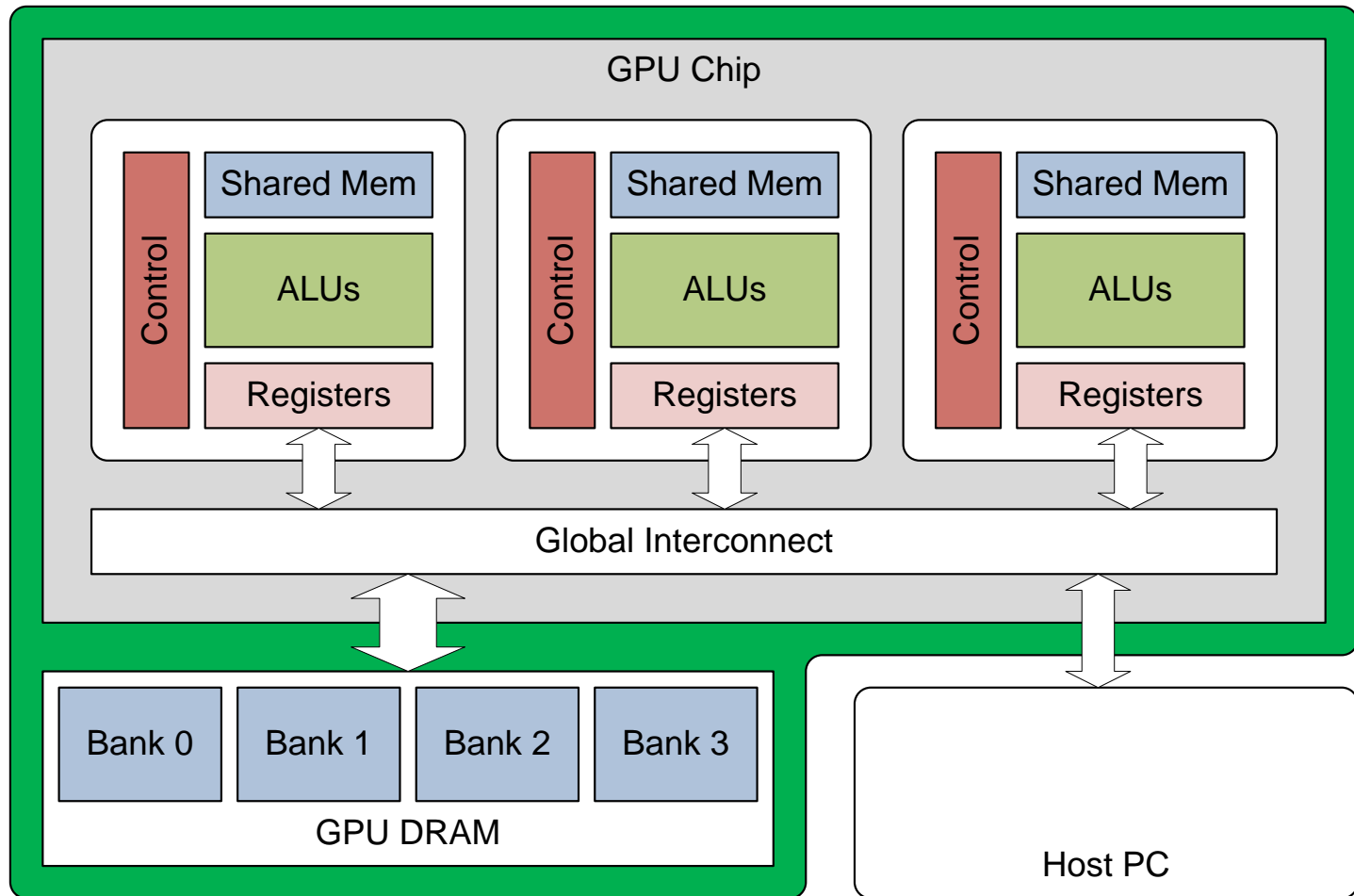
# Putting it all together

- Global RAM has high bandwidth, but there are lots of processors
- If memory traffic is too high, then some processors must wait
- Need lots of warps ready to run: ***hide latency of global memory***



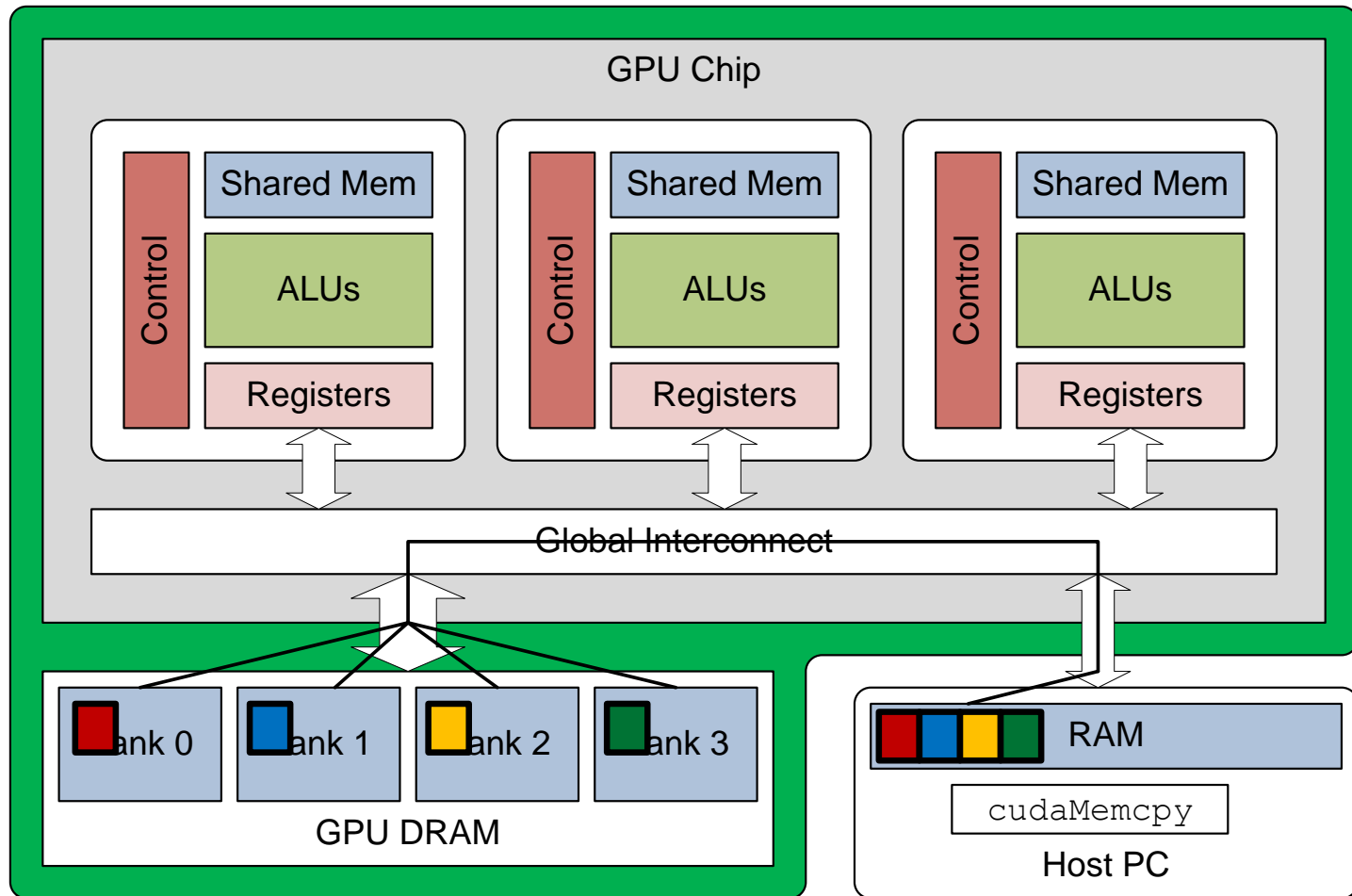
# Putting it all together

- Global interconnect also connects GPU to the outside world
- Typically a high-bandwidth PCIe connection to a host PC
- GPU is quite simple: no OS, no networking, no disk



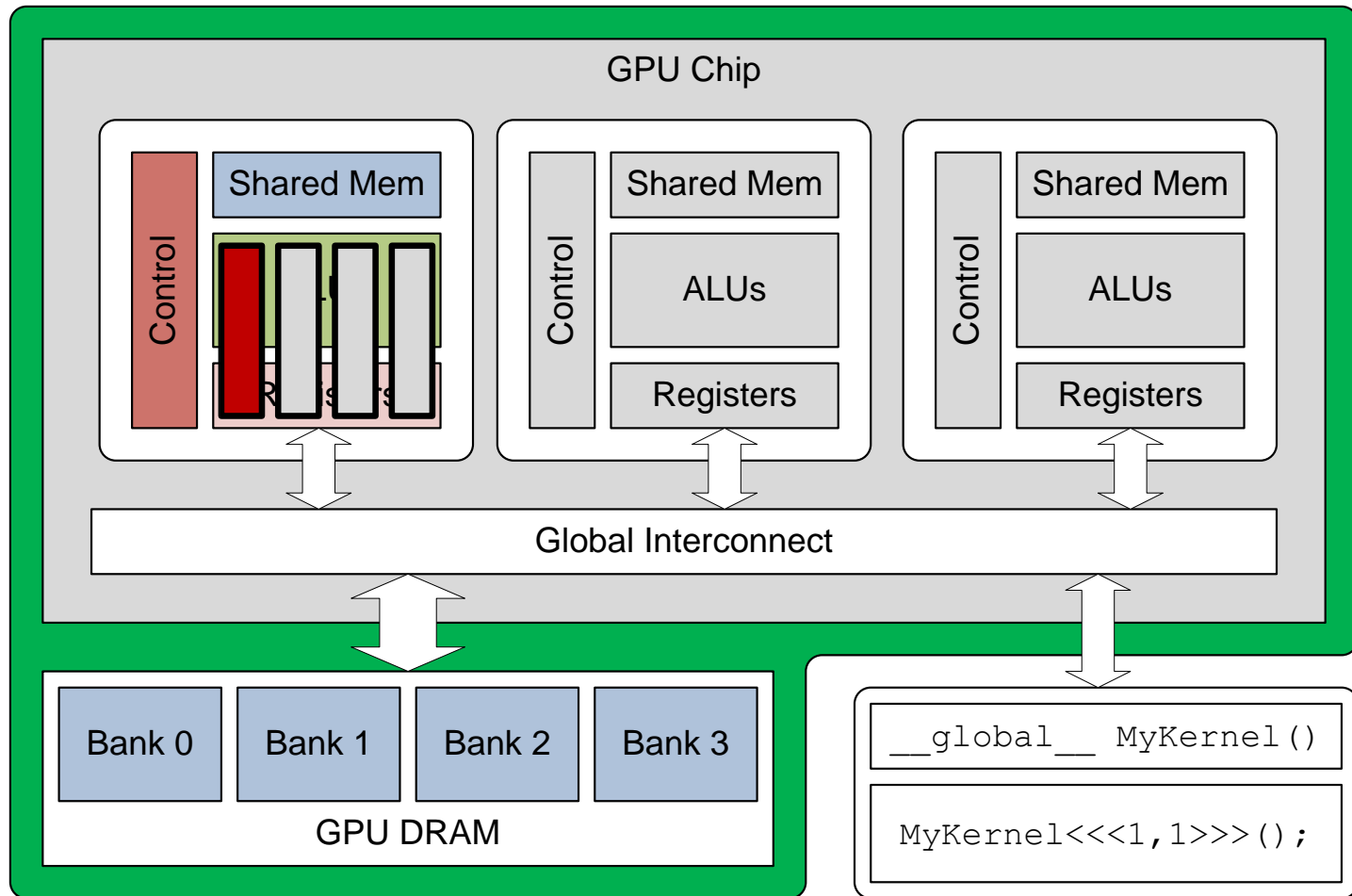
# Putting it all together

- Host PC can read and write to global memory
- Send input: `enqueueWrite`
- Get input: `enqueueRead`



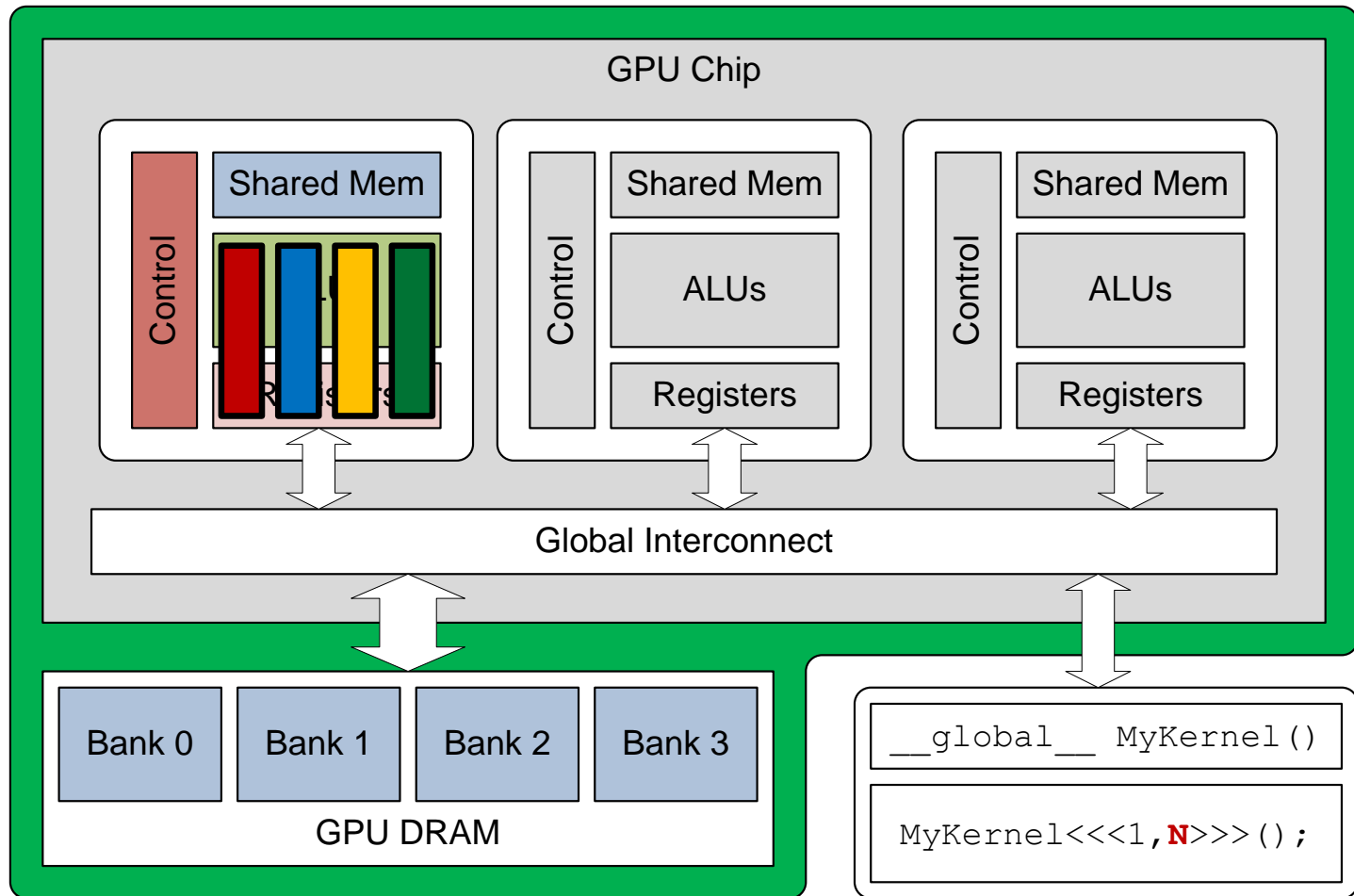
# Putting it all together

- Host PC must launch grids of threads: the CPU can't do it
- Launching a single block with 1 thread is very inefficient
  - Will be much slower than a CPU



# Putting it all together

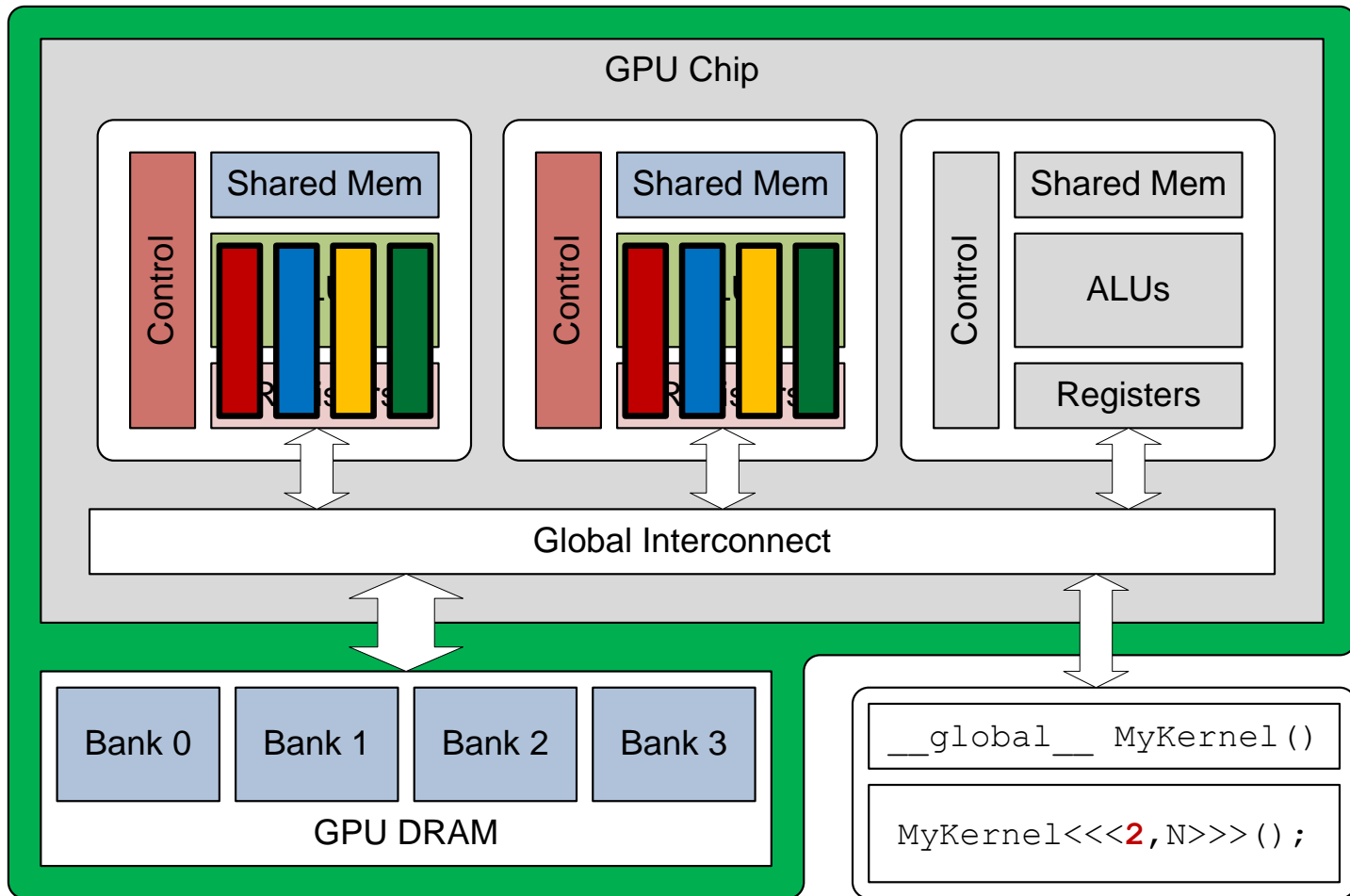
- Each block should contains enough threads to occupy processor
- Threads should be a multiple of warp-size: use all SIMD lanes
- Need multiple warps available: hide ALU and memory latency





# Putting it all together

- We want to try and use multiple processors at once
- Each block must be executed on one processor: need many blocks
- Launch **grids** of blocks, which will be scheduled on all processors



# General Lessons : Iteration Spaces

- Computation is organised into a hierarchy of iteration spaces
  - **Work-item**: granularity of control-flow = *one SIMD lane*
  - **Warp/Wavefront**: granularity of scheduling = *one Program Counter*
  - **Local Workgroup**: collection of work-items within one processor
  - **Global Workgroup**: collection of work-items with shared code
- Need to have an appropriate sizes for each level
  - **Work-item**: Startup cost vs amount of work done
    - *If your kernel doesn't contain a loop, how much work can it do?*
  - **Local group**: balance registers/thread against threads/block
    - *Want lots of warps ready to run; hide ALU and memory latency*
  - **Global group**: Want enough grids to utilise all processors
    - *Balance no. of threads vs startup cost of thread*

# General Lessons : Communication

- **Registers** : Local to just one thread
  - Each thread has a unique copy of variables in the kernel
- **Local Memory** : Shared within just one block
  - Can be used to communicate between threads
  - Threads within warp should try to read/write non-conflicting banks
- **Global Memory** : Shared amongst all work-items in a GPU
  - Threads can communicate with any thread in **any** grid
  - Allocated and freed by host using `cl::Buffer`
  - State is maintained between grid executions
- **Host Memory** : Local to CPU – “normal” RAM
  - GPU and CPU have different address spaces
  - Use `enqueueRead/Write` to move data between them