Programming Heterogeneous Systems

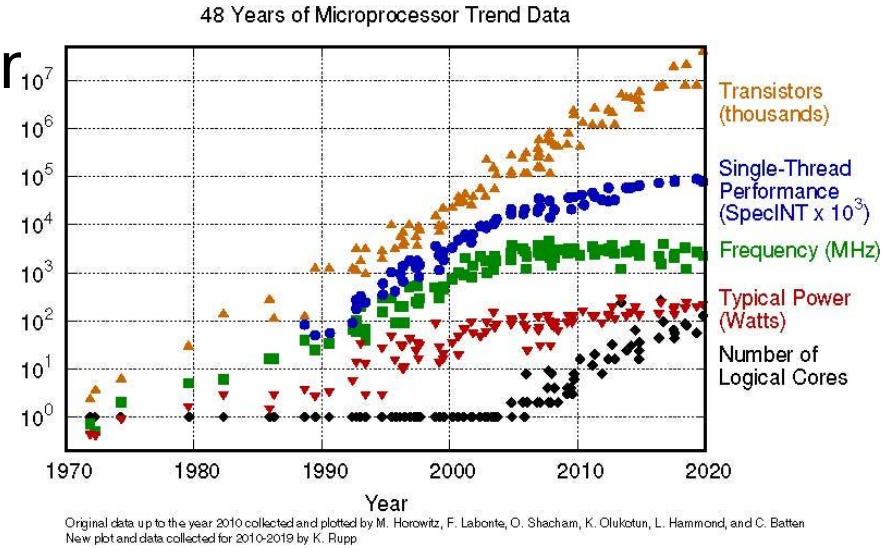
Russel Arbore

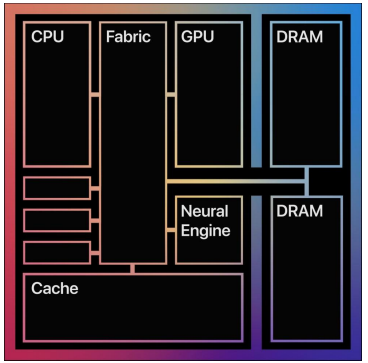
Outline

1. What are heterogeneous systems?
2. Hercules
3. The future

What are Heterogeneous Syster

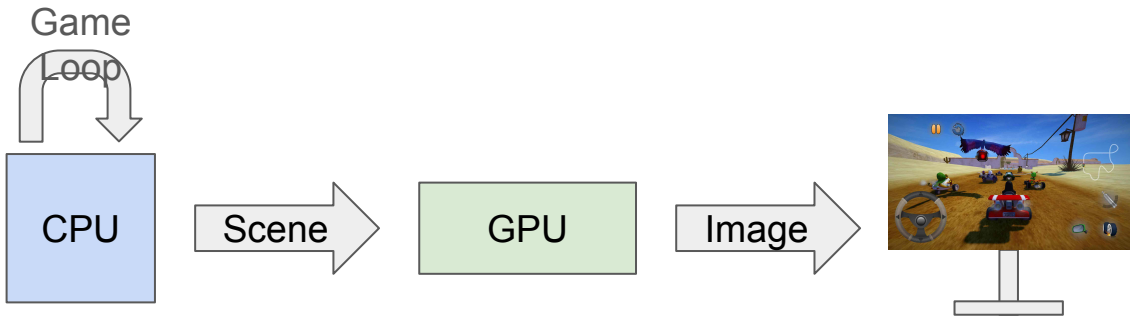
- Hardware specialization has become primary method of improving hardware performance
- Generality is still important for many computing systems
- Many systems integrate different specialized components





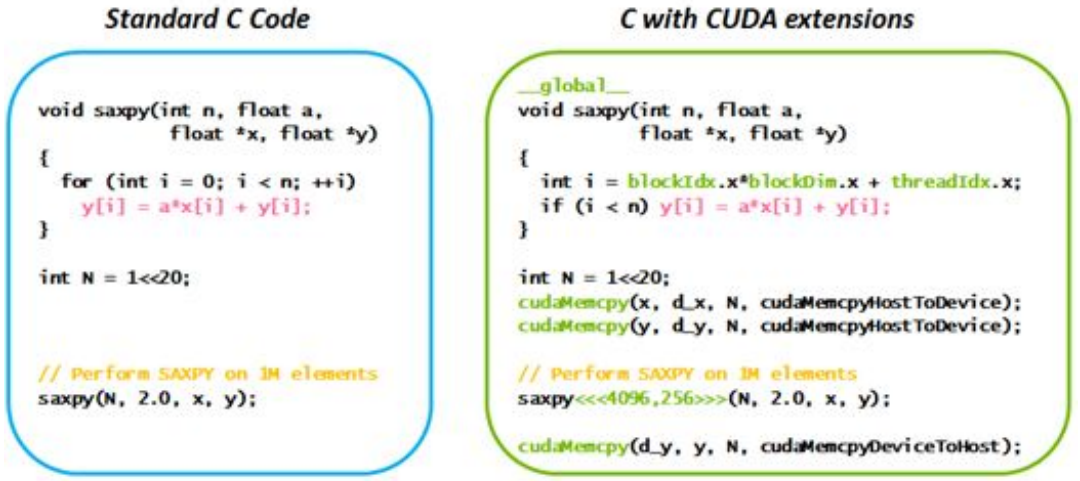
A Simple Example

- Gaming PCs are one of the most prevalent (and oldest) examples of heterogeneous systems
- x86 CPU + NVIDIA GPU



OpenGL, Vulkan, CUDA, OpenCL, …

- Two components to a program
  - Host API code (run on the CPU)
  - Device code (run on the GPU)
- CPU responsible for “controlling” the GPU
- Distributed memory
  - Message passing
- Explicit synchronization
- Often asynchronous



OpenGL, Vulkan, CUDA, OpenCL, …

C with CUDA extensions

```
__global__
void saxpy(int n, float a,
           float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
cudaMemcpy(x, d_x, N, cudaMemcpyHostToDevice);
cudaMemcpy(y, d_y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, x, y);

cudaMemcpy(d_y, y, N, cudaMemcpyDeviceToHost);
```

OpenGL, Vulkan, CUDA, OpenCL, …

1. Explicit partitioning of code

C with CUDA extensions

```
__global__
void saxpy(int n, float a,
           float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
cudaMemcpy(x, d_x, N, cudaMemcpyHostToDevice);
cudaMemcpy(y, d_y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, x, y);

cudaMemcpy(d_y, y, N, cudaMemcpyDeviceToHost);
```

OpenGL, Vulkan, CUDA, OpenCL, …

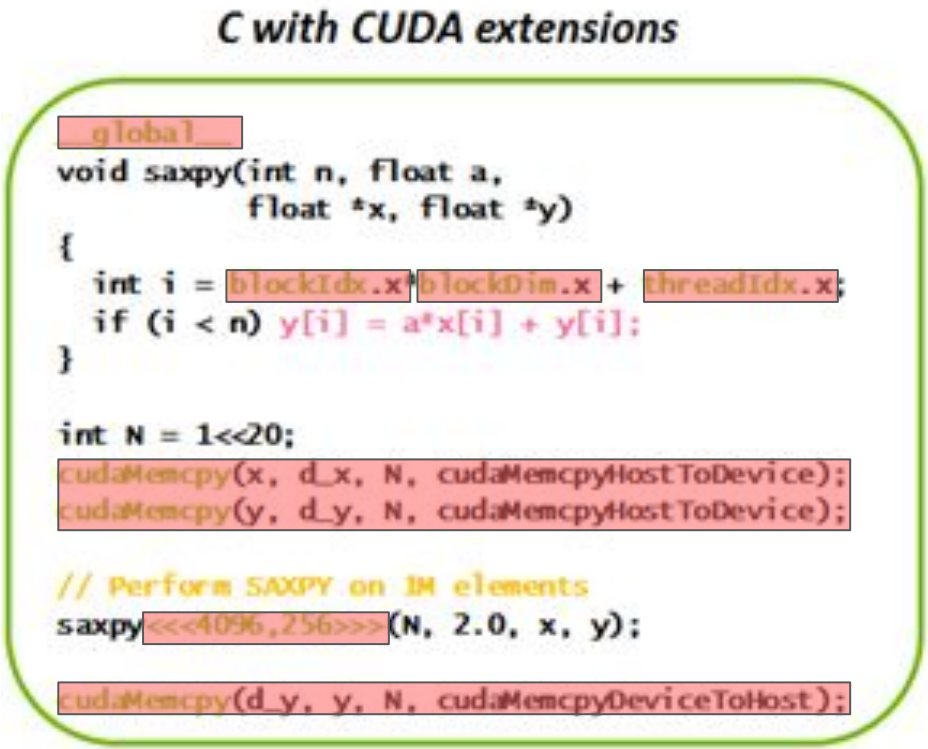
1. Explicit partitioning of code
2. Explicit message passing for communication

C with CUDA extensions

```
__global__
void saxpy(int n, float a,
           float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
cudaMemcpy(x, d_x, N, cudaMemcpyHostToDevice);
cudaMemcpy(y, d_y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, x, y);

cudaMemcpy(d_y, y, N, cudaMemcpyDeviceToHost);
```

OpenGL, Vulkan, CUDA, OpenCL, …

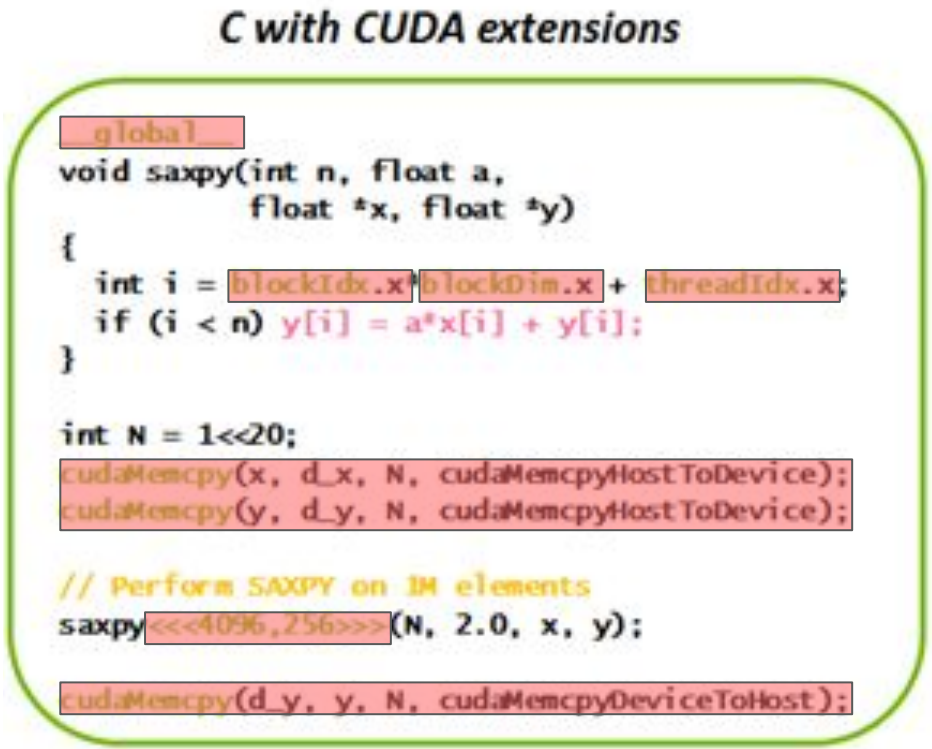
1. Explicit partitioning of code
2. Explicit message passing for communication
3. Device specificities



OpenGL, Vulkan, CUDA, OpenCL, …

1. Explicit partitioning of code
2. Explicit message passing for communication
3. Device specificities

Not shown: synchronization!



Challenges

- APIs are extremely error-prone
  - Synchronization is often a huge pain point
- CPU and GPU languages are different
  - Host API code must match the device code
- Must manually manage memory
- How do you *incrementally* develop a heterogeneous application?
  - Also, specific to a particular machine configuration!
- The most low level API, Vulkan, has struggled to gain adoption due to being “too hard” (!)

Domain specific languages (deep learning)

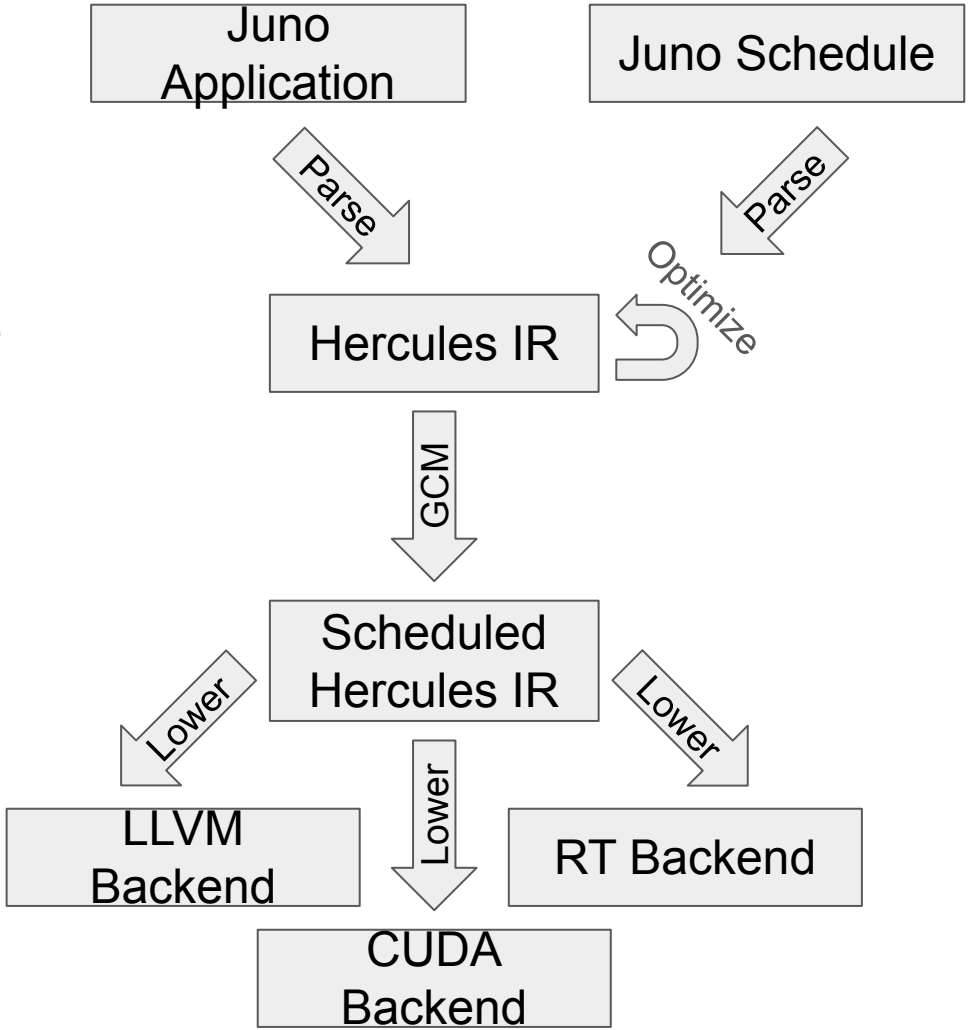
- PyTorch, JAX, Tensorflow, …
- Compile tensor expressions into device and host code automatically
- Pros
  - Mostly transparent to the programmer
  - A *lot* of research has gone into making DL compilers work well in practice (high performance)
- Cons
  - Domain specific
  - Limited control flow
  - Don’t play nice with code “outside” the DSL
  - Still two languages

```
1  dtype = torch.float
2  device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
3  # Randomly initialize weights and put the tensors on a GPU if available
4  a = torch.randn((5, 5), device=device, dtype=dtype)
5  b = torch.randn((5, 5), device=device, dtype=dtype)
6  # Matrix multiplication done on GPU
7  c = torch.mul(a,b)
8  print(" c: {}".format(c))
```

```
c: tensor([[ 0.1512,  0.0265,  1.1087,  0.0544, -2.0399],
       [ 0.0277,  0.2530,  1.4738, -0.1850,  3.9369],
       [ 0.0074,  0.0412,  0.7437, -0.0600, -0.0488],
       [-0.5553, -1.2324,  1.5299, -0.2288,  0.1586],
       [ 0.5845,  0.1385,  0.9252,  0.1865,  0.0159]])

Command took 0.02 seconds -- by jules@databricks.com at 3/25/2021, 2:43:11 PM on Shared Autoscaling Americas
```

Juno & Hercules

- Juno is a programming language for writing code on different devices
  - Single language
  - Hardware agnostic
  - Scheduled
- Hercules is a compiler that lowers Juno code on to different hardware devices



```
#[entry]
fn matmul<n, m, l: usize>
(a: f32[n, m], b: f32[m, l]) -> f32[n, l] {
  let res : f32[n, l];
  @outer for i in 0..n {
    @middle for j in 0..l {
      @inner for k in 0..m {
        res[i, j] += a[i, k] * b[k, j];
      }
    }
  }
  return res;
}
```

Figure 2.1: A Juno program to compute matrix multiplication.

```
// Parallelize by computing output array as 16 blocks
let par = matmul@outer \ matmul@inner;
fork-chunk![4](par);
let (outer, inner, _) = fork-reshape[[0,2],[1],[3]](par);
parallelize!(outer \ inner);
let body = outline(inner);
cpu(body);
// Tile for cache, assuming 64B cache lines
fork-tile![16](body);
let (outer, inner) = fork-reshape[[0,2,4,1,3],[5]](body);
```

Figure 2.2: A schedule to tile and parallelize the matrix multiplication in Figure 2.1.

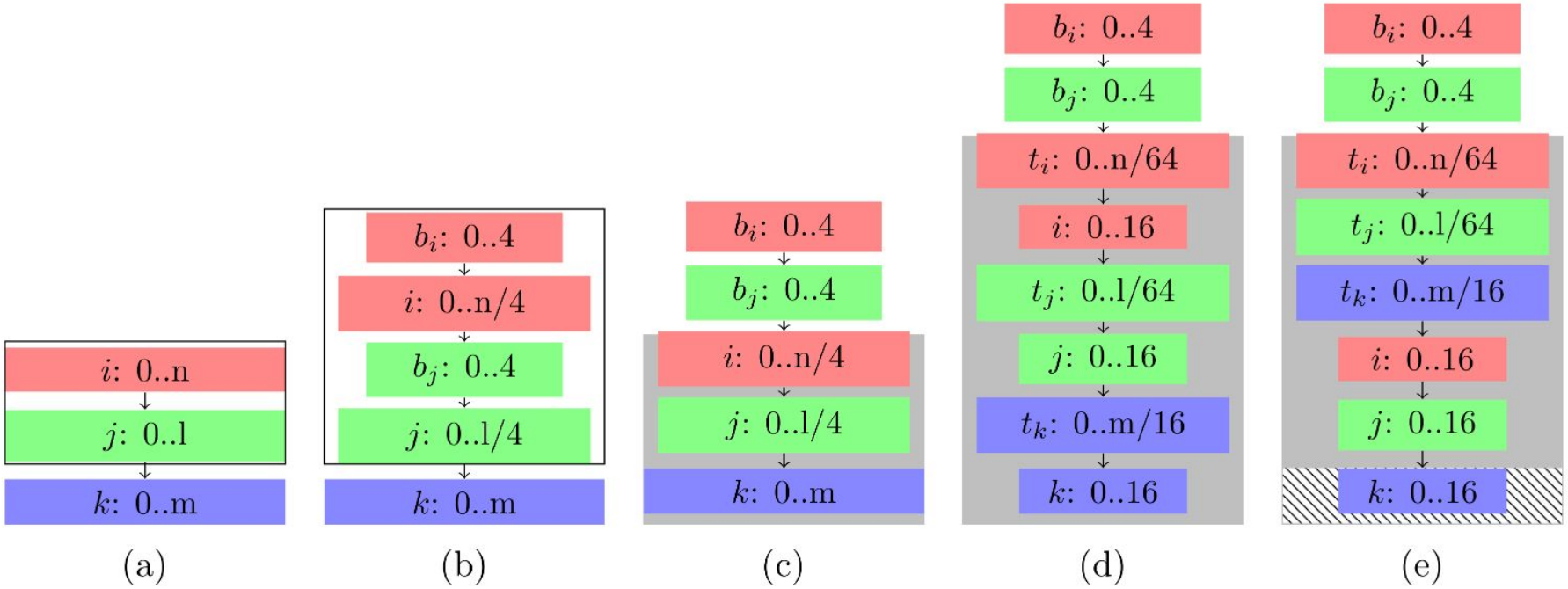


Figure 2.3: The loop structure of `matmul` as it is scheduled. Blue blocks refer to the `@inner` loop, green blocks refer to the `@middle` loop, and red blocks refer to the `@outer` loop.

```
1  #[entry]
2  fn sum<n : usize>(x : f32[n]) -> f32 {
3    let res : f32 = 0;
4    for i = 0 to n {
5      res += x[i];
6    }
7    return res;
8  }
```

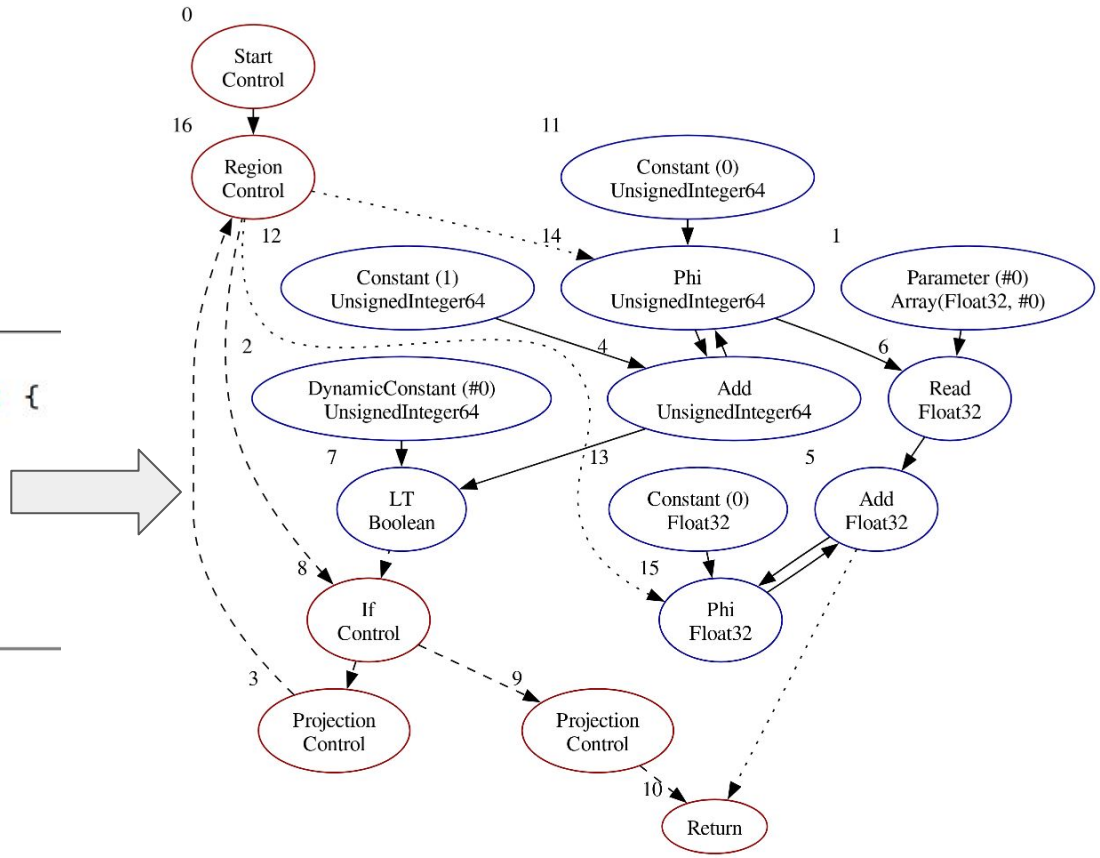


Figure 3.1: A Hercules IR function that adds all the elements in an array. Control nodes are red and data nodes are blue. Data dependencies are solid, control flow edges are dashed, control-data interactions are dotted.

```
1  #[entry]
2  fn matmul<n, m, l: usize>
3  (a: f32[n, m], b: f32[m, l]) -> f32[n, l] {
4    let res : f32[n, l];
5    @outer for i in 0..n {
6      @middle for j in 0..l {
7        @inner for k in 0..m {
8          res[i, j] += a[i, k] * b[k, j];
9        }
10     }
11   }
12   return res;
13 }
```

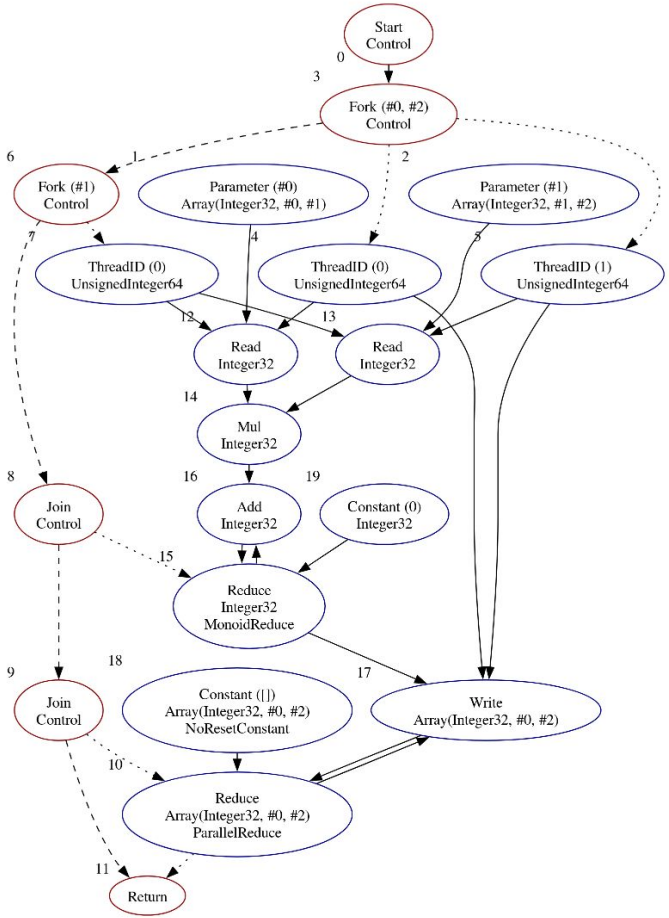


Figure 3.2: A Hercules IR function performing matrix multiplication using fork-joins.

```
#[entry]
fn matmul<n, m, l: usize>
(a: f32[n, m], b: f32[m, l]) -> f32[n, l] {
  let res : f32[n, l];
  @outer for i in 0..n {
    @middle for j in 0..l {
      @inner for k in 0..m {
        res[i, j] += a[i, k] * b[k, j];
      }
    }
  }
  return res;
}
```

1. Single language for both CPU and GPU code

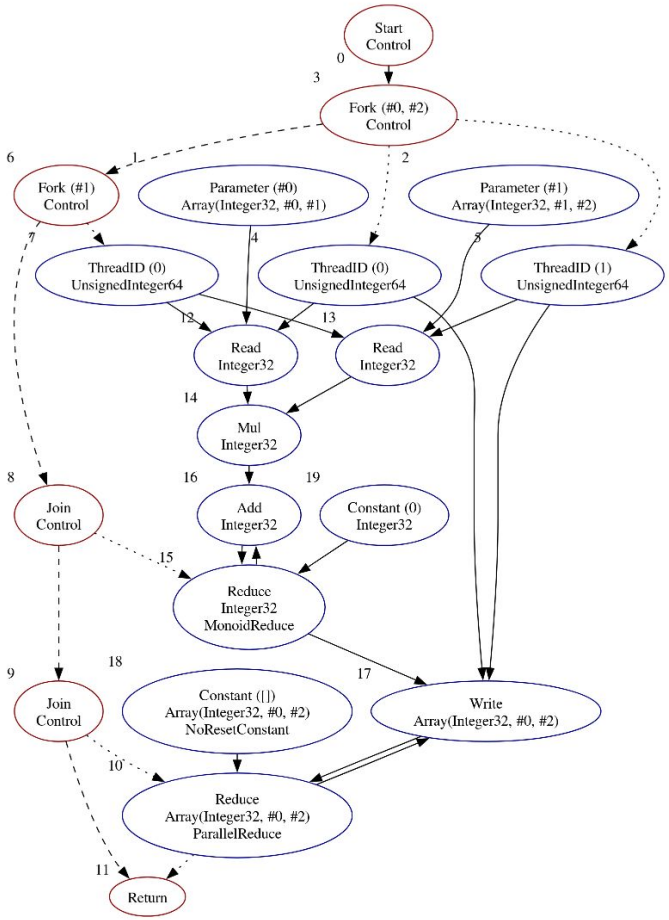


Figure 3.2: A Hercules IR function performing matrix multiplication using fork-joins.

```
#[entry]
fn matmul<n, m, l: usize>
(a: f32[n, m], b: f32[m, l]) -> f32[n, l] {
  let res : f32[n, l];
  @outer for i in 0..n {
    @middle for j in 0..l {
      @inner for k in 0..m {
        res[i, j] += a[i, k] * b[k, j];
      }
    }
  }
  return res;
}
```

1. Single language for both CPU and GPU code
2. Communication is implicit in function type signature

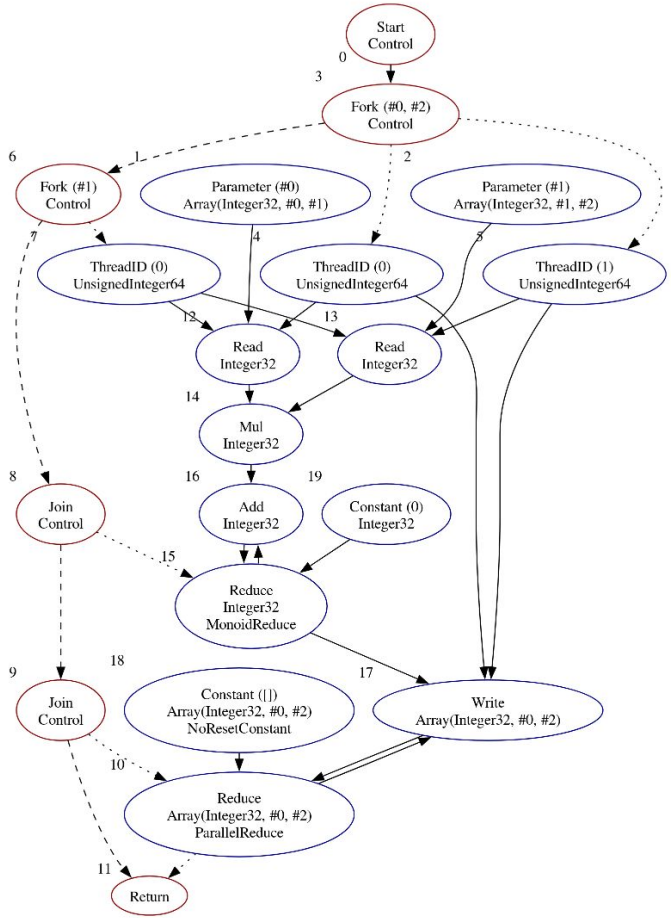


Figure 3.2: A Hercules IR function performing matrix multiplication using fork-joins.

Code Generation

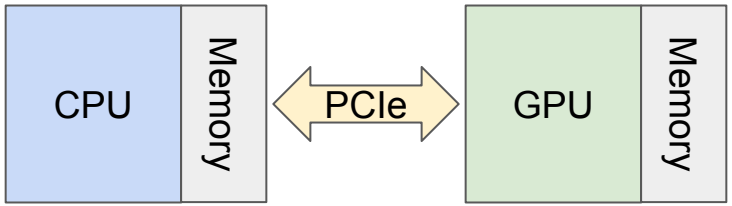
- Each Hercules function is lowered into one device function
  - Outlining splits Juno code into device functions
  - CPU backend lowers code to LLVM
  - GPU backend lowers code to CUDA
  - Intermediate functions (functions that call other functions) are lowered to async Rust
- Entry functions have async Rust interfaces generated

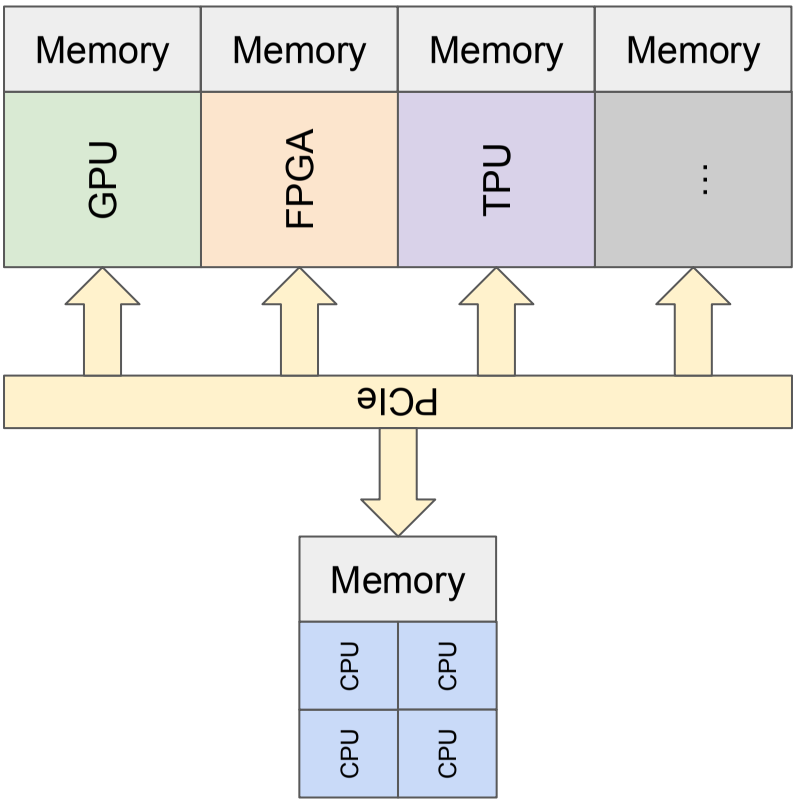
```
1  use hercules_rt::{runner, HerculesRefInto};
2  juno_build::juno!("matmul");
3  async fn invoke_matmul(
4      i: u64, j: u64, k: u64, a: &[f32], b: &[f32]
5  ) -> Box<[f32]> {
6      let mut r = runner!(matmul);
7      let c = r.run(i, j, k, a.to(), b.to()).await;
8      return c.as_slice().into();
9  }
```

Figure 6. Rust calling a matrix multiply Hercules function.

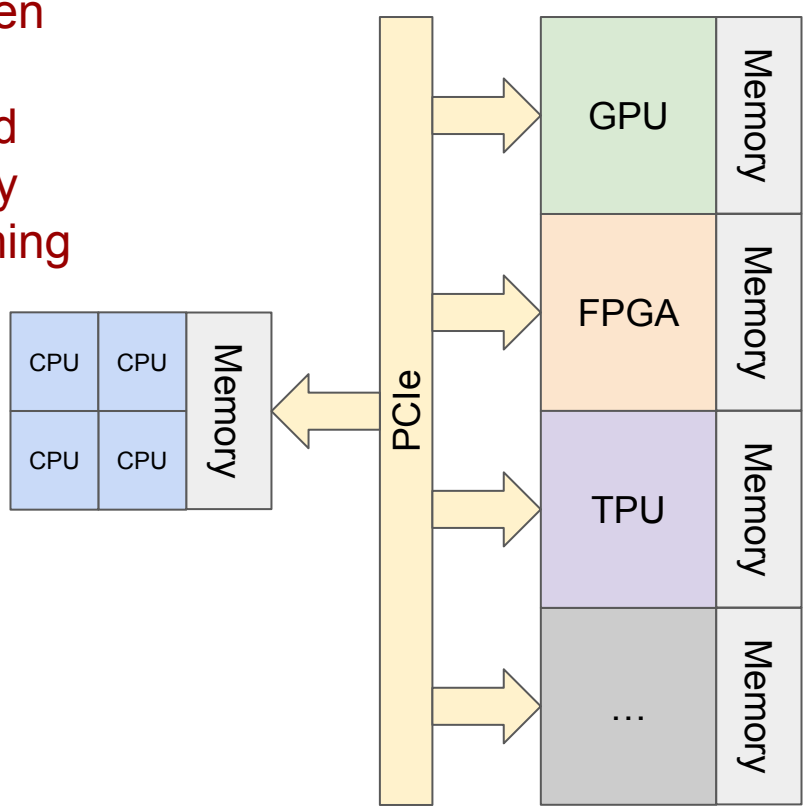
Takeaways

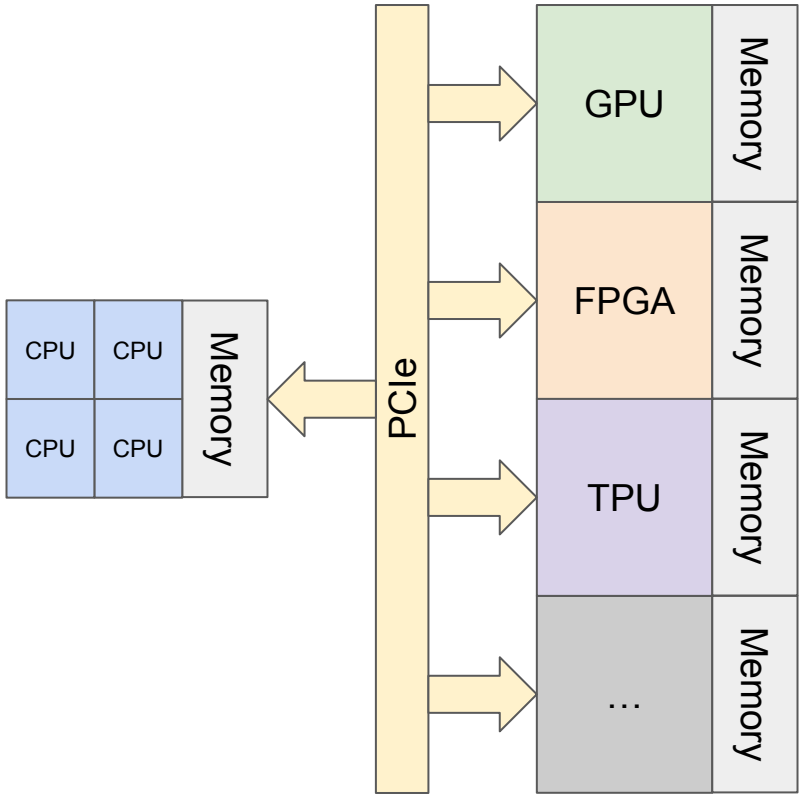
- Write application logic in a single, target agnostic language
- Use schedules to compile programs into heterogeneous device code
  - Write applications once, schedule per target machine configuration
- Competitive performance (see pre-print for details)

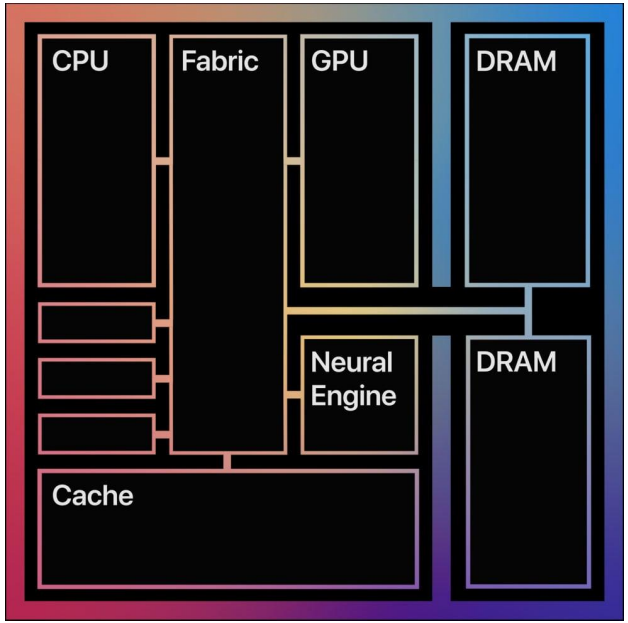

CPU
Memory
PCIe
GPU
Memory

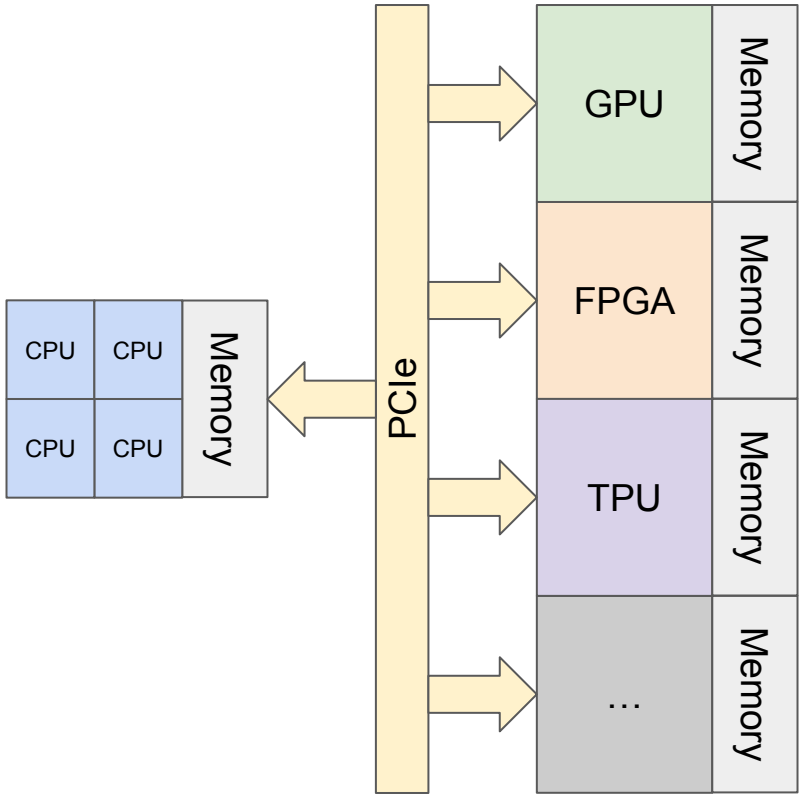

Memory
Memory
Memory
Memory
GPU
FPGA
TPU
...
PCIe
Memory
CPU
CPU
CPU
CPU

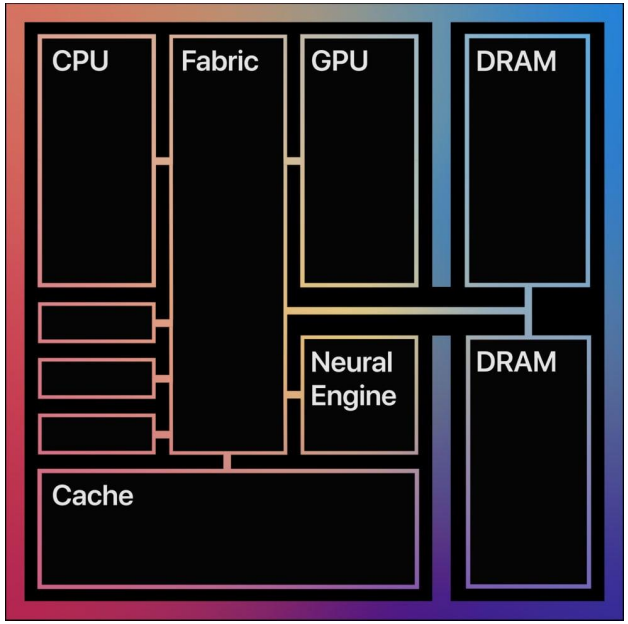
Conflation between compute heterogeneity and memory hierarchy makes programming much harder



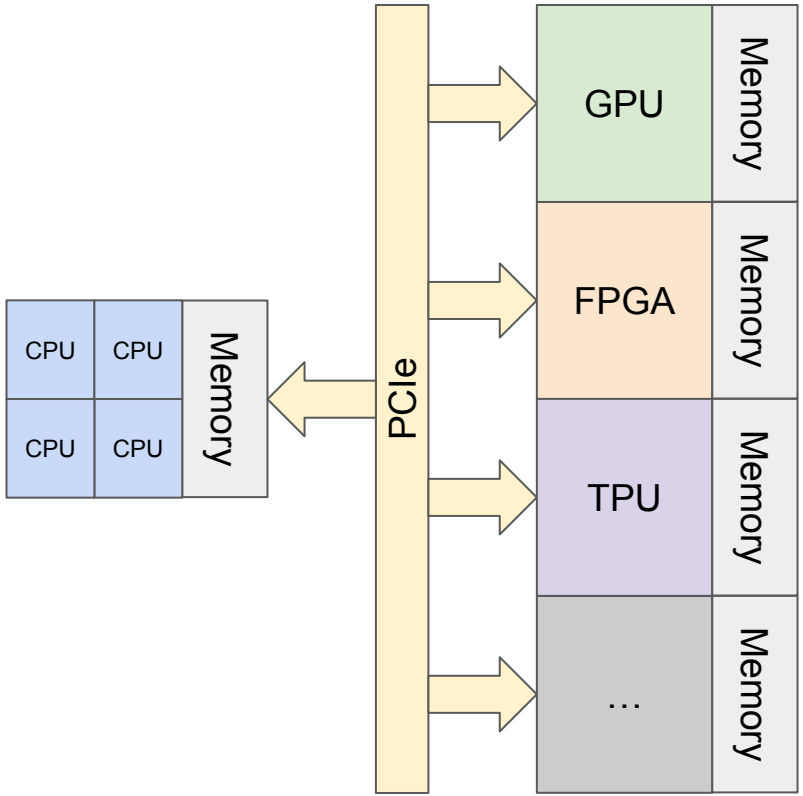

CPU
CPU
CPU
CPU
Memory
PCIe
GPU
FPGA
TPU
...
Memory
Memory
Memory
Memory

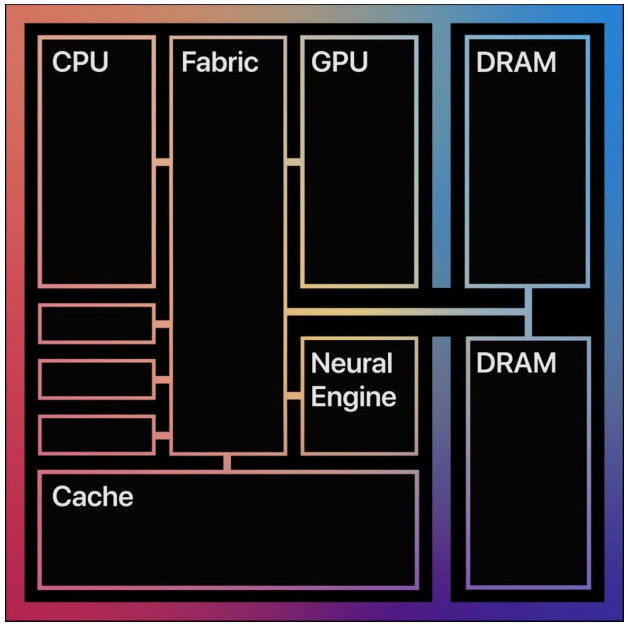

CPU
Fabric
GPU
DRAM
Neural Engine
DRAM
Cache





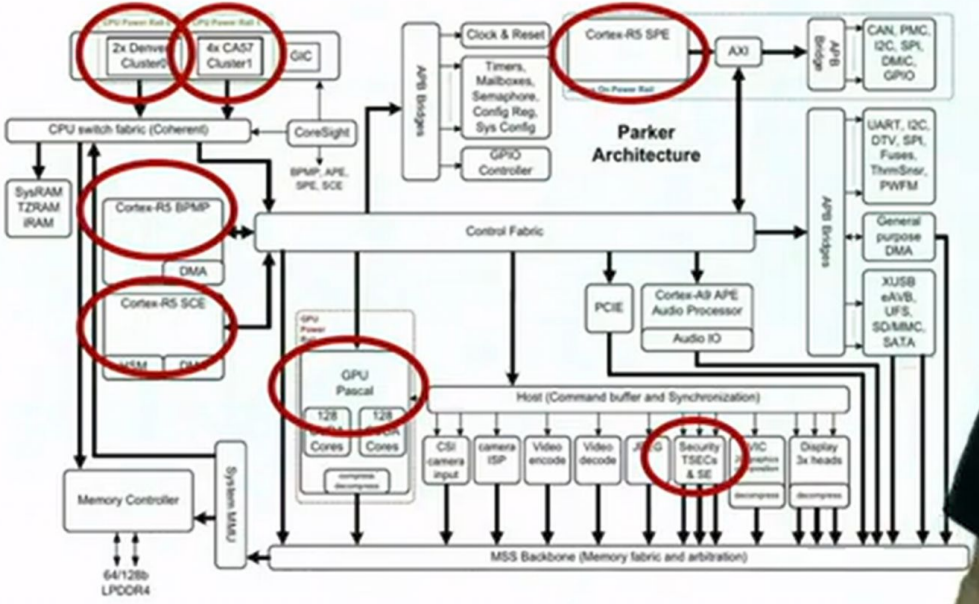
- Single coherent address space





- Single coherent address space
- Not reflected in the programming model!

Reality?


2x Denver Cluster0
4x CA57 Cluster1
GIC
CPU switch fabric (Coherent)
CoreSight
BPMP, APE, SPE, SCE
SysRAM TZRAM iRAM
Cortex-R5 BPMP
DMA
Cortex-R5 SCE
GPU Pascal
128 Cores
128 Cores
Memory Controller
System MMU
64/128b LPDDR4
APB bridges
Clock & Reset
Timers, Mailboxes, Semaphore, Config Reg, Sys Config
GPIO Controller
Cortex-R5 SPE
AXI
APB bridge
CAN, PMC, I2C, SPI, DMIC, GPIO
Parker Architecture
Control Fabric
PCIE
Cortex-A9 APE Audio Processor
Audio IO
UART, I2C, DTV, SPI, Fuses, ThrmSnsr, PWFM
General purpose DMA
XUSB eAVB, UFS, SD/MMC, SATA
Host (Command buffer and Synchronization)
CSI camera input
camera ISP
Video encode
Video decode
Security TSECs & SE
Display 3x heads
decompress
MSS Backbone (Memory fabric and arbitration)