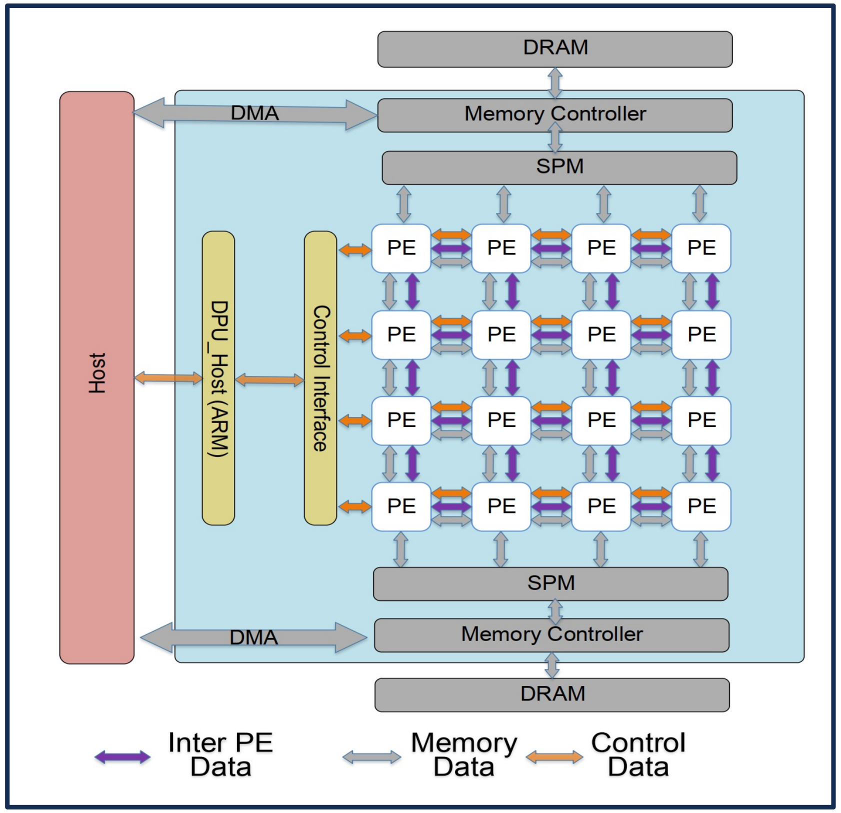
DFU: Dataflow Processing Unit



Contents

- Hardware Capabilities
  - Processor Element
  - Inter-core communication
  - Memory Hierarchy
  - Common optimizations
- CUDA-Like API
- Ideal dataflow programming Model
- Discussion

Hardware Capabilities

Hardware Capabilities: Processing Element

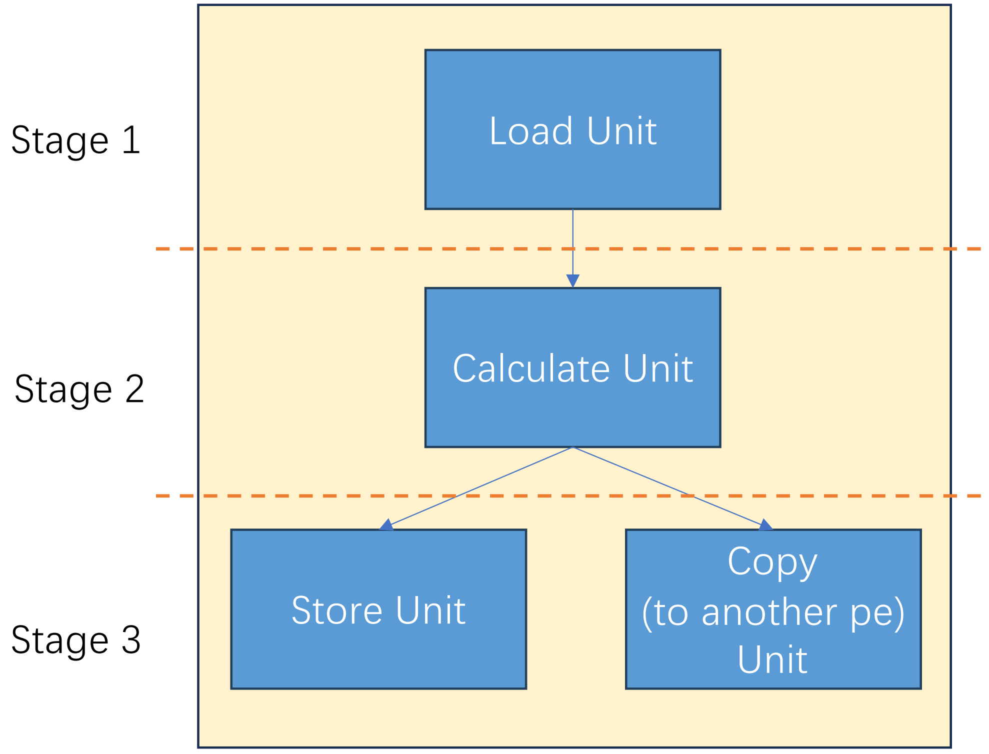
- The 16 PEs are the identical, all supporting load and store.
- You can think of PE as a single 1024-bit vector CPU core without control flow.
- All the instructions are vector instructions, which means the only way for DFU to support constant operations is to leave all other vector slots empty.

Instruction Set

- Calculate
- Load: Load from SPM (SRAM on chip)
- Store: Store to SPM.
- Flow: Send data to another PE.

FRCP↵
FSQRT↵
FSIN↵
FCOS↵
FLOG2↵
FEXP2↵

- Four types of instructions corresponds to four function units in a PE.
- They form a 3-stage pipeline (or 4-unit overlapping)



Hardware Capabilities: Processing Element

Instruction level parallelism -- Assuming there only exists one thread on one PE

```
Load X0, Offset
Load X1, Offset
ADD X1, X1, X0
Store X1, Offset

Load X2, Offset
Load X3, Offset
SUB X2, X2, X3
Copy X2, to another PE
```

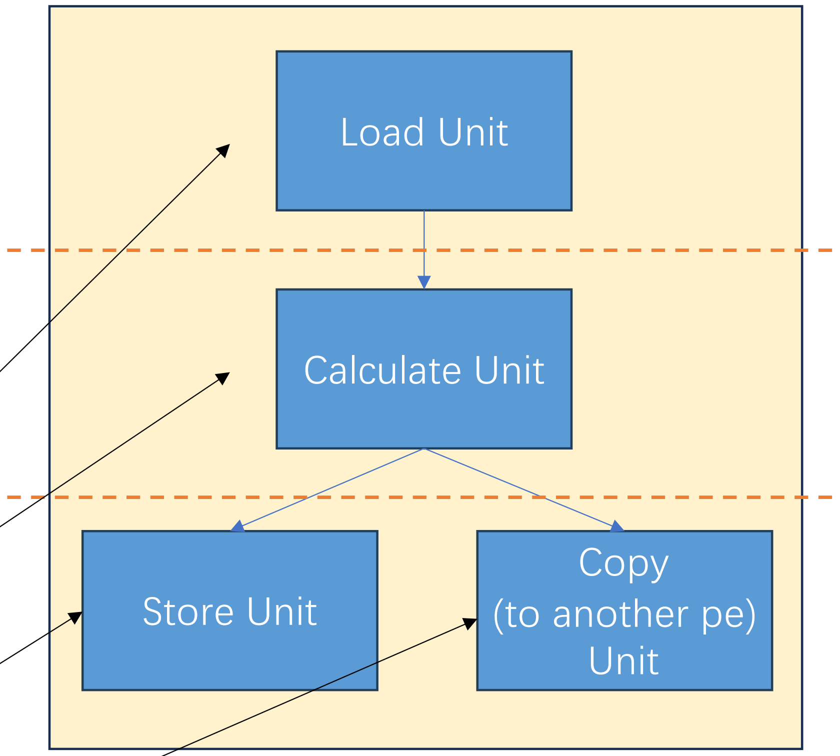
```
Load X0, Offset
Load X1, Offset
Load X2, Offset
Load X3, Offset

ADD X1, X1, X0
SUB X2, X2, X3

Store X1, Offset

Copy X2, to another PE
```

Load Unit Instruction buffer

Cal Unit Instruction buffer

Store Unit Instruction buffer

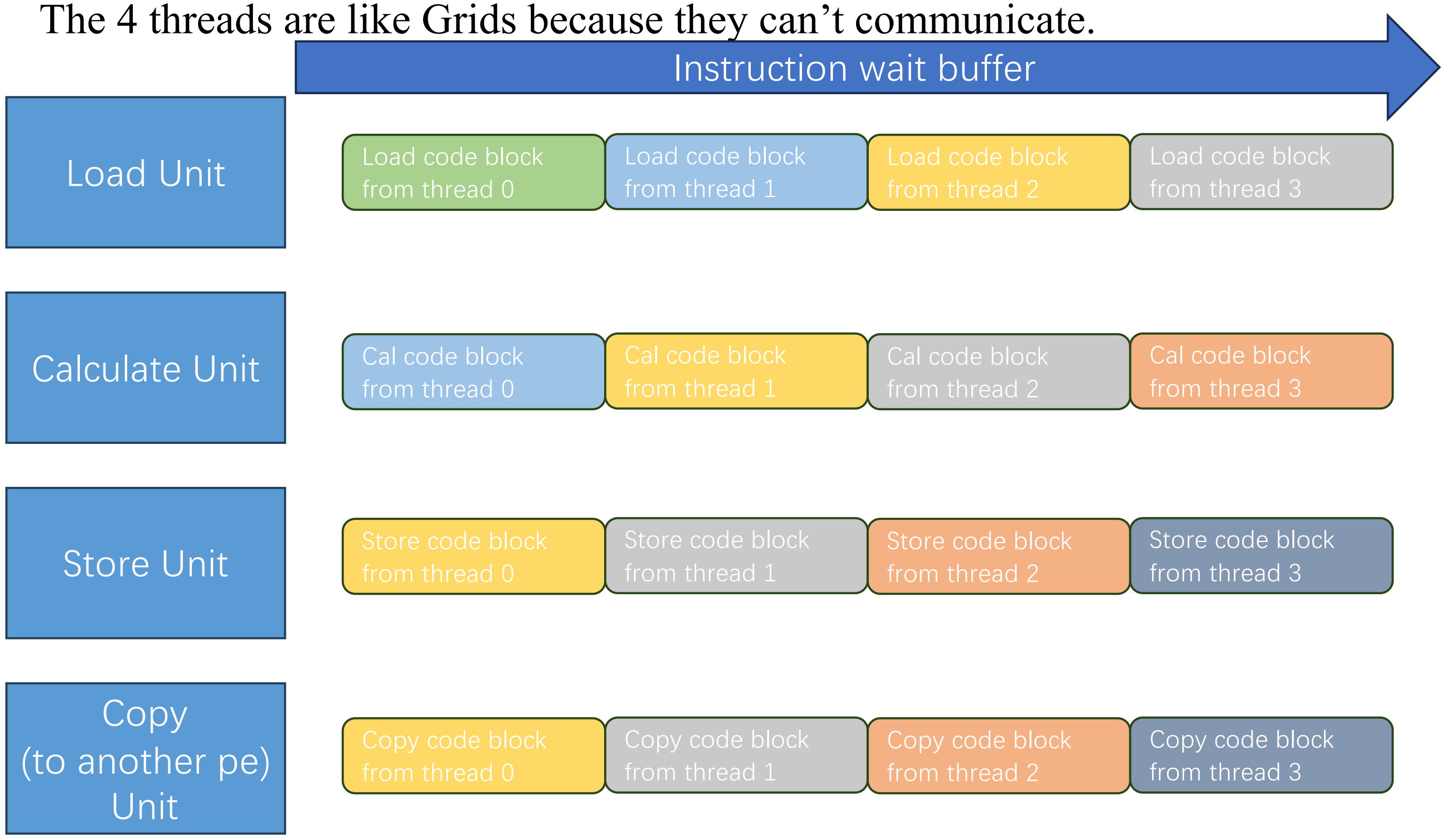
Copy Unit Instruction buffer



An instruction gets executed when its operands are ready.

Hardware Capabilities: Processing Element

Thread level parallelism – The hardware supports up to 4 physical threads(tasks) running on a PE core. The 4 threads are like Grids because they can't communicate.

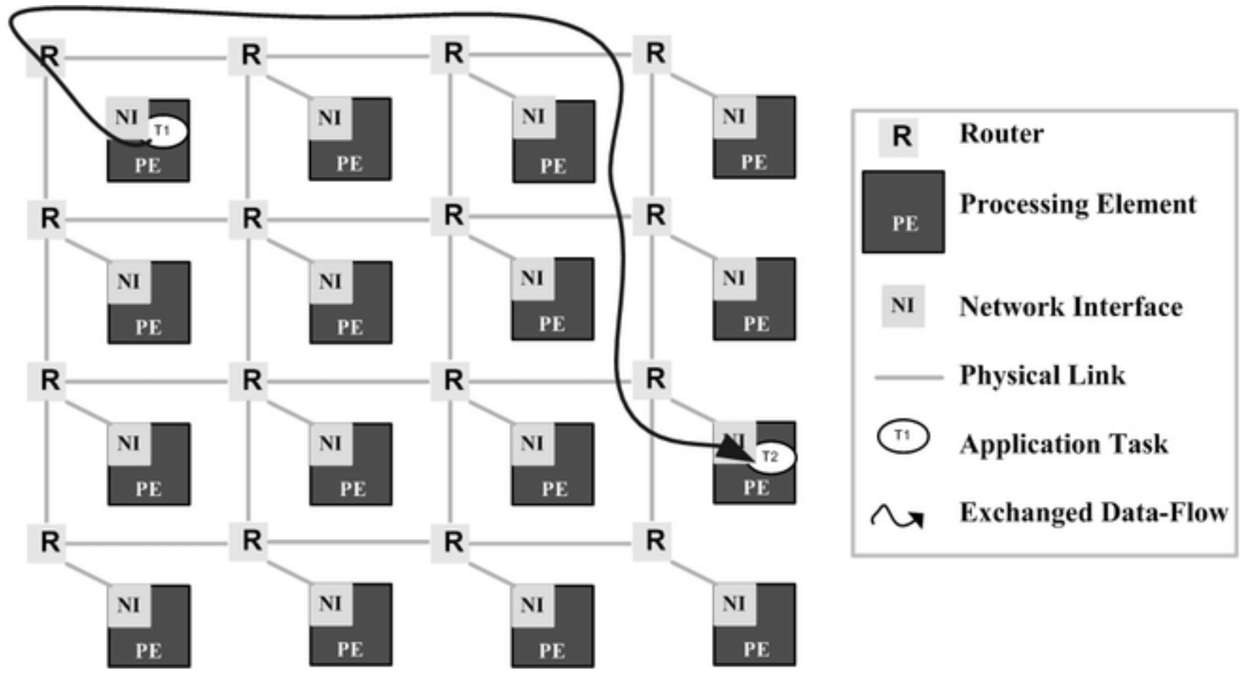

Instruction wait buffer
Load Unit
Calculate Unit
Store Unit
Copy (to another pe) Unit
Load code block from thread 0
Load code block from thread 1
Load code block from thread 2
Load code block from thread 3
Cal code block from thread 0
Cal code block from thread 1
Cal code block from thread 2
Cal code block from thread 3
Store code block from thread 0
Store code block from thread 1
Store code block from thread 2
Store code block from thread 3
Copy code block from thread 0
Copy code block from thread 1
Copy code block from thread 2
Copy code block from thread 3

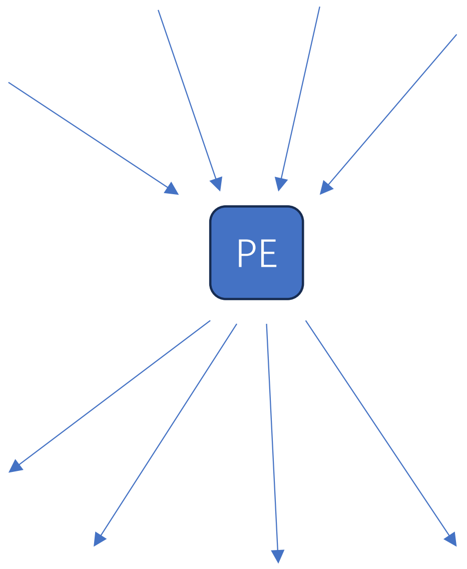
- Assuming that the execution time of each code block in every component of each thread is the same, it can be understood as a pipeline of thread instruction blocks.

- If copy takes the same time as store, PEs communicating is similar to GPU thread communicating through Shared Memory.
- Actually, copy runs faster than store so we an do many copys.

Hardware Capabilities: Inter-core communication

Thread level parallelism – PEs communicate through 2D mesh on chip.



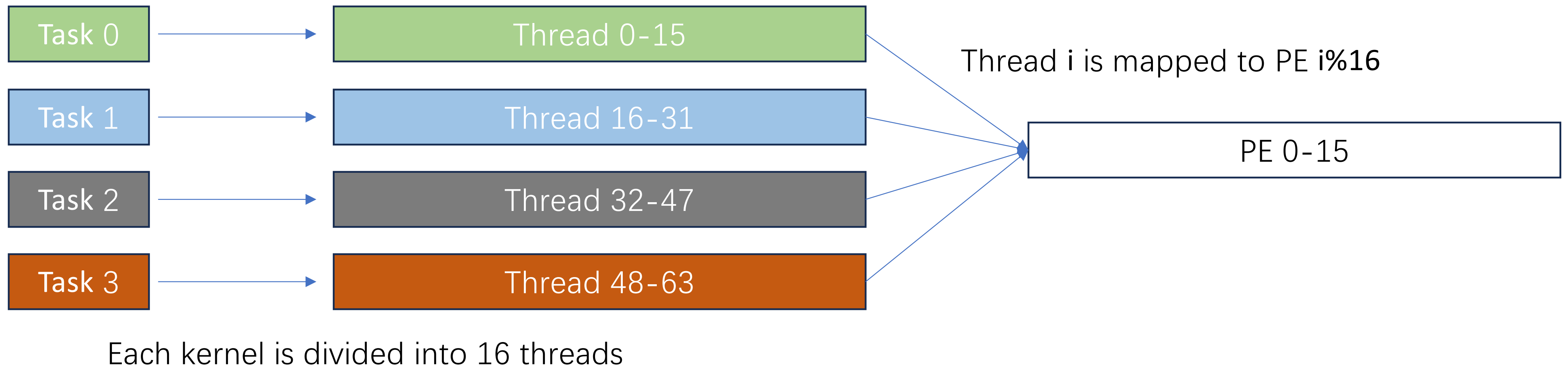


Rapid communication between PEs is a major advantage of DFU compared to others, as inter-CPU core communication occurs through shared memory while DFU can utilize on-chip mesh.

A PE can have up to 4 out-edges and 4 in-edges only through mesh.

Hardware Constraints: Task can’t communicate

- There are 64 threads runs simultaneously on 16 PEs.
- But you can't assume this is the same as mapping 64 threads onto 16 1024b-wide vector CPUs that support 4-thread time-division multiplexing.



- The 64 threads are divided into 4 tasks (jargon of DFU). Tasks are like Grids and there is no way for them to communicate.
- From the view of a single PE, 4 thread from 4 tasks runs simultaneously on it.
- The reason why more than 4 tasks are not supported is that ID of the each task is stored in hardware.

Hardware Constraints: Hardware loop for range(1024)

- The hardware supports loop for range(1024), which means each iteration in the hardware loop must be independent.

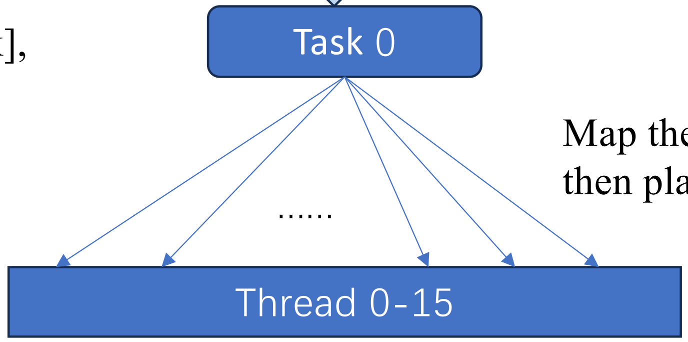
```
for(int i=0; i<512; i++) {
    for(int j=0; j<1024; j++) {
        for(int k=0; k<512; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

- Select one to be vectorized, which be divided by factor 32. i

```
for(int j=0; j<1024; j++) {
    for(int i=0; i<16; i++) {
        for(int k=0; k<512; k++) {
            C[i, ⋯, i+31][j] += A[i, ⋯, i+31][k] * B[k][j];
        }
    }
}
```

The for loop of j is totally handled by hardware

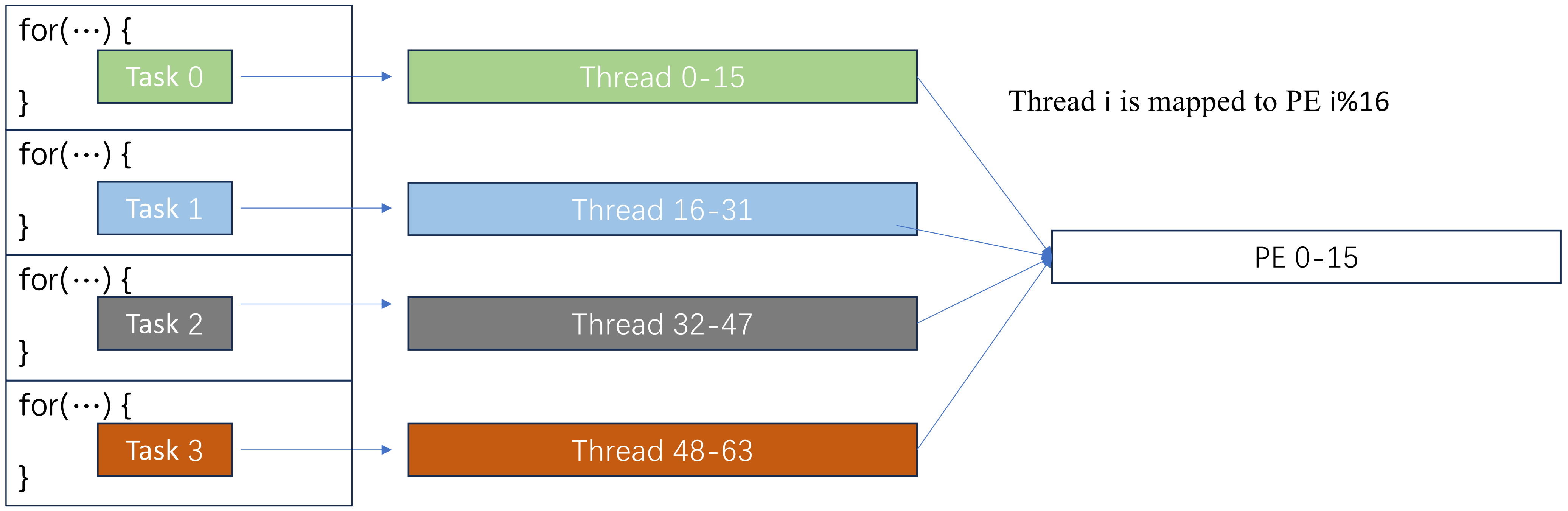
- 32 operands in a vector, such as A[i, …, i+31][k], will be place continuously in SPM by DMA.



Map the loop body to 16 threads and then place them on 16 PEs.

Hardware Constraints: Hardware loop for range(1024)

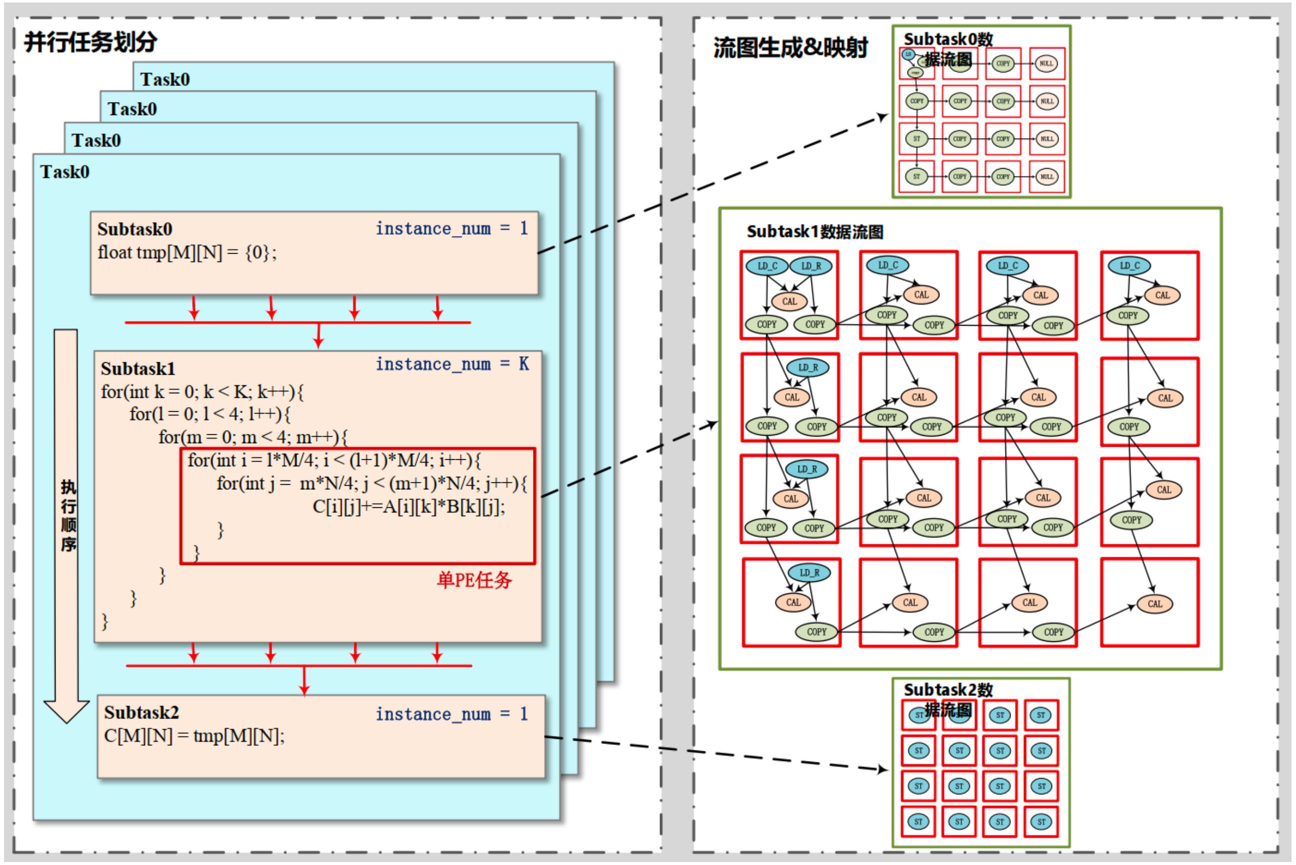
- Hardware will automatically execute the same code segment for multiple times.
- The hardware loop idx is global variables for all 16 PEs.
- The 16 PEs enters the next iteration at the same time.



- If there is no such loop suitable for hardware loop, the DFU will do a for loop for range (1) and early exit.

Hardware Constraints: Barrier by restart

- In an invocation of DFU, synchronization cannot be achieved.
- If it is necessary to use a barrier, a restart of the DFU must be initiated.



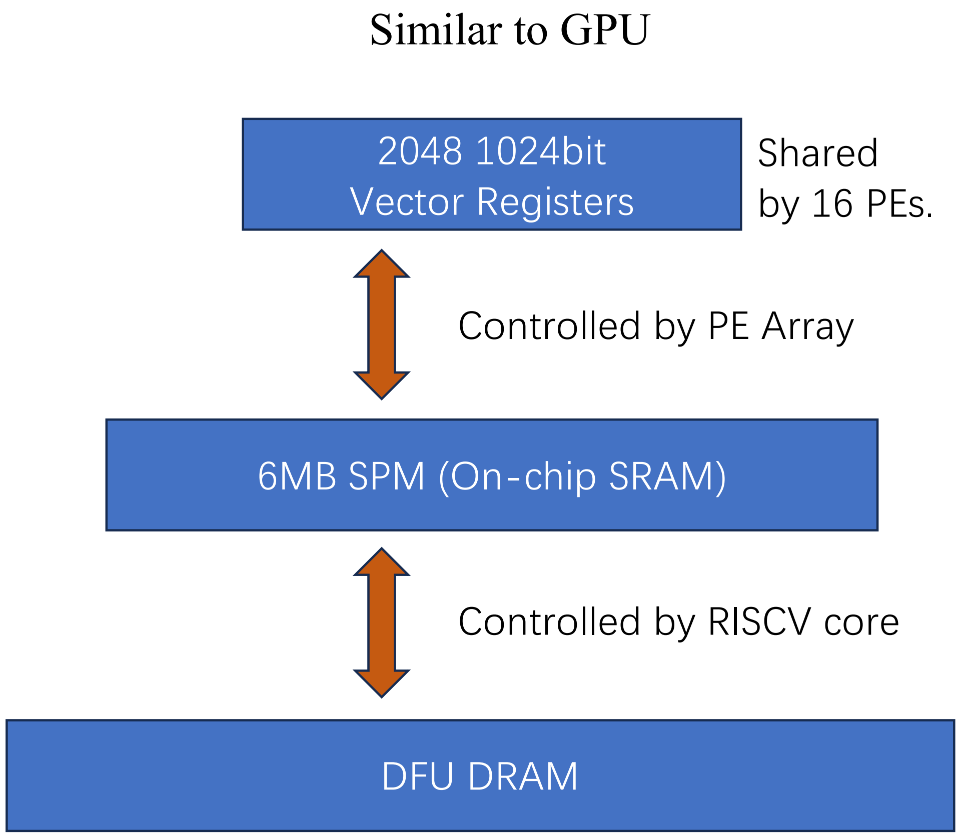
Jargon of DFU:

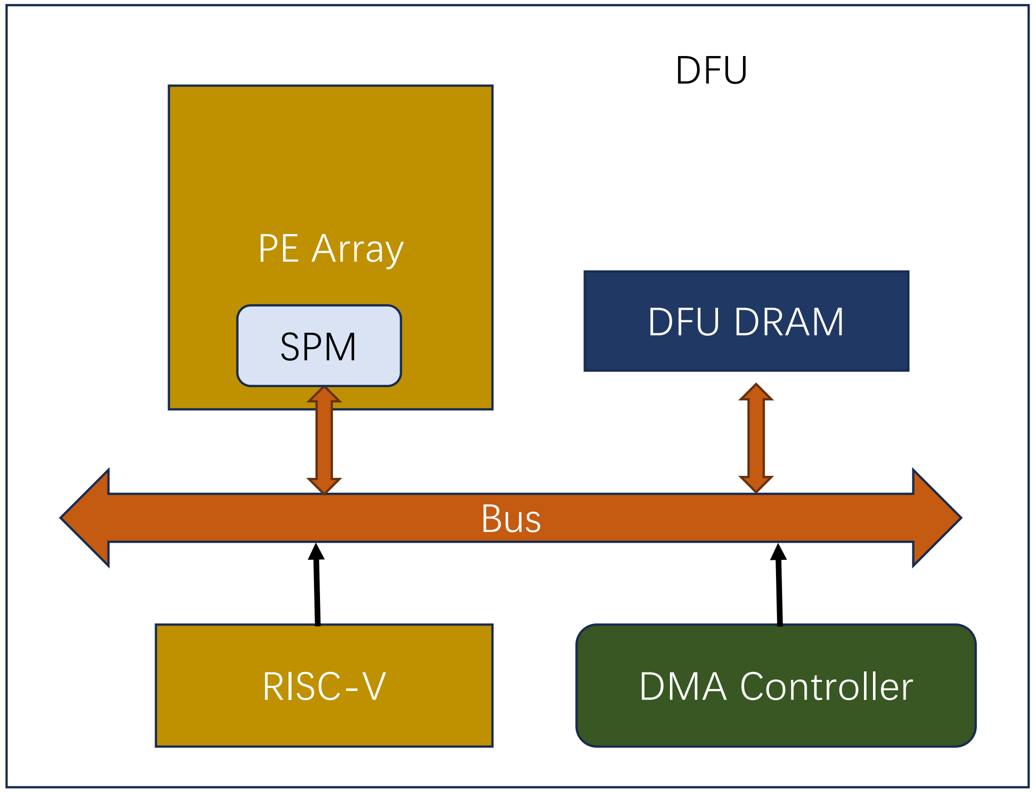
1. Instance: One iteration of the hardware loop.
2. Subtask: Code segment with no barrier

Refresh only at first launch:

- The on-chip registers and SPM of DFU are refreshed only at the first boot.
- If the RISCV Core boots the DFU twice, the state of DFU at the end of the first boot can be retrieved during the second boot.

Hardware Capabilities: Memory Hierarchy

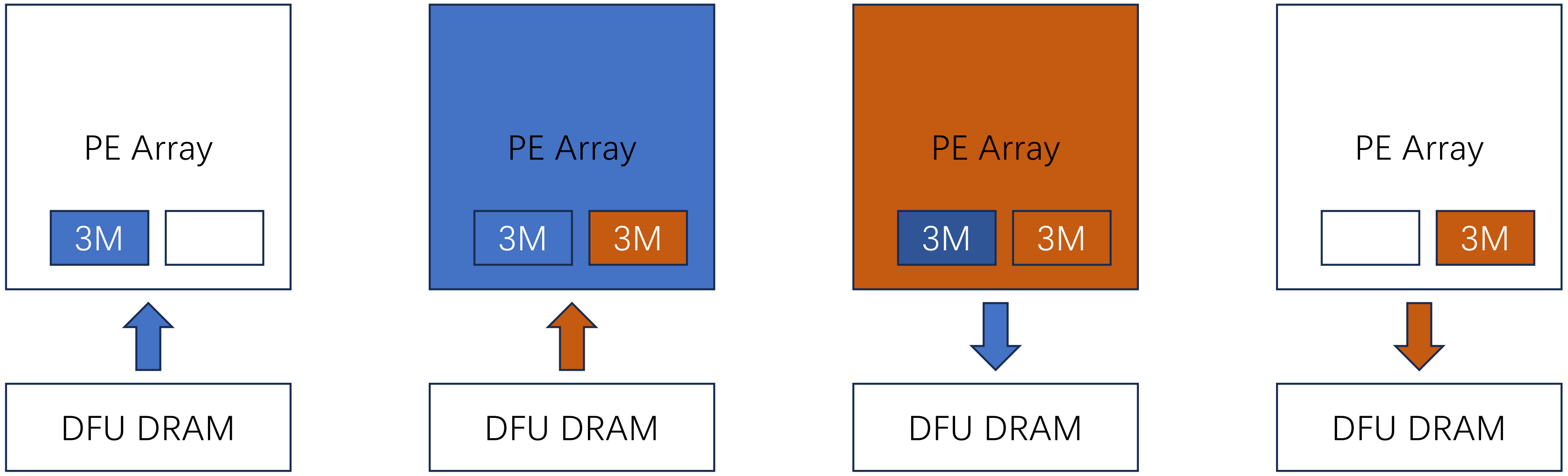




Common Optimizations

1. Double buffering

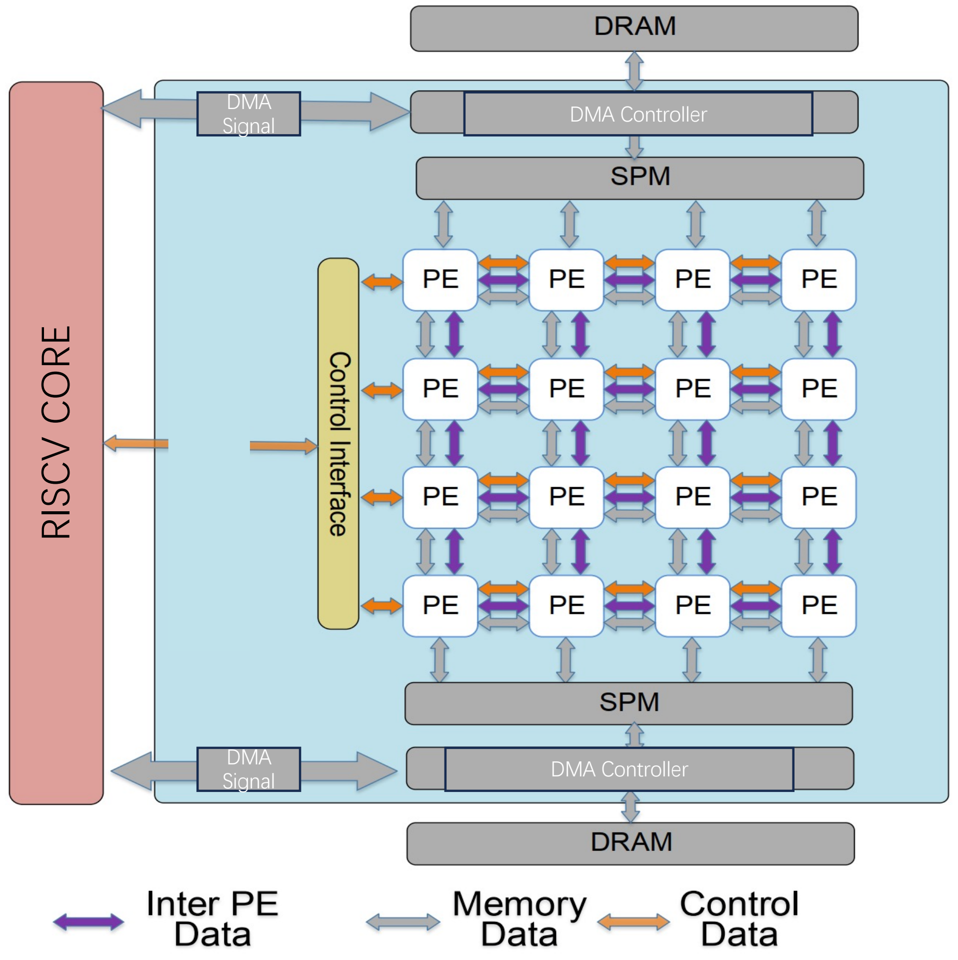
DMA between DRAM and SPM can be overlapped.
Double-buffer optimizations can be applied when PE Array needs to be launched twice.



2. One load and multiple copy

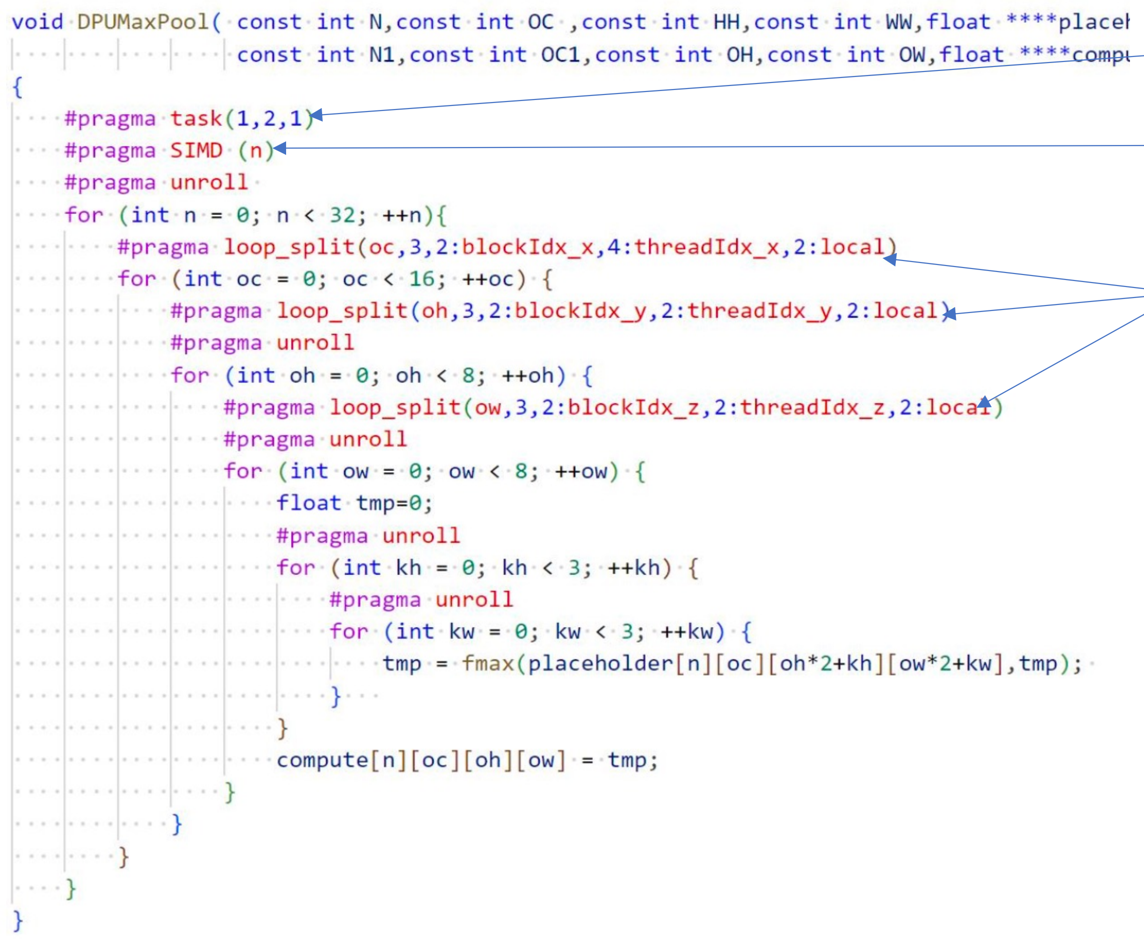
If multiple PEs need the same data, only one PE on the edge do actual load and it broadcasts the data to other PEs using copy.

Hardware Capabilities: A high level view



CUDA-Like API

CUDA-Like API



```
void DPUMaxPool( const int N,const int OC ,const int HH,const int WW,float ****placeh
                 const int N1,const int OC1,const int OH,const int OW,float ****compu
{
    #pragma task(1,2,1)
    #pragma SIMD (n)
    #pragma unroll
    for (int n = 0; n < 32; ++n){
        #pragma loop_split(oc,3,2:blockIdx_x,4:threadIdx_x,2:local)
        for (int oc = 0; oc < 16; ++oc) {
            #pragma loop_split(oh,3,2:blockIdx_y,2:threadIdx_y,2:local)
            #pragma unroll
            for (int oh = 0; oh < 8; ++oh) {
                #pragma loop_split(ow,3,2:blockIdx_z,2:threadIdx_z,2:local)
                #pragma unroll
                for (int ow = 0; ow < 8; ++ow) {
                    float tmp=0;
                    #pragma unroll
                    for (int kh = 0; kh < 3; ++kh) {
                        #pragma unroll
                        for (int kw = 0; kw < 3; ++kw) {
                            tmp = fmax(placeholder[n][oc][oh*2+kh][ow*2+kw],tmp);
                        }
                    }
                    compute[n][oc][oh][ow] = tmp;
                }
            }
        }
    }
}
```

First, configure the number of physical tasks.

Then, select the hardware loop.

After that, select one or two or three loops and define ThreadBlocks by splitting them.

Finaly, mark all the others loops as unroll.

Mapping to PE Array

- Block: executes one by one.
- Thread: mapped to PE Array in a %16 fashion. 64 threads launch PE Array 4 times.
- Local: the actual loop iterations done by a single thread. They must be unrolled and done one by one.

CUDA-Like API

- In conclusion, the CUDA-like API view the PE array as one streaming multiprocessor with 4 active warp and 512(16 PE * 32 Vector) floating point units.
- Dataflow occurs only in data loading if using *one load multiple copy* optimization.
- No dataflow at runtime.

Ideal Dataflow Programming Model

Ideal Dataflow Programming Model

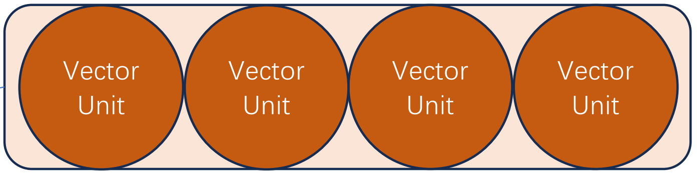
Algorithm

$$C_i = A_i + B_i$$

Shapes of A, B, C are all (1024,)

Programmer only knows the algorithm and an abstraction of hardware.

Hardware abstraction



The mapping of algorithm to hardware abstraction is done by the programmer.

Manually optimized Code

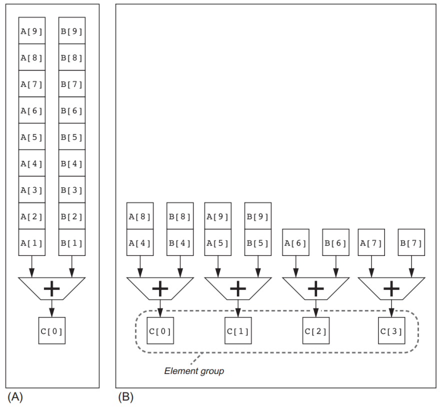
```
for(int i=0;i<1024;i+=4){
   C[i] = A[i] + B[i];
   C[i+1] = A[i+1] + B[i+1];
   C[i+2] = A[i+2] + B[i+2];
   C[i+3] = A[i+3] + B[i+3];
}
```

```
for(int i=0;i<1024;i+=4){
   VectorAdd_Float_4(C[i], A[i], B[i]);
}
```

Hardware Execution

```
Loop:
        ...... Prologue ......
fadd C[i],  A[i], B[i]
fadd C[i+1], A[i+1], B[i+1]
fadd C[i+2], A[i+2], B[i+2]
fadd C[i+3], A[i+3], B[i+3]

add i, i, 4
blt loop, i, 4
```

```
Loop:
        ...... Prologue ......
vectorAdd C[i],  A[i], B[i]

add i, i, 4
blt loop, i, 4
```



Read hardware

The high level API can be seen as:

1. a wrapper of multiple assembly instructions to help coding. (From the view of the programmer).
2. the hardware abstraction. This is how hardware designers hope the hardware appears in the minds of software developers. (From the view of the hardware designer)

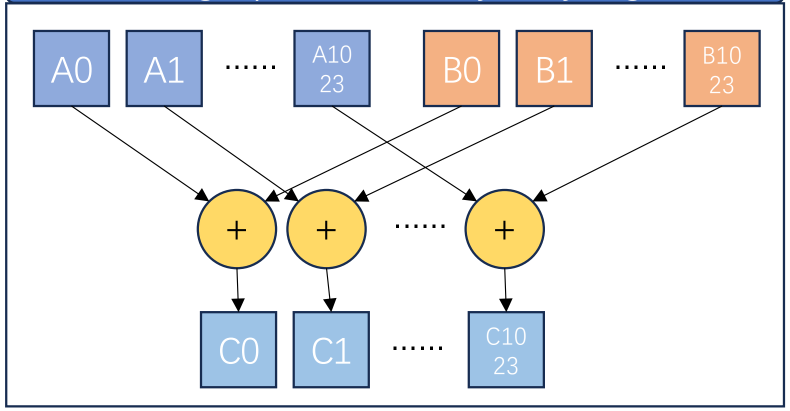
Ideal Dataflow Programming Model

Algorithm

$$C_i = A_i + B_i$$

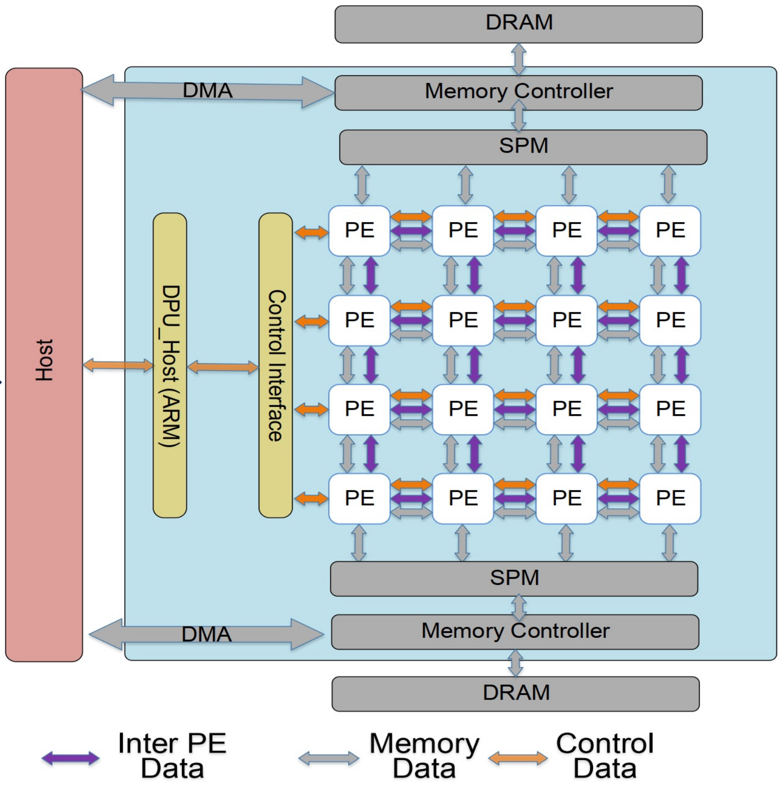
Shapes of A, B, C are all (1024,)

Dataflow graph defined by only algorithm



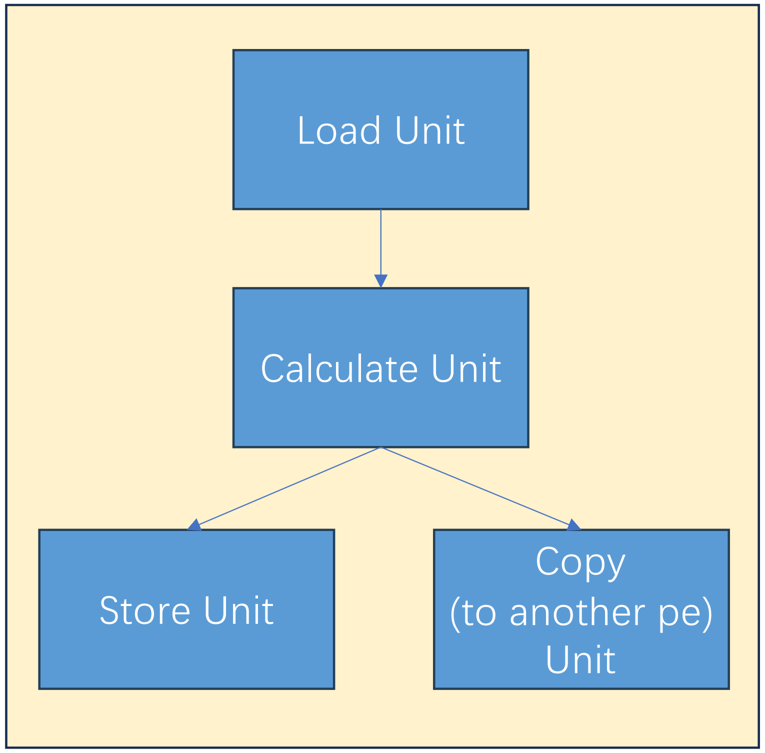
Hardware abstraction (DFU-Level ISA)

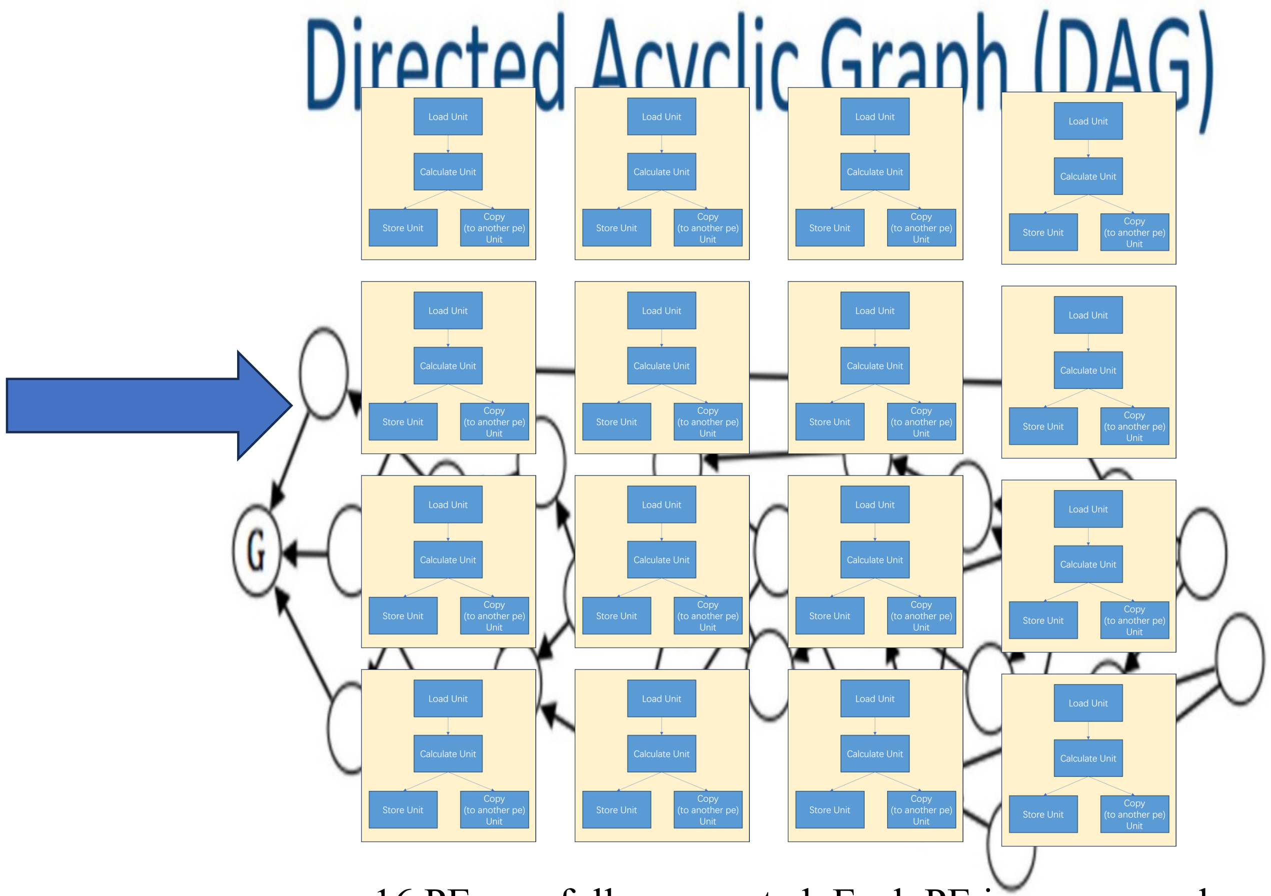
Read hardware



Ideal Dataflow Programming Model

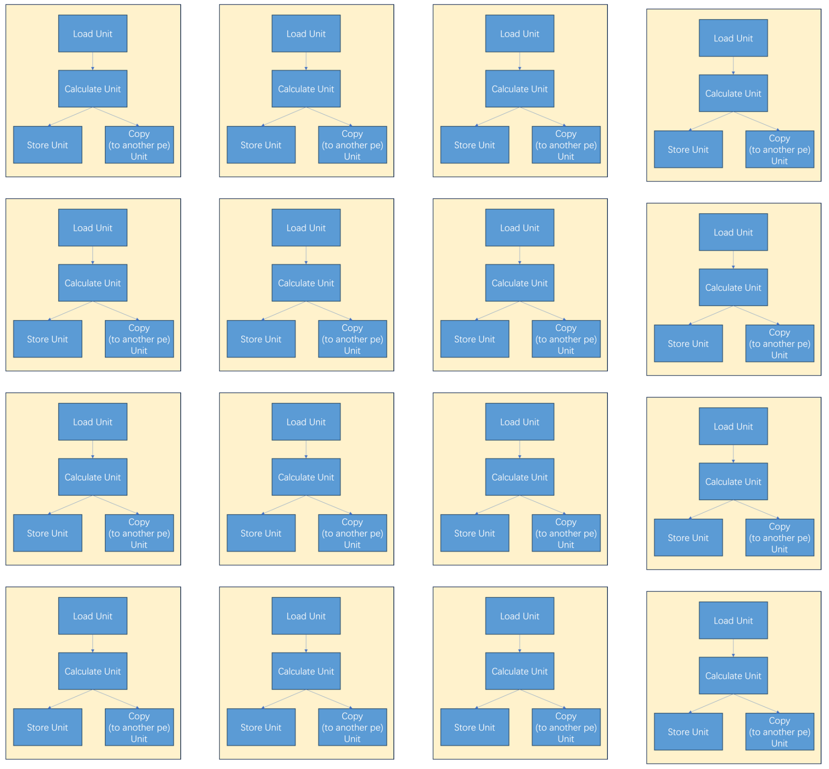
Dataflow inside single PE

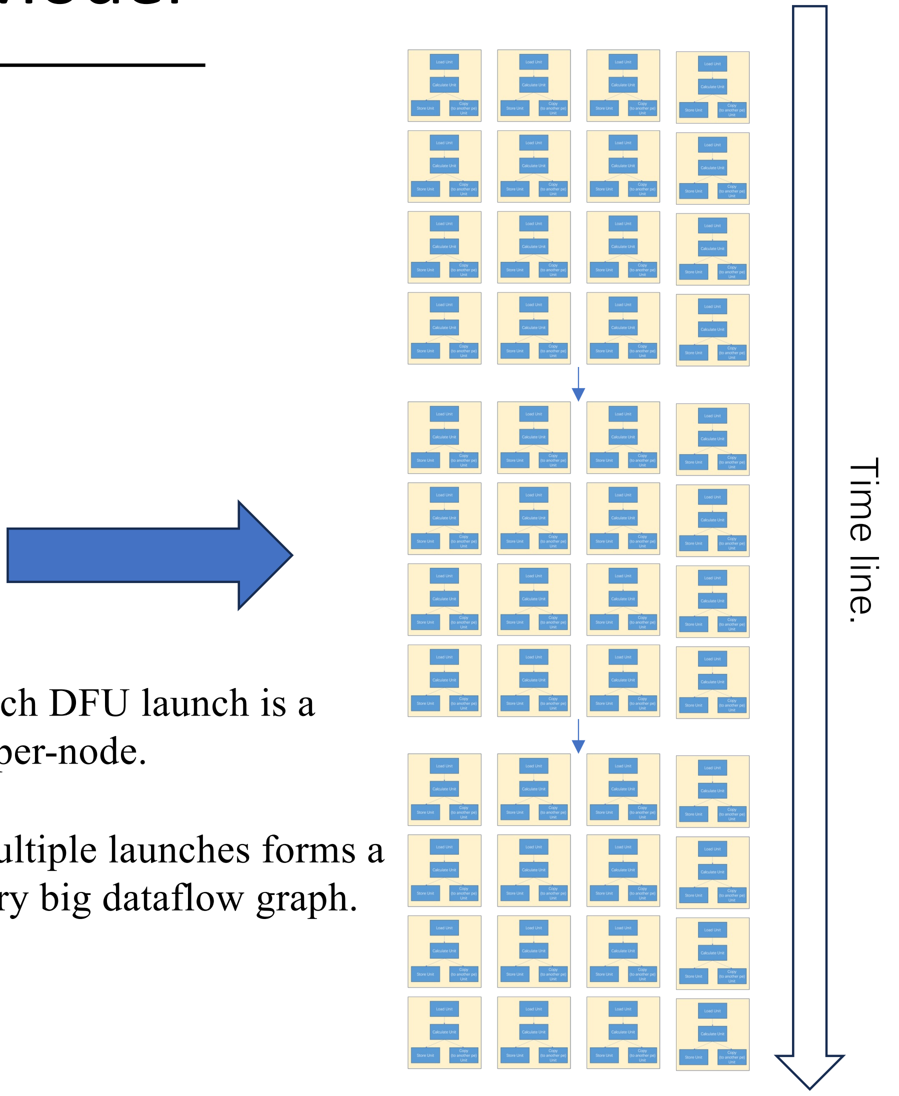




16 PEs are fully connected. Each PE is a super node.

Ideal Dataflow Programming Model

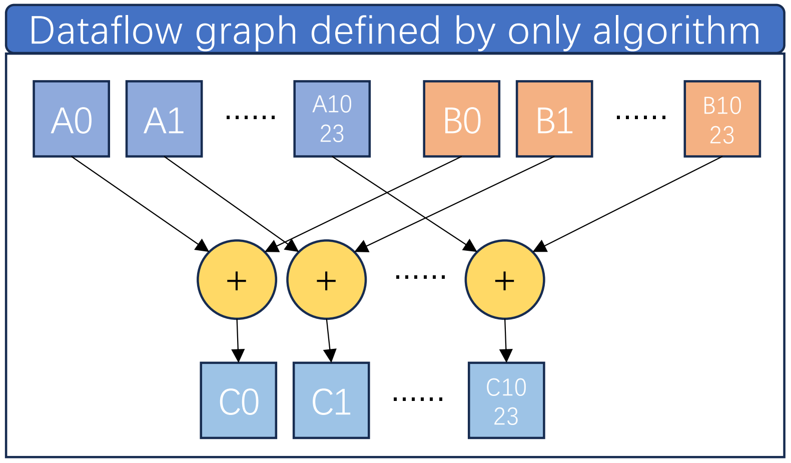


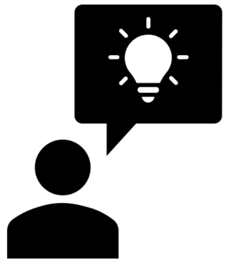


Each DFU launch is a super-node.

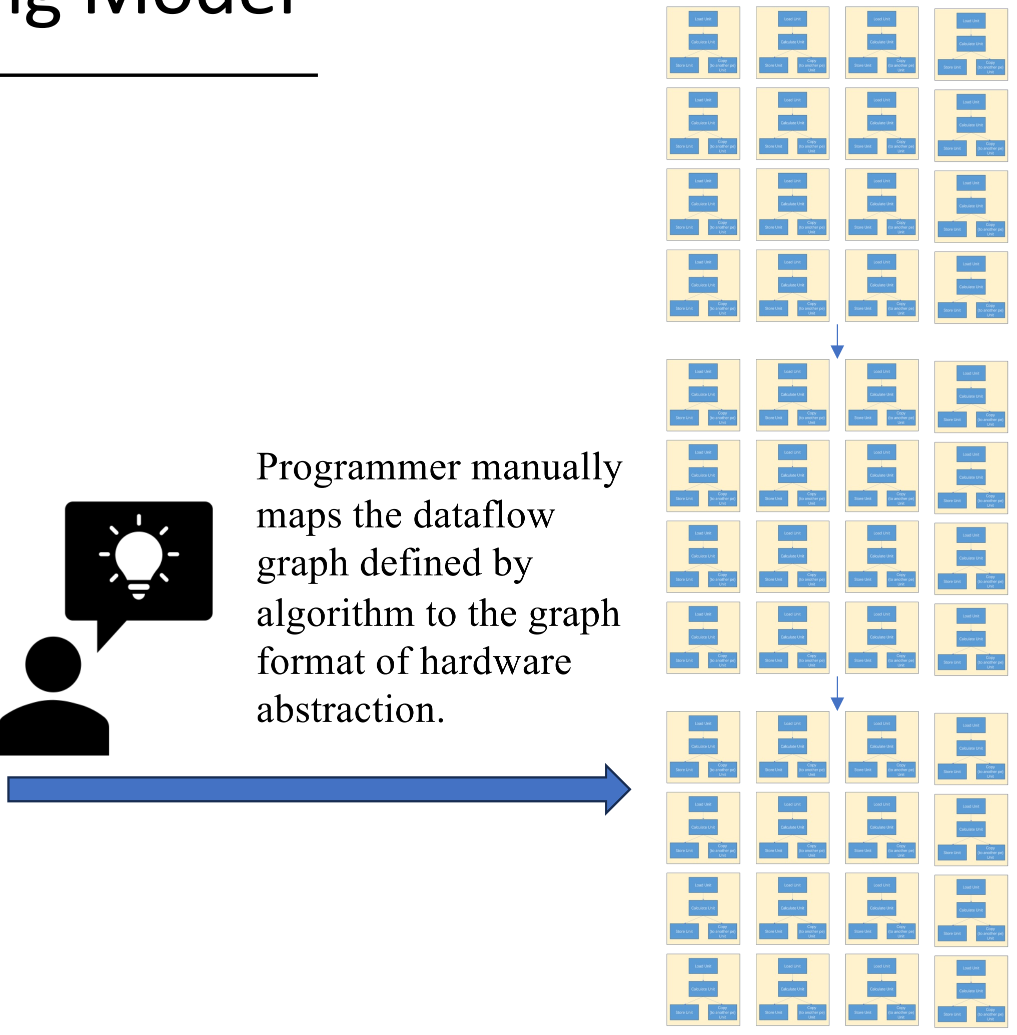
Multiple launches forms a very big dataflow graph.

Ideal Dataflow Programming Model





Programmer manually maps the dataflow graph defined by algorithm to the graph format of hardware abstraction.



Hardware abstraction

Thanks!