Lien vers les slides ⇨

FPGA : du registre à DOOM

Rémy Citérin

Au menu

- Ray-tracing
- DOOM
- UNIX
- GPU

Accélérateur de Ray-Tracing

- Pipeline + FSM + réseaux systoliques
- Une dizaine de FPS
- Parcours d'arbre avec Bounding Volume Hierarchy

Accélérateur de Ray-Tracing

- $320 * 240 = 76800$ pixels
- 240 triangles, on a besoin de calculer 18 millions d'intersections par image
- 19 additions, 27 multiplications et une division par intersections

Accélérateur de Ray-Tracing

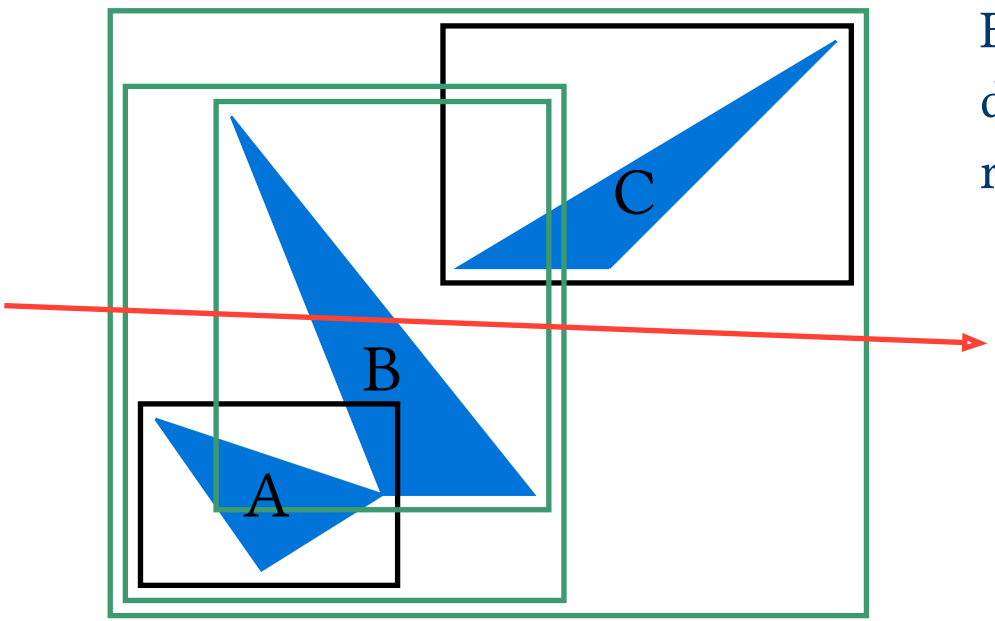
- $320 * 240 = 76800$ pixels
- 240 triangles, on a besoin de calculer 18 millions d'intersections par image
- 19 additions, 27 multiplications et une division par intersections

On overflow le nombre de multiplieurs du FPGA même pour avoir 10 FPS...

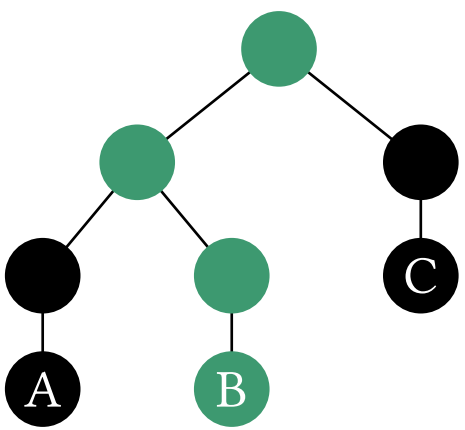
On a besoin de:

1. Meilleurs algorithmes pour économiser des intersections
2. Algorithmes d'intersections plus efficaces (9 multiplications dans mon cas)

Bonding Volume Hierarchy

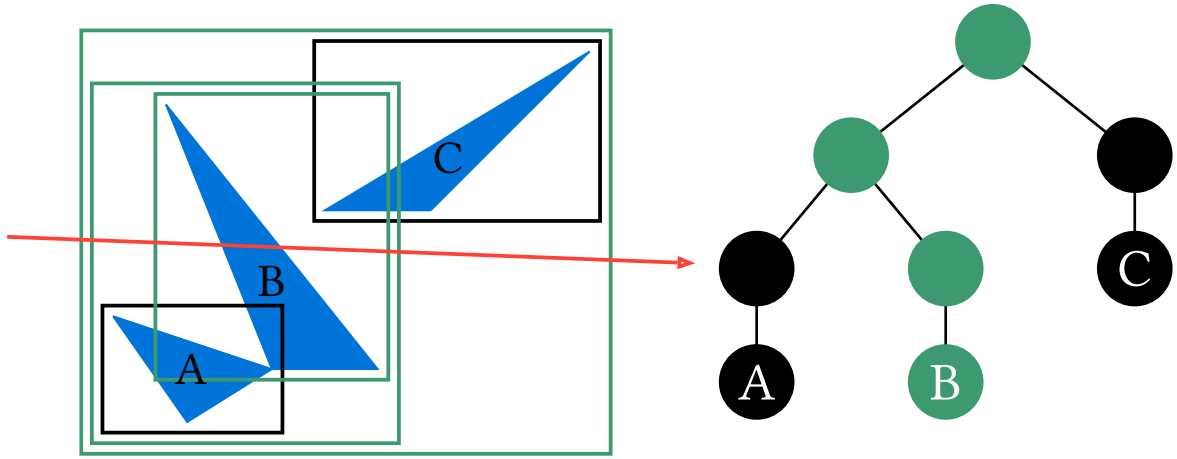


En utilisant 3 intersections rayon/box on peut déterminer que seulement une des intersections rayon/triangle est nécessaire au lieu de 3!



Bonding Volume Hierarchy

```
def intersect_tree(root, rayon):
  currentHit = inf
  stack = [root]

  while !stack.empty():
    node = stack.pop()
    t = node.box.intersect(rayon)

    if t < currentHit and node.leaf():
      currentHit =
        min(currentHit, node.tri.intersect(rayon))

    if t < currentHit and !node.leaf():
      stack.append(node.rightChild)
      stack.append(node.leftChild)
```



time	stack	node	currentHit	t
t_0	{ABC}	ABC	inf	5
t_1	{AB, C}	C	inf	inf
t_2	{AB}	AB	inf	5
t_3	{A, B}	B	inf	6
t_4	{A}	A	7	inf

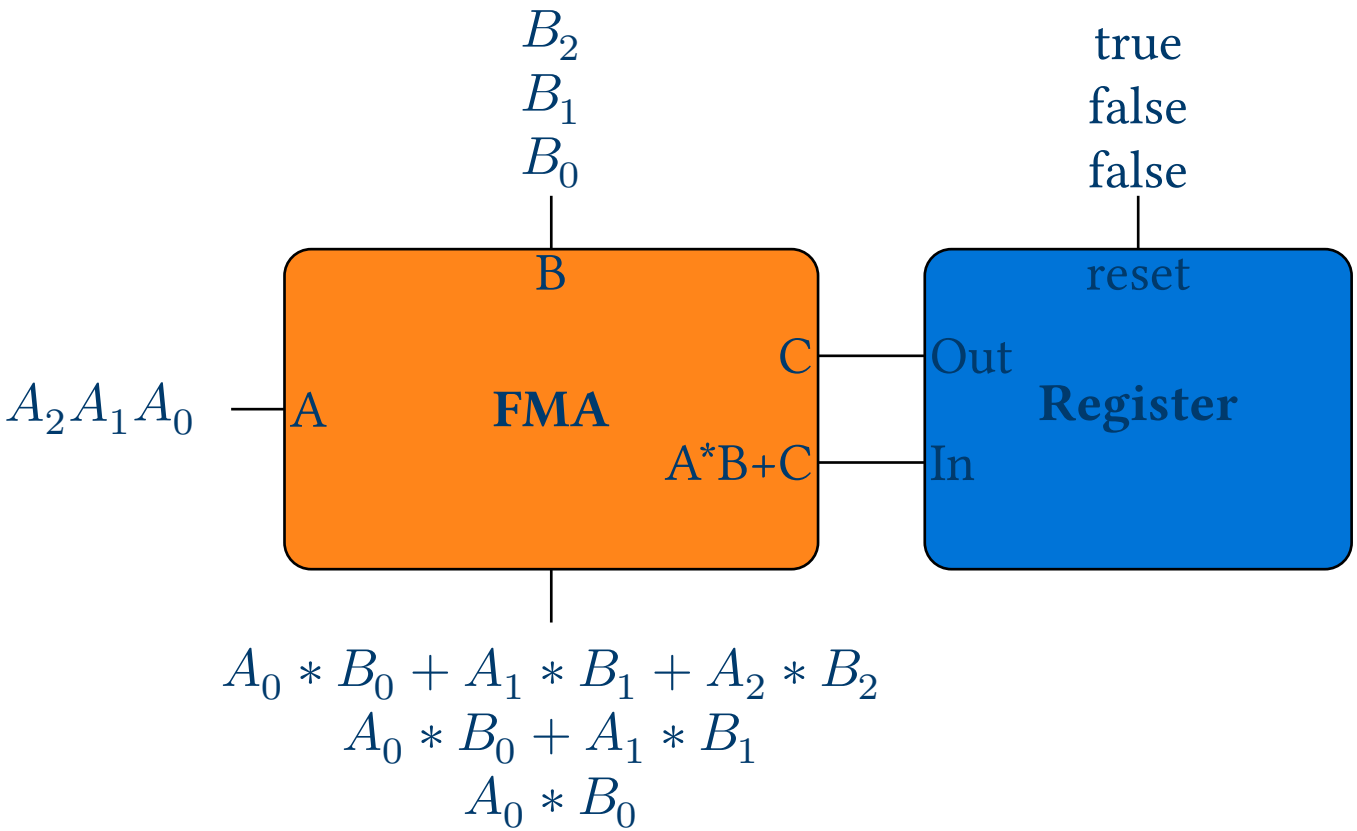
Bonding Volume Hierarchy

```
while (notDone) seq
  node <= nodes.read;
  tmin <= rayonNodeIntersection(node, ray);

  action
    if (tmin < currentHit.t && node.isLeaf)
      intersectQ.enq(tuple2(node.firstTri, node.length));

    if (tmin < currentHit.t && !node.isLeaf) begin
      nodes.readRequest(node.leftChild+1);
      stack.push(node.leftChild);
    end

    // Backtrack
    if (tmin >= currentHit.t || node.isLeaf) begin
      notDone <= !stack.empty;
      if (!stack.empty) begin
        nodes.readRequest(stack.read);
        stack.pop;
      end
    end
  endaction
endseq
```

Möller-Trumbore

```
void intersect(triangle_t* tri, ray_t* ray) {
  float3 edge1 = tri->vertex1 - tri->vertex0;
  float3 edge2 = tri->vertex2 - tri->vertex0;
  float3 h = cross3(ray->direction, edge2);
  float a = dot3(edge1, h);
  float f = 1.0 / a;
  float3 s = ray->origin - tri->vertex0;
  float u = f * dot3(s, h);
  float q = cross3(s, edge1);
  float v = f * dot3(ray->direction, q);
  float t = f * dot3(edge2, q);

  // (u,v,1-u-v) : barycentric coordinate of the intersection in the triangle
  // origin + ray->direction * t : coordinate of the intersection
  ...
}
```

Réseau systolic : produit scalaire en hardware



Réseau systolic : produit scalaire en hardware

```
Reg#(Vector#(3,F16)) x1 <- mkReg(replicate(?));
Reg#(Vector#(3,F16)) x2 <- mkReg(replicate(?));
Fifo#(2, F16) outputs <- mkFifo;
Reg#(Bit#(3)) reset <- mkReg(0);
Reg#(F16) acc <- mkReg(0);

rule step if (reset != 0);
  let fma = acc + x1[0] * x2[0];
  acc <= reset[0] == 1 ? 0 : fma;
  reset <= reset >> 1;
  x1 <= rotate(x1);
  x2 <= rotate(x2);

  if (reset[0] == 1) outputs.enq(fma);
endrule
```

```
method request(Vec3 lhs, Vec3 rhs) if (reset == 0);
  x1 <= vec(lhs.x, lhs.y, lhs.z);
  x2 <= vec(rhs.x, rhs.y, rhs.z);
  reset <= 3'b100;
endmethod

interface response = toGet(outputs).get;
```

DOoOM Out of Order Machine

Un RISC-V out-of-order avec :

- Multiplication/Division
- Flotants
- Prédiction de branche
- Store buffer/Load queue (spéculation des dépendances)
- Caches L1i et L1d
- VGA, SD-CARD, UART
- 32MB de SDRAM

▹ En bref

- $\approx$ 9000 lignes de Bluespec
- $\approx$ 30MHz sur un EPC5
- $\approx$ 30K 4-LUT sur un EPC5
- Coremark : 2.7

Finite state machine (FSM)

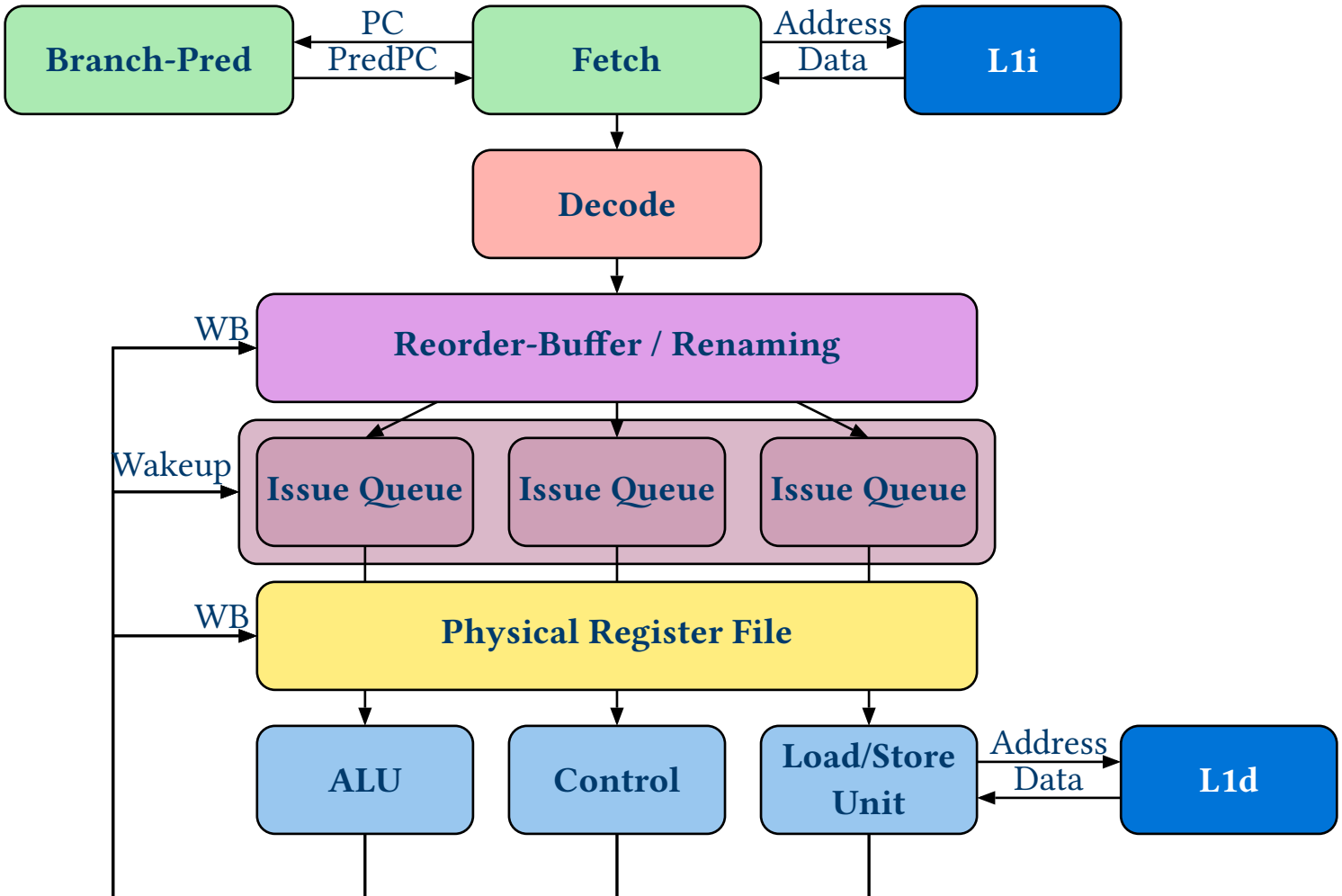
FSM	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
add t0, a1, a2	F	D	Rr	Ex	Wb								
lw t1, 0(t0)						F	D	Rr	Ex	Ex	Wb		
mul t2, a1, a2												F	D

Pipeline (In order)

Pipelined	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
add t0, a1, a2	F	D	Rr	Ex	Wb								
lw t1, 0(t0)		F	D		Rr	Ex	Ex	Wb					
mul t2, a1, a2			F	D		Rr	Ex	Ex	Ex	Wb			
add t3, t0, a2				F	D		Rr	Ex			Wb		
addi t3, t3, 42					F	D					Rr	Ex	Wb

Out of order (OOO)

Out-of-order	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
add t0, a1, a2	F	D	I	Ex	Wb	C						
lw t1, 0(t0)		F	D	I	I	Ex	Ex	Wb	C			
mul t2, a1, a2			F	D	I	Ex	Ex	Ex	Wb	C		
add t3, t0, a2				F	D	I	Ex	Wb			C	
addi t3, t3, 42					F	D	I	I	Ex	Wb		C


Branch-Pred
PC
PredPC
Fetch
Address
Data
L1i
Decode
WB
Reorder-Buffer / Renaming
Wakeup
Issue Queue
Issue Queue
Issue Queue
WB
Physical Register File
ALU
Control
Load/Store Unit
Address
Data
L1d

3DRiscV : Un Soc avec CPU + GPU

Un RISC-V in-order avec :

- Multiplication/Division
- Prédiction de branche
- Caches L1i et L1d cohérents avec le GPU
- MMU (Testé avec UNIX-v6)
- VGA, SD-CARD, UART
- 32MB de SDRAM

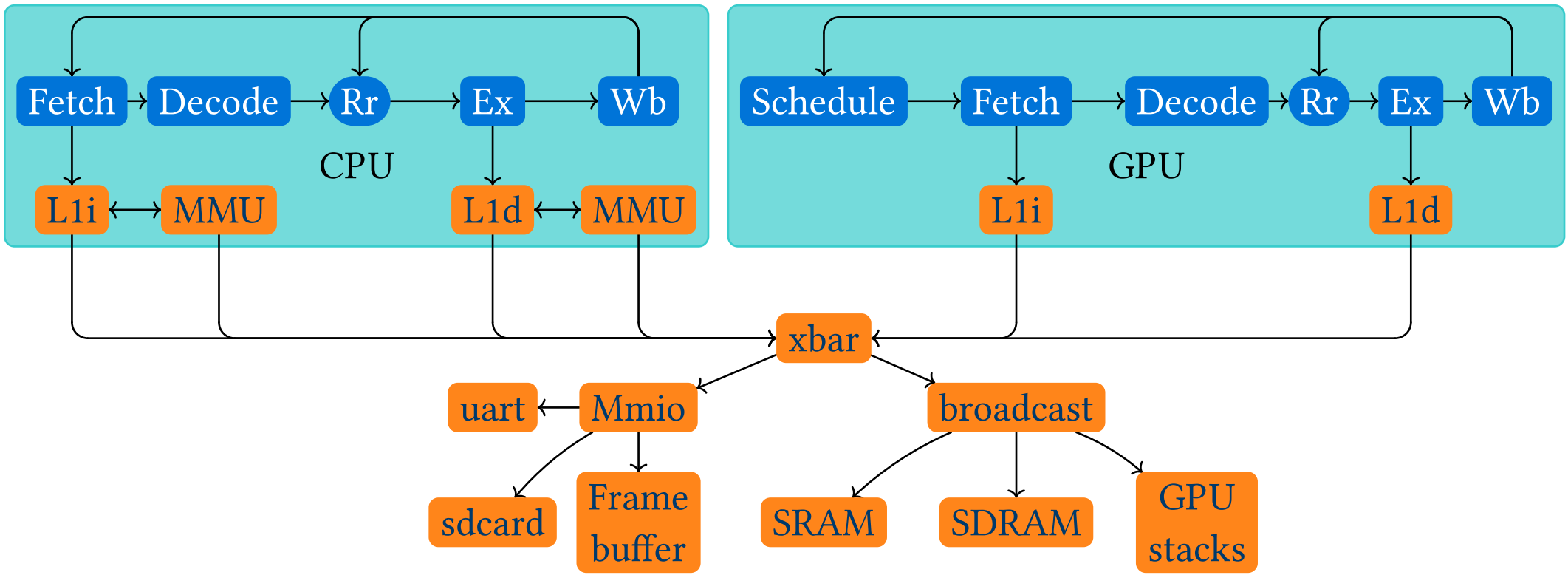
Un GPU basé sur RISC-V avec :

- 16 warp de 4 threads
  (max 4 instructions par cycle)
- Multiplication/Division
- Fusion des opérations mémoire
- Caches L1i et L1d

▹ En bref

- $\approx$ 8500 lignes de Haskell +
  $\approx$ 6700 lignes de Rust pout le compilateur (optimisant) du GPU
- $\approx$ 30MHz sur un EPC5
- $\approx$ 50K 4-LUT sur un EPC5

3DRiscV : Un Soc avec CPU + GPU



GPU : streaming machine

On exécute 16 **warp** de 4 threads en même parallèle avec ***le même pointeur d'instruction***

- Jusqu'à 4 instructions peuvent retourner en même temps
- Pas de dépendance entre les registres

Instruction	warp/mask	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
add t0, a1, a2	0/0101	S	F	D	Rr	Ex	Wb				
lw t1, 0(t0)	1/1111		S	F	D	Rr	Ex	Ex	Ex	Wb	
add t0, a1, a2	0/1010			S	F	D	Rr	Ex	Wb		
div t0, a1, a2	13/0001				S	F	D	Rr	Ex	Ex	Wb

GPU : streaming machine

Entrée :

```
foo(x, y, z) {
  if (x == 0) {
    y = y + 1;
  } else {
    z = z + 1;
  }
  y = y / z;
  return y;
}
```

Programme compilé :

```
global foo
foo:
  bnez a0, .else
  addi a1, a1, 1
  j .end_if
.else:
  addi a2, a2, 1
.end_if:
  div a0, a1, a2
  ret
```

cycle	mask	Instruction
0	1111	bnez a0, .else
1	0101	addi a2, a2, 1
2	1010	addi a1, a1, 1
3	0101	j .end_if
33	1010	div a0, a1, a2
63	0101	div a0, a1, a2

GPU : reconvergence de threads

Problème : Après une boucle / un if, les threads n'ont plus les mêmes pointeurs d'instruction!

GPU : reconvergence de threads

Problème : Après une boucle / un if, les threads n'ont plus les mêmes pointeurs d'instruction!

On ajoute des instructions pour resynchroniser les threads après un if/for/while :

```
foo(x,y,z) {                    ⇒   global foo
  if (x == 0) {                     foo:
    y = y + 1;                        push_level      ; Divergence point
  } else {                            bnez a0, .else
    z = z + 1;                        addi a1, a1, 1
  }                                   j .end_if
  y = y / z;                      .else:
  return y;                         addi a2, a2, 1
}                                 .end_if:
                                    pop_level       ; Convergence point
                                    div a0, a1, a2
```

puis il suffit d'executer seulement les threads dont le level est maximale :

Les threads qui finissent en premier le if attendent les autres

GPU : streaming machine

Entrée :

```
foo(x, y, z) {
  if (x == 0) {
    y = z + 1;
  } else {
    z = z + 1;
  }
  y = y / z;
  return y;
}
```

Programme compilé :

```
global foo
foo:
  push_level
  bnez a0, .else
  addi a1, a1, 1
  j .end_if
.else:
  addi a2, a2, 1
.end_if:
  pop_level
  div a0, a1, a2
  ret
```

cycle	mask	Instruction	Level
0	1111	push_level	1
1	1111	bnez a0, .else	1
2	0101	addi a2, a2, 1	1
3	1010	addi a1, a1, 1	1
4	0101	j .end_if	1
5	1010	pop_level	0
6	0101	pop_level	0
36	1111	div a0, a1, a2	0