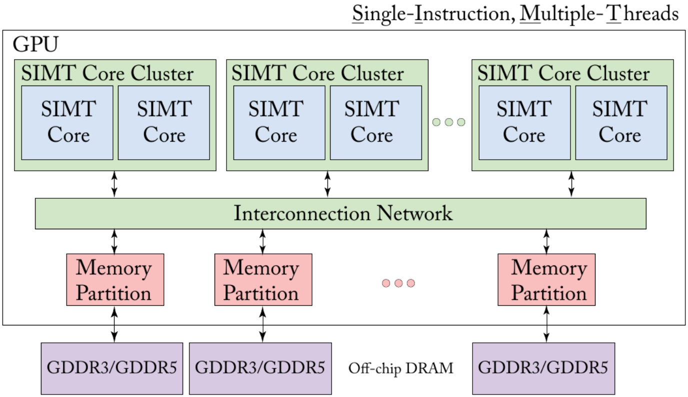
GPUs

Pawel Dybiec

November 19, 2019

Generic GPU architecture



Programming model

- Looks like MIMD
- Warps (wavefronts) are executed in lockstep, called SIMT (Single Instruction, Multiple-Thread)

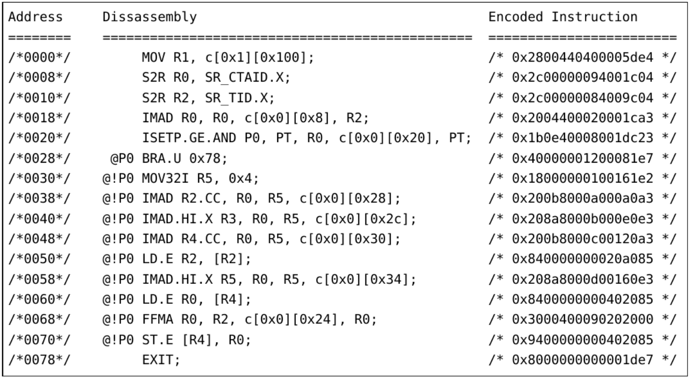
CUDA code for SAXPY

```
__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 main() {
        float *h_x, *h_y;
        int n;
        // omitted: allocate CPU memory for h_x and h_y and initialize contents
        float *d_x, *d_y;
        int nblocks = (n + 255) / 256;
        cudaMalloc( &d_x, n * sizeof(float) );
        cudaMalloc( &d_y, n * sizeof(float) );
        cudaMemcpy( d_x, h_x, n * sizeof(float), cudaMemcpyHostToDevice );
        cudaMemcpy( d_y, h_y, n * sizeof(float), cudaMemcpyHostToDevice );
        saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
        cudaMemcpy( h_x, d_x, n * sizeof(float), cudaMemcpyDeviceToHost );
        // omitted: use h_y on CPU, free memory pointed to by h_x, h_y, d_x, and d_y
}
```

PTX for SAXPY

```
.visible .entry _Z5saxpyifPfS_(
.param .u32 _Z5saxpyifPfS___param_0,
.param .f32 _Z5saxpyifPfS___param_1,
.param .u64 _Z5saxpyifPfS___param_2,
.param .u64 _Z5saxpyifPfS___param_3
)
{
.reg .pred %p<2>;
.reg .f32 %f<5>;
.reg .b32 %r<6>;
.reg .b64 %rd<8>;
ld.param.u32 %r2, [_Z5saxpyifPfS___param_0];
ld.param.f32 %f1, [_Z5saxpyifPfS___param_1];
ld.param.u64 %rd1, [_Z5saxpyifPfS___param_2];
ld.param.u64 %rd2, [_Z5saxpyifPfS___param_3];
mov.u32 %r3, %ctaid.x;
mov.u32 %r4, %ntid.x;
mov.u32 %r5, %tid.x;
mad.lo.s32 %r1, %r4, %r3, %r5;
setp.ge.s32 %p1, %r1, %r2;
@%p1 bra BB0_2;
cvta.to.global.u64 %rd3, %rd2;
cvta.to.global.u64 %rd4, %rd1;
mul.wide.s32 %rd5, %r1, 4;
add.s64 %rd6, %rd4, %rd5;
ld.global.f32 %f2, [%rd6];
add.s64 %rd7, %rd3, %rd5;
ld.global.f32 %f3, [%rd7];
fma.rn.f32 %f4, %f2, %f1, %f3;
st.global.f32 [%rd7], %f4;
BB0_2:
ret;
}
```

SASS (Fermi - CUDA 8) for SAXPY



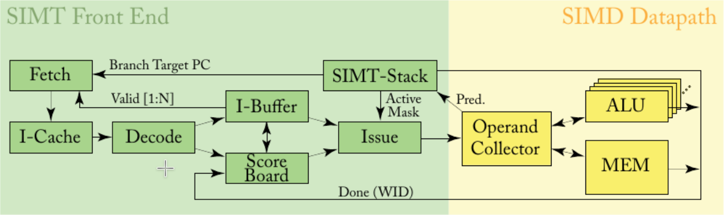
```
Address     Dissassembly                                     Encoded Instruction
========    ===============================================  ========================
/*0000*/         MOV R1, c[0x1][0x100];                      /* 0x2800440400005de4 */
/*0008*/         S2R R0, SR_CTAID.X;                         /* 0x2c00000094001c04 */
/*0010*/         S2R R2, SR_TID.X;                           /* 0x2c00000084009c04 */
/*0018*/         IMAD R0, R0, c[0x0][0x8], R2;               /* 0x2004400020001ca3 */
/*0020*/         ISETP.GE.AND P0, PT, R0, c[0x0][0x20], PT;  /* 0x1b0e40008001dc23 */
/*0028*/     @P0 BRA.U 0x78;                                 /* 0x40000001200081e7 */
/*0030*/    @!P0 MOV32I R5, 0x4;                             /* 0x18000000100161e2 */
/*0038*/    @!P0 IMAD R2.CC, R0, R5, c[0x0][0x28];           /* 0x200b8000a000a0a3 */
/*0040*/    @!P0 IMAD.HI.X R3, R0, R5, c[0x0][0x2c];         /* 0x208a8000b000e0e3 */
/*0048*/    @!P0 IMAD R4.CC, R0, R5, c[0x0][0x30];           /* 0x200b8000c00120a3 */
/*0050*/    @!P0 LD.E R2, [R2];                              /* 0x840000000020a085 */
/*0058*/    @!P0 IMAD.HI.X R5, R0, R5, c[0x0][0x34];         /* 0x208a8000d00160e3 */
/*0060*/    @!P0 LD.E R0, [R4];                              /* 0x8400000000402085 */
/*0068*/    @!P0 FFMA R0, R2, c[0x0][0x24], R0;              /* 0x3000400090202000 */
/*0070*/    @!P0 ST.E [R4], R0;                              /* 0x9400000000402085 */
/*0078*/         EXIT;                                       /* 0x8000000000001de7 */
```

OpenCL

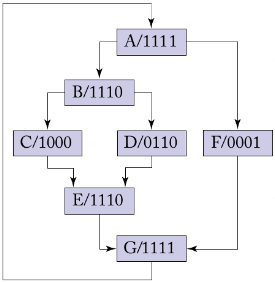
```
float fn0(float a,float b)
{
        if(a>b)
                return(a * a - b);
        else
                return(b * b - a);
}
```

```
// Registers r0 contains "a", r1 contains "b"
// Value is returned in r2
v_cmp_gt_f32 r0, r1 // a>b
s_mov_b64 s0, exec
// Save current exec mask
s_and_b64 exec, vcc, exec // Do "if"
s_cbranch_vccz label0 // Branch if all lanes fail
v_mul_f32 r2, r0, r0 // result = a * a
v_sub_f32 r2, r2, r1 // result = result - b
label0:
s_not_b64 exec, exec // Do "else"
s_and_b64 exec, s0, exec // Do "else"
s_cbranch_execz label1 // Branch if all lanes fail
v_mul_f32 r2, r1, r1 // result = b * b
v_sub_f32 r2, r2, r0 // result = result - a
label1:
s_mov_b64 exec, s0
// Restore exec mask
```

Microarchitecture of GPU core



SIMD Stack



(a) Example Program

	Ret./Reconv. PC	Next PC	Active Mask
	-	G	1111
	G	F	0001
TOS →	G	B	1110

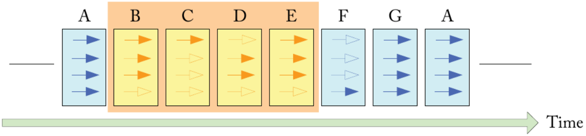
(c) Initial State

	Ret./Reconv. PC	Next PC	Active Mask	
	-	G	1111	
	G	F	0001	
	G	E	1110	(i)
	E	D	0110	(ii)
TOS →	E	C	1000	(iii)

(d) After Divergent Branch

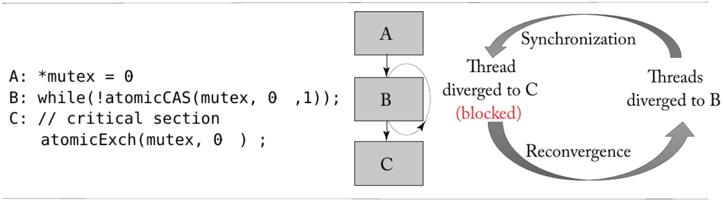
	Ret./Reconv. PC	Next PC	Active Mask
	-	G	1111
	G	F	0001
TOS →	G	E	1110

(e) After Reconvergence



(b) Re-convergence at Immediate Post-Dominator of B

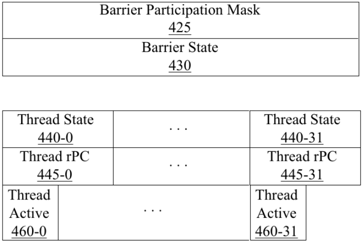
Deadlock - SIMD Stack



```
A: *mutex = 0
B: while(!atomicCAS(mutex, 0  ,1));
C: // critical section
   atomicExch(mutex, 0  ) ;
```

Figure 3.5: SIMT deadlock example (based on Figure 1 from ElTantawy and Aamodt [2016]).

Stackless



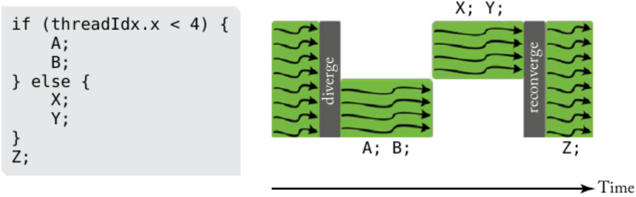
Barrier Participation Mask 425
Barrier State 430

Thread State 440-0	. . .	Thread State 440-31
Thread rPC 445-0	. . .	Thread rPC 445-31
Thread Active 460-0	. . .	Thread Active 460-31

Figure 3.6: Alternate stack-less convergence barrier based branch divergence handling mechanism recently described by NVIDIA (based on Figure 4B from Diamos et al. [2015]).

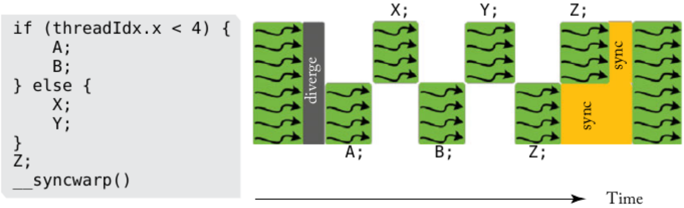
- Barrier Participation Mask - tracks which threads may participate in given barrier
- Barrier State - which threads have arrived to given barrier
- Thread State - Blocked | Ready | Yielded
- rPC - program counter
- thread active - is thread active

Volta reconvergence



```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

Figure 3.9: Example showing behavior of stack-based reconvergence (based on Figure 20 from Nvidia [2017]).



```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```

Figure 3.10: Example showing behavior of Volta reconvergence (based on Figure 23 from Nvidia [2017]).

Unified L1 architecture

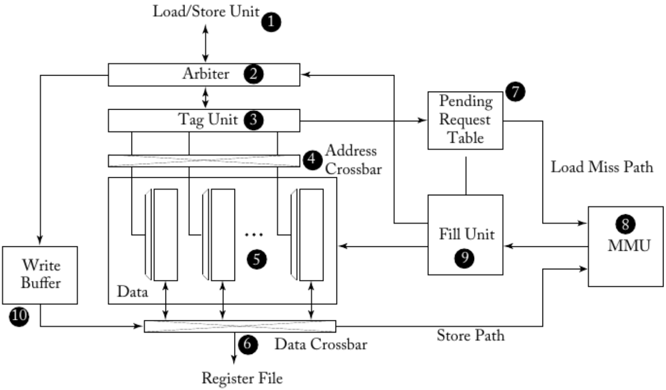
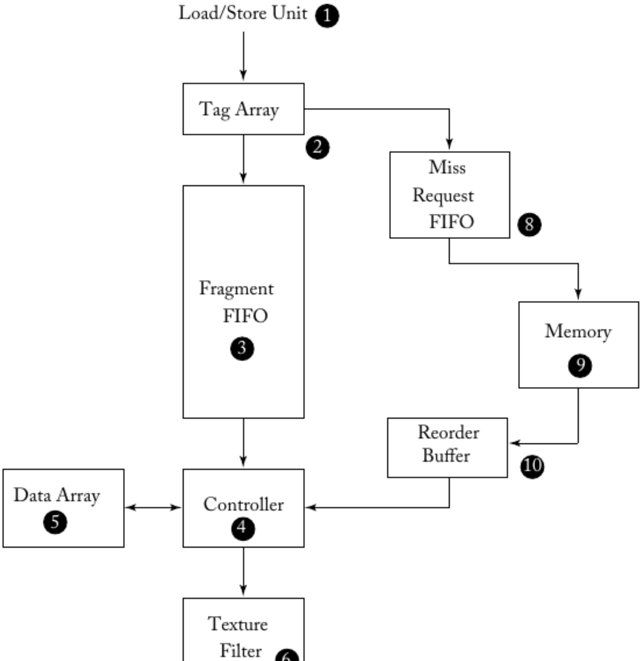


Figure 4.1: Unified L1 data cache and shared memory [Minkin et al., 2012].

Unified L1 architecture

- Shared by all threads in CTA
- Bank conflicts
- Coalesced accesses

L1 texture cache


Load/Store Unit 1
Tag Array
2
Miss Request FIFO
8
Fragment FIFO
3
Memory
9
Reorder Buffer
10
Data Array
5
Controller
4
Texture Filter
6