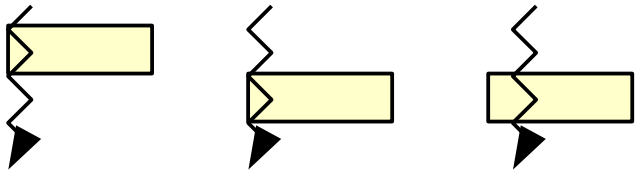
GPU programming: Warp-synchronous programming

Sylvain Collange
Inria Rennes – Bretagne Atlantique
sylvain.collange@inria.fr

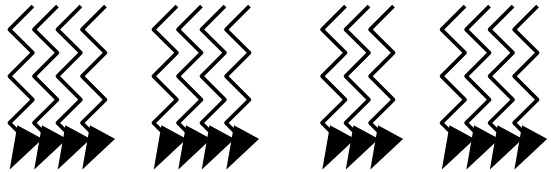
Bypassing SIMT limitations

Multi-thread + SIMD



- Across SIMD lanes of same thread
  - Direct communication without synchronization
- Between threads
  - All synchonization primitives, wait-based and lock-based algorithms

SIMT model



- Common denominator of operations allowed in SIMD and multi-thread
  - No direct communication
  - No blocking algorithm

- If we know the warp/thread organization, we can pretend we program a multi-thread + SIMD machine
  - Drawback: give up automatic divergence management

Outline

- Inter-thread communication
- Warp-based execution and divergence
- Intra-warp communication:
  warp-synchronous programming

Inter-thread/inter-warp communication

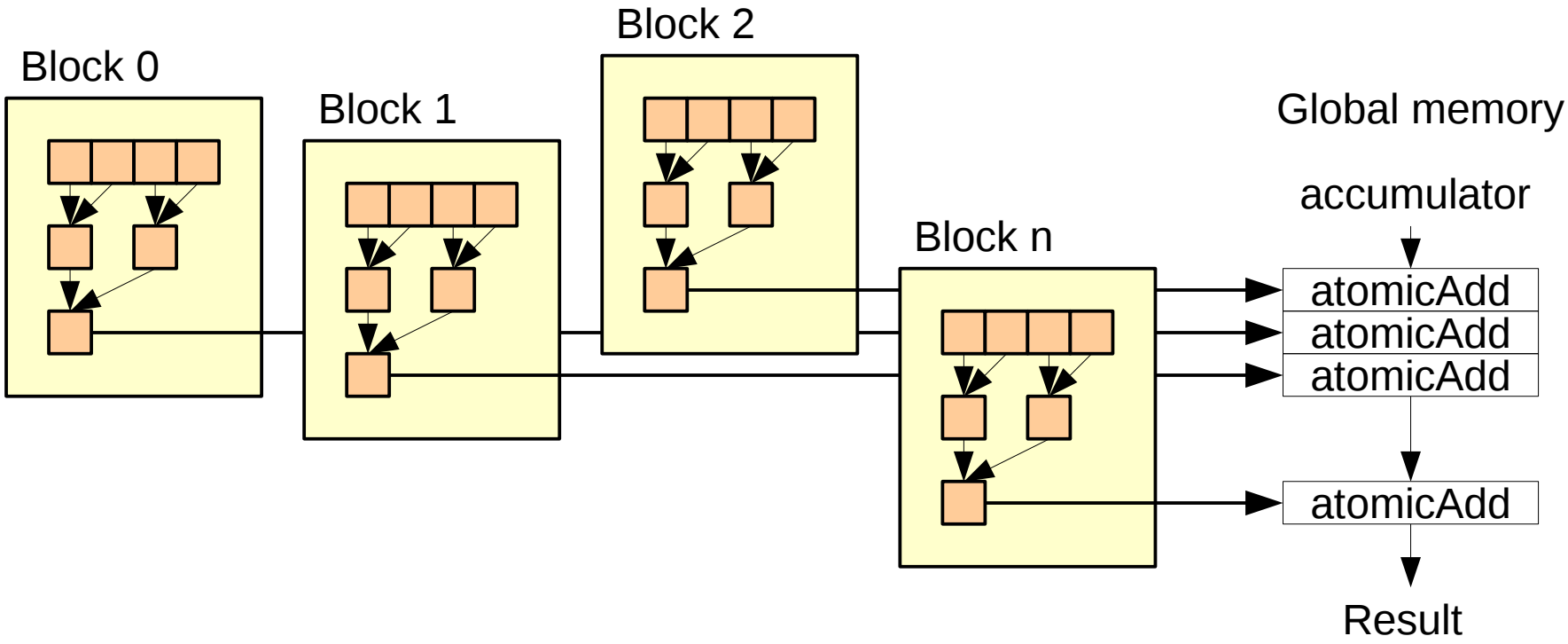
- Barrier: simplest form of synchronization
  - Easy to use
  - Coarse-grained
- Atomics
  - Fine-grained
  - Can implement *wait-free* algorithms
  - May be used for blocking algorithms (locks)
    among **warps** of the **same block**, but not recommended
- Communication through memory
  - Beware of consistency

Atomics

- Read, modify, write in one operation
  - Cannot be mixed with accesses from other thread
- Available operators
  - Arithmetic: atomic{Add,Sub,Inc,Dec}
  - Min-max: atomic{Min,Max}
  - Synchronization primitives: atomic{Exch,CAS}
  - Bitwise: atomic{And,Or,Xor}
- On global memory, and shared memory
- Performance impact in case of contention
  - Atomic operations to the same address are serialized

Example: reduction

- After local reduction inside each block,
  use atomics to accumulate the result in global memory



- Complexity?
- Time including kernel launch overhead?

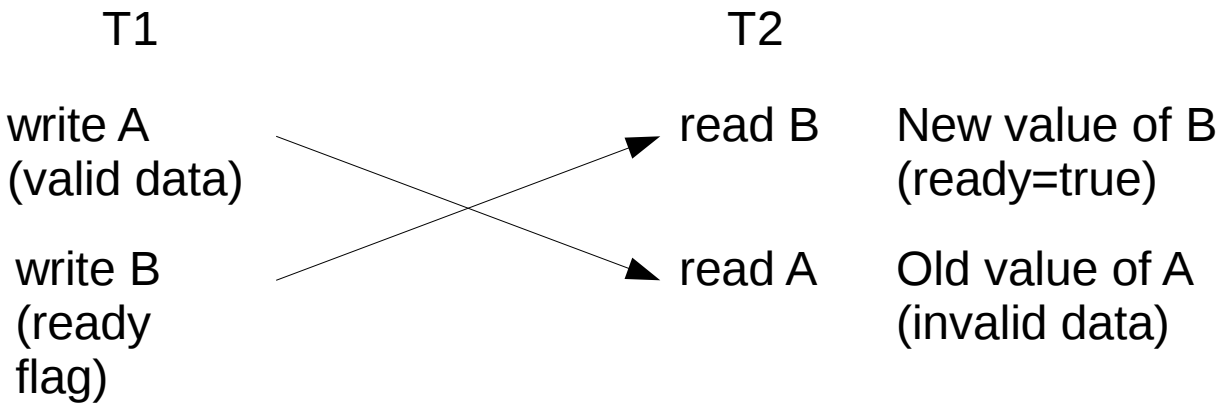
Example: compare-and-swap

- Use case: perform an arbitrary associative and commutative operation atomically on a single variable
- atomicCAS(p, old, new) does atomically
  - if *p == old then assign *p ← new, return old
  - else return *p

```
__shared__ unsigned int data;
unsigned int old = data;                        // Read once
unsigned int assumed;
do {
    assumed = old;
    newval = f(old, x);                         // Compute
    old = atomicCAS(&data, old, newval);        // Try to replace
} while(assumed != old);                        // If failed, retry
```

Memory consistency model

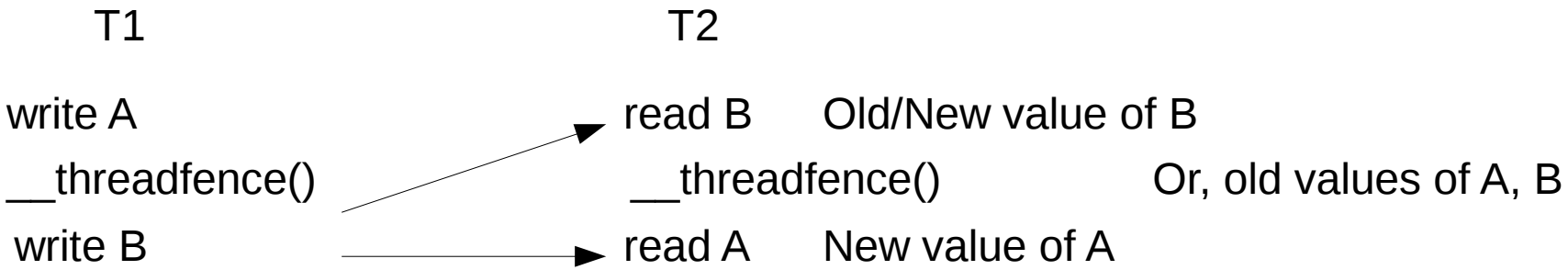
- Nvidia GPUs (and compiler) implement a relaxed consistency model
  - No global ordering between memory accesses
  - Threads may not see the writes/atomics in the same order



- Need to enforce explicit ordering

Enforcing memory ordering: fences

- `__threadfence_block`
  `__threadfence`
  `__threadfence_system`
  - Make writes preceding the fence appear before writes following the fence for the other threads at the block / device / system level
  - Make reads preceding the fence happen after reads following the fence



- Declare shared variables as **volatile**
  to make writes visible to other threads
  (prevents compiler from removing “redundant” read/writes)
- `__syncthreads` implies `__threadfence_block`

Outline

- Inter-thread communication
- Warp-based execution and divergence
- Intra-warp communication:
  warp-synchronous programming

Warp-based execution

Reminder

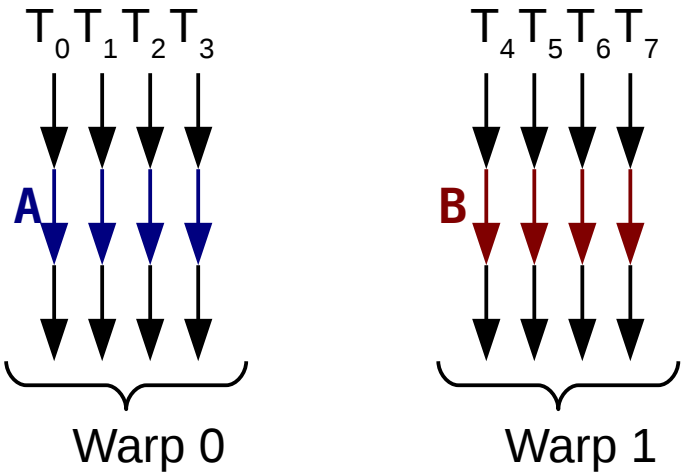
- Threads in a warp run in lockstep
- On current NVIDIA architectures, warp is 32 threads
- A block is made of warps
  - Warps do not cross block boundaries
  - Block size multiple of 32 for best performance

Branch divergence

- Conditional block

```
if(c) {
    // A
}
else {
    // B
}
```

- When all threads of a warp take the same path:



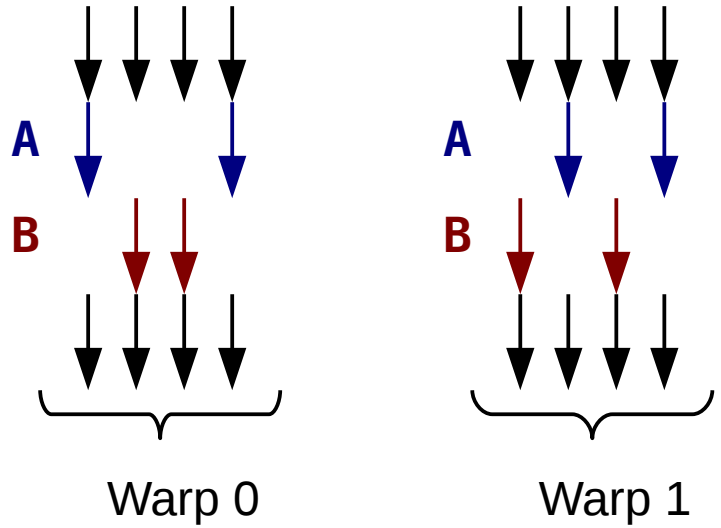
With imaginary 4-thread warps

Branch divergence

- Conditional block

```
if(c) {
    // A
}
else {
    // B
}
```

- When threads in a warp take different paths:



- Warps have to go through both A and B: lower performance

Avoiding branch divergence

- Hoist identical computations and memory accesses outside conditional blocks

```
if(tid % 2) {
    s += 1.0f/tid;
}
else {
    s -= 1.0f/tid;
}
```

```
float t = 1.0f/tid;
if(tid % 2) {
    s += t;
}
else {
    s -= t;
}
```

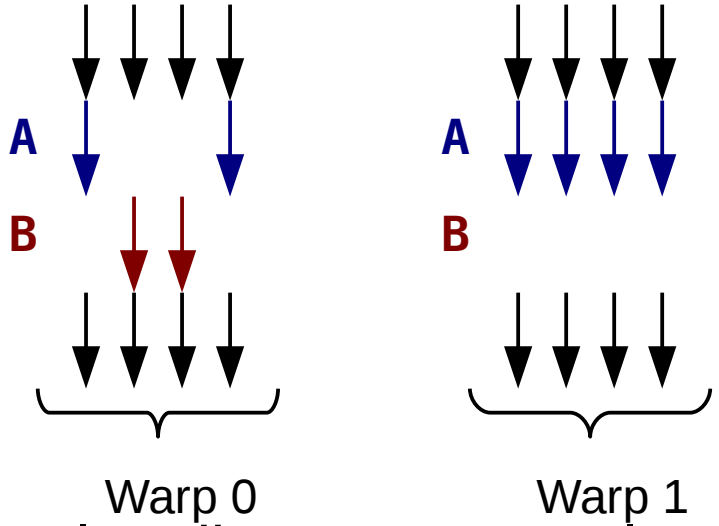
- When possible, re-schedule work to make non-divergent warps

```
// Compute 2 values per thread
int i = 2 * tid;
s += 1.0f/i – 1.0f/(i+1);
```

- What if I use C's ternary operator (`?:`) instead of `if`?
  (or tricks like ANDing with a mask, multiplying by a boolean...)

Ternary operator ? good : bad

- Run both branches and select: `R = c ? A : B;`
  - No more divergence?
- All threads have to take both paths
  No matter whether the condition is divergent or not



- Does **not** solve divergence: we lose in all cases!
- Only benefit: fewer instructions
  - May be faster for short, often-divergent branches
- Compiler will choose automatically when to use predication
  - Advice: keep the code readable, let the compiler optimize

Barriers and divergence

- Remember barriers cannot be called in divergent blocks
  - All threads or none need to call __syncthreads()
- If: need to split

```
if(p) {
  ...
  // Sync here?
  ...
}
```

```
if(p) {
  ...
}
__syncthreads();
if(p) {
  ...
}
```

- Loops: what if trip count depends on data?

```
while(p) {
  ...
  // Sync here?
  ...
}
```

?

Barriers instructions

- Barriers with boolean reduction
  - `__syncthreads_or(p)` / `__syncthreads_and(p)`
    Synchronize, then returns the boolean OR / AND of all thread predicates p
  - `__syncthreads_count(p)`
    Synchronize, then returns the number of non-zero predicates
- Loops: what if trip count depends on data?
  - Loop while at least one thread is still in the loop

```
while(p) {
  ...
  // Sync here?
  ...
}
```

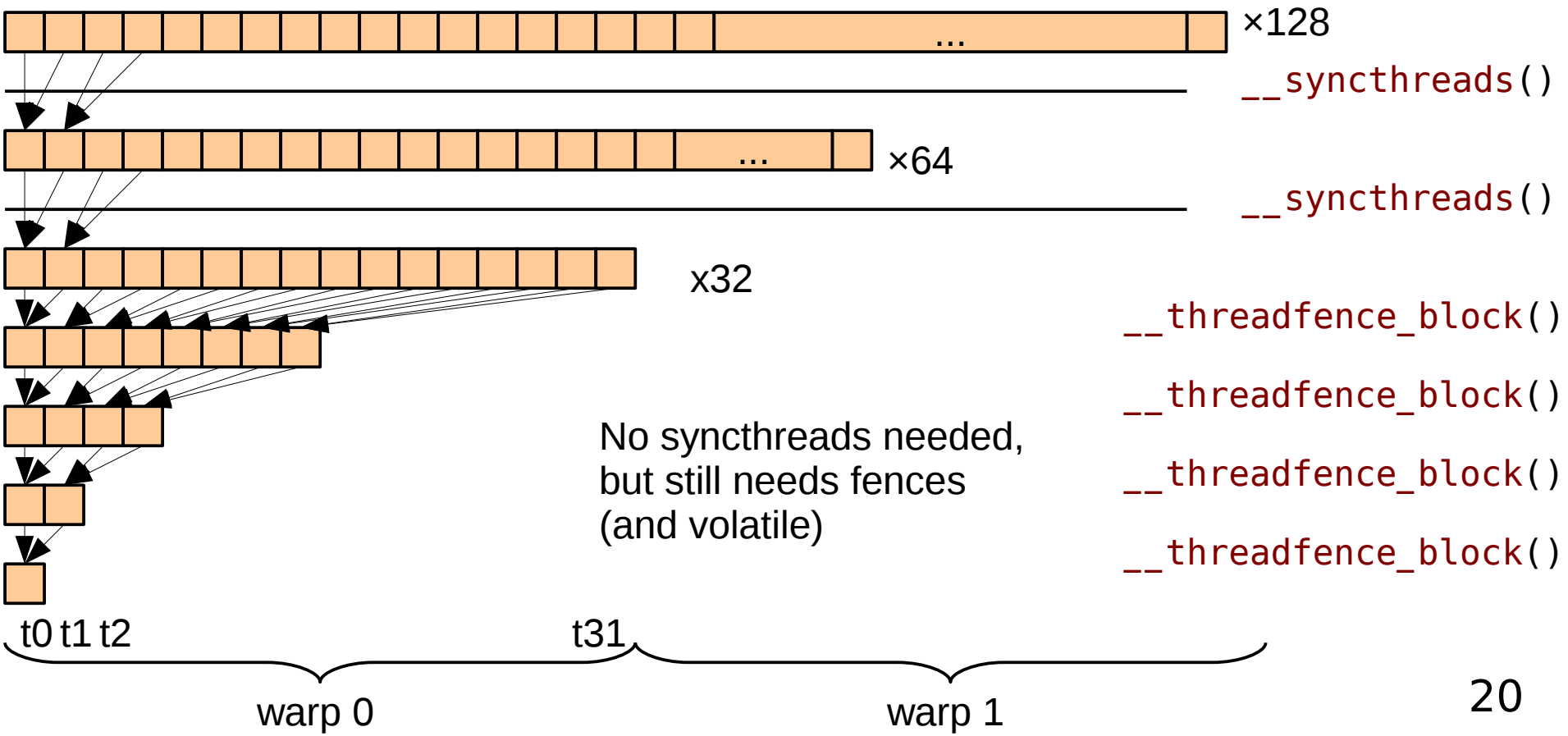
```
while(__syncthreads_or(p)) {
  if(p) {
  ...
  }
  __syncthreads();
  if(p) {
  ...
  }
}
```

Outline

- Inter-thread communication
- Warp-based execution and divergence
- Intra-warp communication:
  warp-synchronous programming

Warp-synchronous programming

- We know threads in a warp run synchronously
  - No need to synchronize them explicitly
- Can use SIMD (PRAM-style) algorithms inside warps
- Example: last steps of a reduction



Warp-synchronous programming: tools

We need warp size, and thread ID inside a warp: lane ID

- Predefined variable: `warpSize`
- Lane ID exists in PTX, not in C for CUDA
  Can be recovered using inline PTX

```
__device__ unsigned int laneID() {
    unsigned int lane;
    asm volatile("mov.u32 %0, %%laneid;" : "=r"(lane));
    return lane;
}
```

We can emit PTX using the asm keyword,
but remember PTX is not assembly.

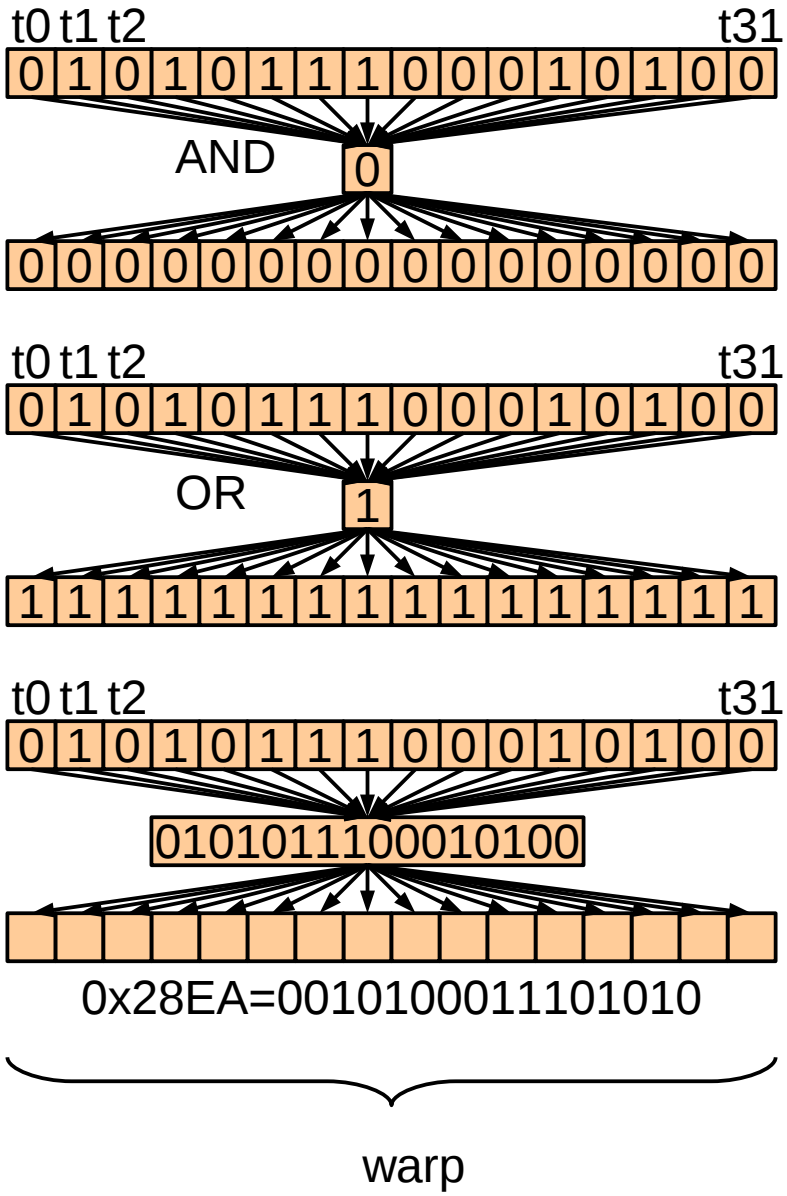
Warp vote instructions

- `p2 = __all(p1)`
  horizontal AND between predicates p1 of all threads in the warp

- `p2 = __any(p1)`
  OR between all p1

- `n = __ballot(p)`
  Set bit *i* of integer n
  to value of p for thread *i*
  i.e. get bit mask as an integer

Like __syncthreads_{and,or} for a warp
Use: take control decisions for the whole warp



Manual divergence management

- How to write an if-then-else in warp-synchronous style?
  i.e. without breaking synchronization
  - Using predication:
    execute both sides always

```
if(p) { A(); B(); }
else { C(); }
```

```
if(p) { A(); }
// Threads synchronized
if(p) { B(); }
// Threads synchronized
if(!p) { C(); }
```

  - Using vote instructions:
    only execute taken paths
    Skip block if no thread takes it

```
if(__any(p)) {
    if(p) { A(); }
    // Threads synchronized
    if(p) { B(); }
}
if(__any(!p)) {
    // Threads synchronized
    if(!p) { C(); }
}
```

- How to write a while loop?

Ensuring reconvergence

When and where do threads reconverge?

- No hard guarantees today
  - e.g. Can the compiler alter the control flow?
  - Will hopefully get standardized [1]
- In practice, we can assume reconvergence:
  - After structured if-then-else
  - After structured loops
  - At function return
- Example: can we assume reconvergence here?

```
if(threadIdx.x != 0) {            ← Divergent condition
   if(blockIdx.x == 0) return;    ← Uniform condition
}
// Threads synchronized at this point ?
```

[1] B. Gaster. *An Execution Model for OpenCL 2.0.* 2014

Ensuring reconvergence

When and where do threads reconverge?

- No hard guarantees today
  - e.g. Can the compiler alter the control flow?
  - Will hopefully get standardized [1]
- In practice, we can assume reconvergence:
  - After structured if-then-else
  - After structured loops
  - At function return
- Example: can we assume reconvergence here?

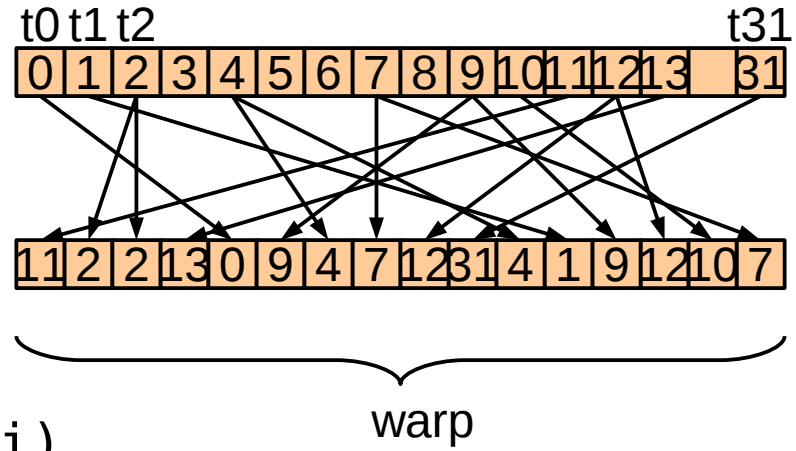
```
if(threadIdx.x != 0) {            ← Divergent condition
    if(blockIdx.x == 0) return;   ← Uniform condition
}
// Threads synchronized at this point ?
```

  - **No.** Return creates unstructured control flow.
    Will or won't reconverge depending on the architecture.

[1] B. Gaster. *An Execution Model for OpenCL 2.0.* 2014

Shuffle

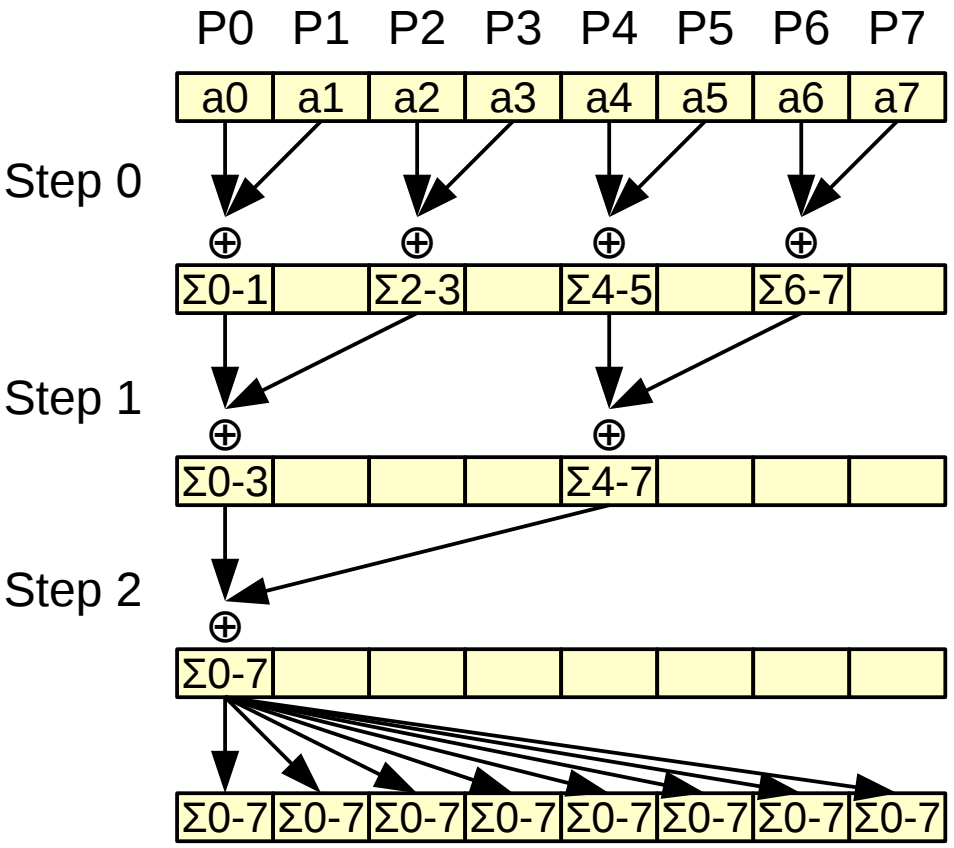
Exchange data between lanes



- `__shfl(v, i)`
  Get value of thread i in the warp
  - Use: 32 concurrent lookups in a 32-entry table
  - Use: arbitrary permutation...
- `__shfl_up(v, i)` ≈ `__shfl(v, tid-i)`, `__shfl_down(v, i)` ≈ `__shfl(v, tid+i)`
  Same, indexing relative to current lane
  - Use: neighbor communication, shift
- `__shfl_xor(v, i)` ≈ `__shfl(v, tid ^ i)`
  - Use: exchange data pairwise: “butterfly”

Example: reduction + broadcast

- Naive algorithm

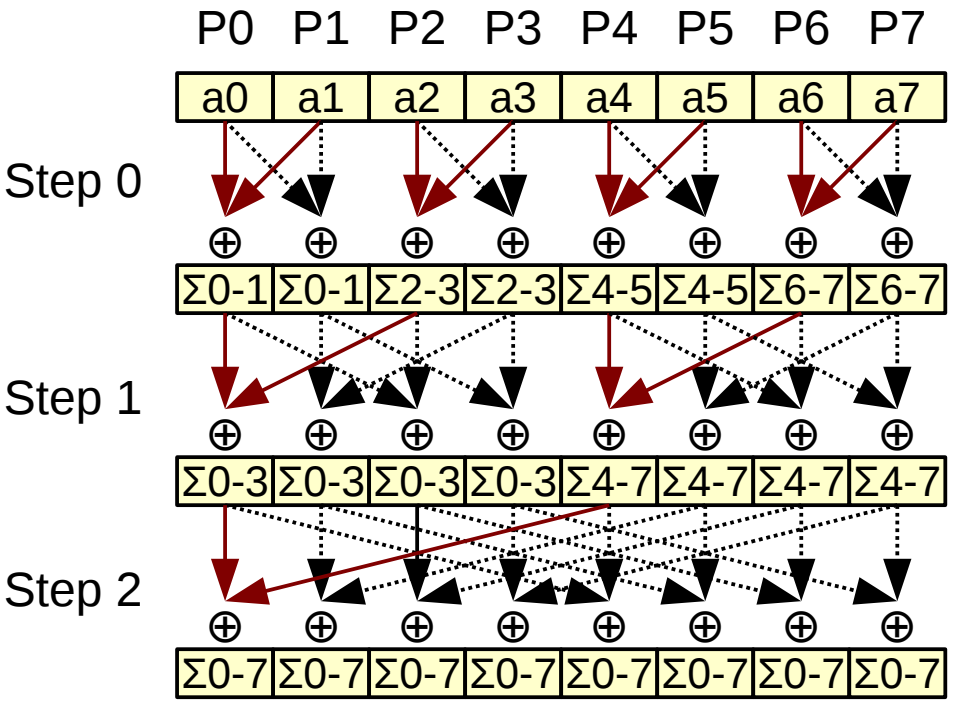



```
a[2*i]←a[2*i] + a[2*i+1]
a[4*i]←a[4*i] + a[4*i+2]
a[8*i]←a[8*i] + a[8*i+4]
a[i]←a[0]
```

- Let's rewrite it using shuffle

Example: reduction + broadcast

- With shuffle




```
ai += __shfl_xor(ai, 1);
```

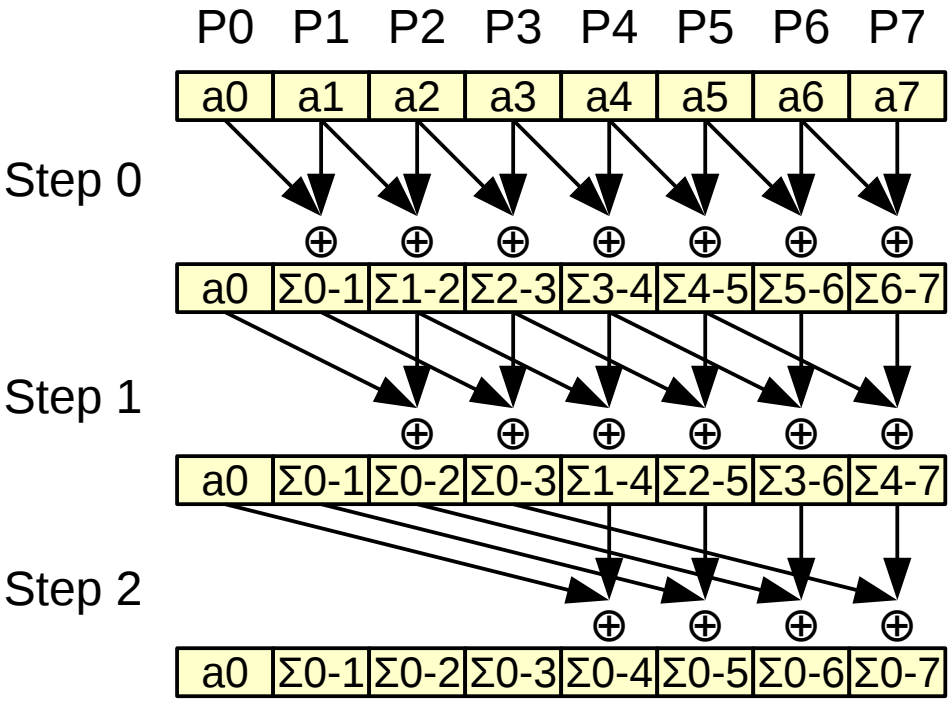
```
ai += __shfl_xor(ai, 2);
```

```
ai += __shfl_xor(ai, 4);
```

- Exercise: implement complete reduction without __syncthreads
  - Hint: use shared memory atomics

Other example: parallel prefix

- Remember our PRAM algorithm



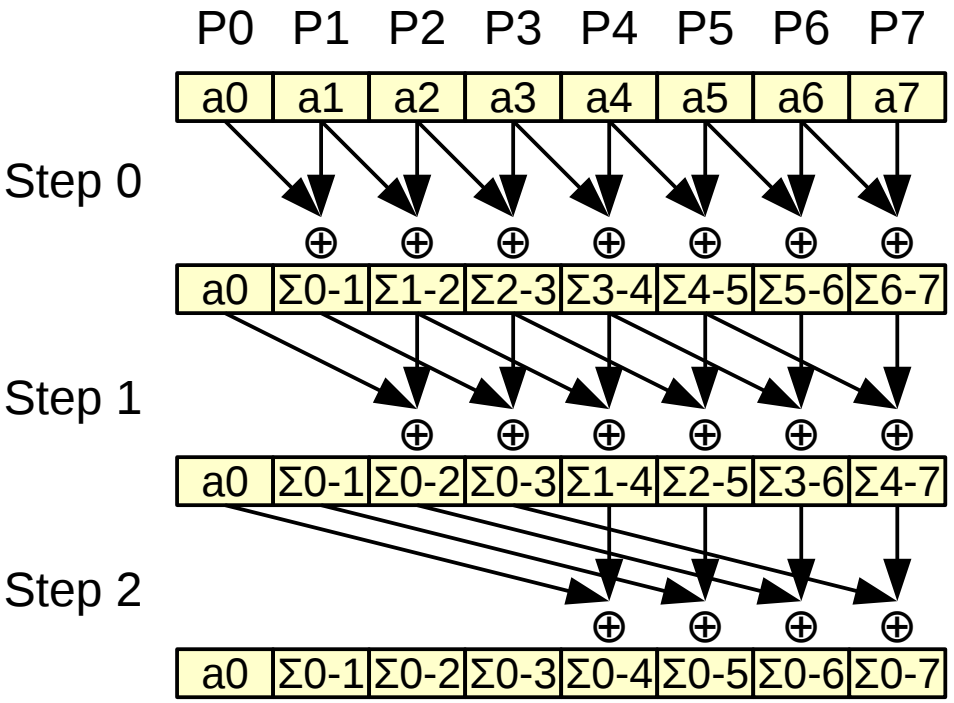

```
s[i] ← a[i]
if i ≥ 1 then
    s[i] ← s[i-1] + s[i]

if i ≥ 2 then
    s[i] ← s[i-2] + s[i]

if i ≥ 4 then
    s[i] ← s[i-4] + s[i]
```

Σi-j is the sum $\sum_{k=i}^{j} a_k$

Step d:
```
if i ≥ 2^d then
    s[i] ← s[i-2^d] + s[i]
```

Other example: parallel prefix

- Using warp-synchronous programming



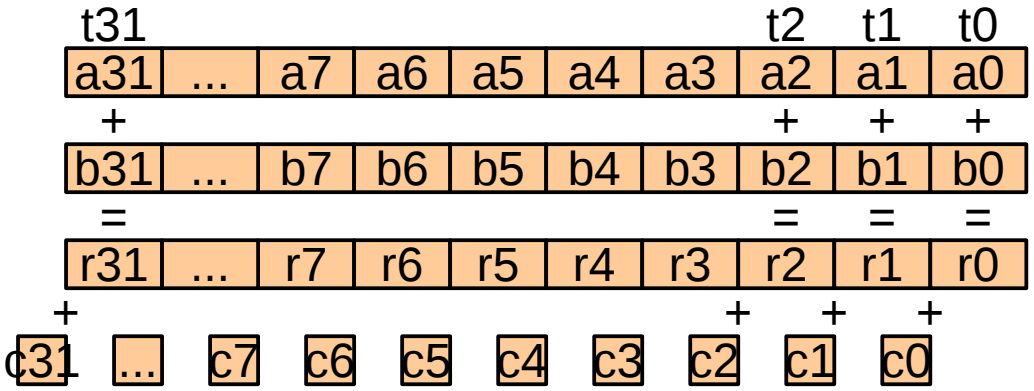
Σi-j is the sum $\sum_{k=i}^{j} a_k$

```
s = a;
n = __shfl_up(s, 1);
if(laneid >= 1)
    s += n;
n = __shfl_up(s, 2);
if(laneid >= 2)
    s += n;
n = __shfl_up(s, 4);
if(laneid >= 4)
    s += n;
```

```
for(d = 1; d <= 5; d *= 2) {
  n = __shfl_up(s, d);
  if(laneid >= d)
    s += n;
}
```

Example: multi-precision addition

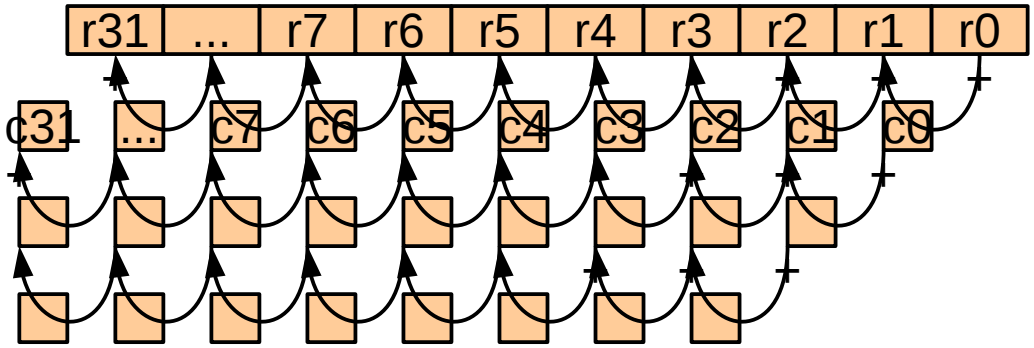
- Do an addition on 1024-bit numbers
- Represent numbers as vectors of 32×32-bit
  - A warp works on a vector
- First step: add elements of the vectors in parallel and recover carries




```
uint32_t a = A[tid],
         b = B[tid], r, c;

r = a + b;   // Sum

c = r < a;   // Get carry
```

Second step: propagate carries

- This is a parallel prefix operation
  - We can do it in log(n) steps
- But in most cases, one step will be enough
  - Loop until all carries are propagated

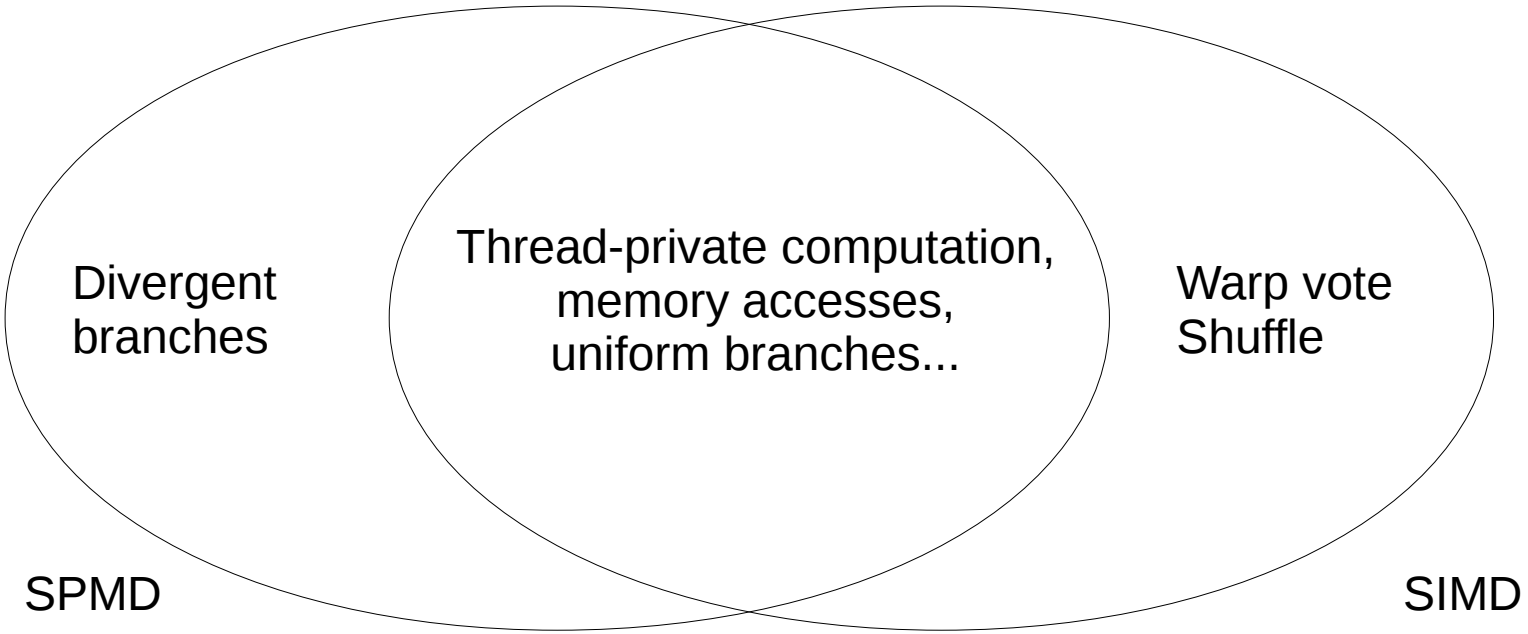


...

```
uint32_t a = A[tid],
         b = B[tid], r, c;

r = a + b;   // Sum
c = r < a;   // Get carry
while(__any(c)) {   // Carry left?
    c = __shfl_up(c, 1); // Move left
    if(laneid == 0) c = 0;
    r = r + c;   // Sum carry
    c = r < c;   // New carry?
}
R[tid] = r;
```

Mixing SPMD and SIMD

- Most language features work in either SPMD mode or SIMD mode



- Avoid divergent branches when writing SIMD code
- Consequence: SIMD code may call SPMD functions, not the other way around

Takeaway

- Two ways to program an SIMT GPU
  - With independent threads, grouped in warps
  - With warps operating on vectors
- 3 levels
  - Blocks in grid: independent tasks, no synchronization
  - Warps in block: concurrent “threads”, explicitly synchronizable
  - Threads in warp: implicitly synchronized

- Warp-synchronous still somehow uncharted territory in CUDA, but widely used in libraries
- If you know warp-synchronous programming in CUDA, you know SIMD programming with masking (Xeon Phi, AVX-512…)

Device functions

- Kernel can call functions
- Need to be marked for GPU compilation

```
__device__ int foo(int i) {
}
```

- A function can be compiled for both host and device

```
__host__ __device__ int bar(int i) {
}
```

- Device functions can call device functions
  - Newer GPUs support recursion

Local memory

- Registers are fast but
  - Limited in size
  - Not addressable
- Local memory used for
  - Local variables that do not fit in registers (*register spilling*)
  - Local arrays accessed with indirection

```
int a[17];
b = a[i];
```

- **Warning**: local is a misnomer!
  - Physically, local memory usually goes off-chip
  - About same performance as coalesced access to global memory

Loop unrolling

- Can improve performance
  - Amortizes loop overhead over several iterations
  - May allow constant propagation, common sub-expression elimination...
- Unrolling is **necessary** to keep arrays in registers

Not unrolled

```
int a[4];
for(int i = 0; i < 4; i++) {
    a[i] = 3 * i;
}
```

Indirect addressing:
a in local memory

Unrolled

```
int a[4];
a[0] = 3 * 0;
a[1] = 3 * 1;
a[2] = 3 * 2;
a[3] = 3 * 3;
```

Trivial computations: optimized away

Static addressing:
a in registers

- The compiler can unroll for you

```
#pragma unroll
for(int i = 0; i < 4; i++) {
    a[i] = 3 * i;
}
```

Device-side functions: C library support

- Extensive math function support
  - Standard C99 math functions in single and double precision:
    e.g. `sqrtf, sqrt, expf, exp, sinf, sin, erf, j0`...
  - More exotic math functions:
    `cospi, erfcinv, normcdf`...
  - Complete list with error bounds in the CUDA C programming guide
- Starting from CC 2.0
  - `printf`
  - `memcpy, memset`
  - `malloc, free`: allocate global memory on demand

Device-side intrinsics

- C functions that translate to one/a few machine instructions
  - Access features not easily expressed in C
- Hardware math functions in single-precision
  - GPUs have dedicated hardware for
    reverse square root, inverse, log2, exp2, sin, cos in single precision
  - Intrinsics: `__rsqrt, __rcp, exp2f, __expf, __exp10f, __log2f, __logf, __log10f, __sinf, __cosf, __sincosf, __tanf, __powf`
  - Less accurate than software implementation (except `exp2f`),
    but much faster
- Optimized arithmetic functions
  - `__brev, __popc, __clz, __ffs`...
    Bit reversal, population count, count leading zeroes, find first bit set…
  - Check the CUDA Toolkit Reference Manual
- We will see other intrinsics in the next part