GPU programming: Code optimization part 3 and advanced features

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

Outline

- Instruction-level optimization
  - A few device code features
  - Warp divergence
- Advanced features
  - Device-mapped memory, Unified virtual address space, Unified memory
  - Global and shared memory atomics
  - Warp-synchronous programming, warp vote, shuffle

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
  - Older GPUs do not 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

Outline

- Instruction-level optimization
  - A few device code features
  - Warp divergence
- Advanced features
  - Device-mapped memory, Unified virtual address space, Unified memory
  - Global and shared memory atomics
  - Warp-synchronous programming, warp vote, shuffle

Warp-based execution

Reminder

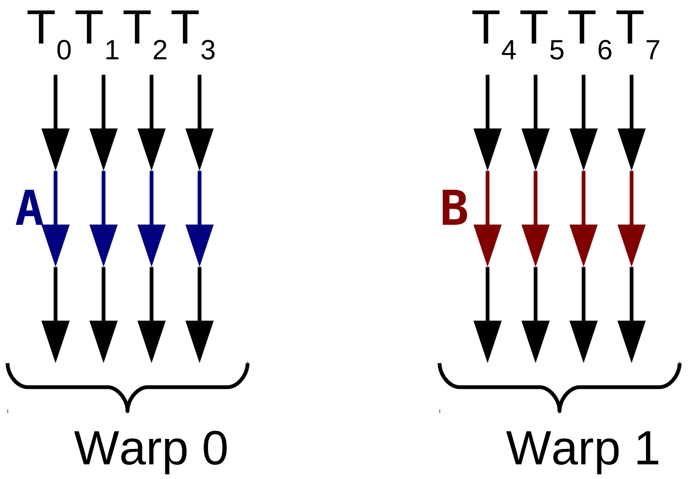
- Threads in a warp run in lockstep
- On current NVIDIA architectures, warp is 32 threads
- A block is made of warps
  - Warps do not cross block boundaries
  - Block size multiple of 32 for best performance

Branch divergence

- Conditional block

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

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



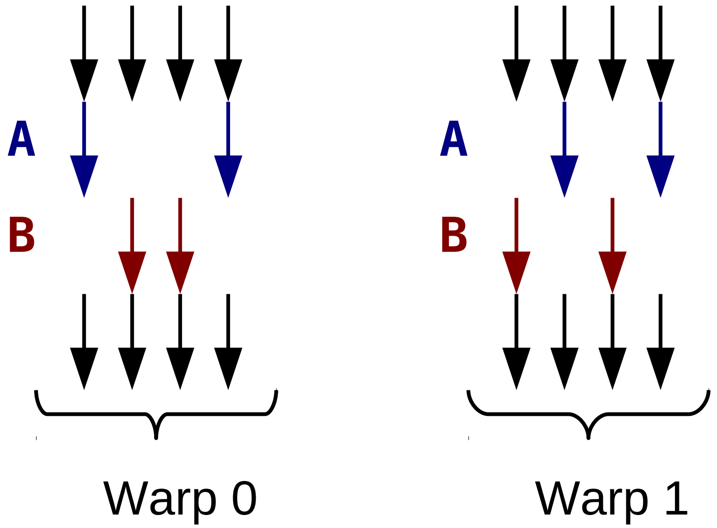
With imaginary 4-thread warps

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;
}
```

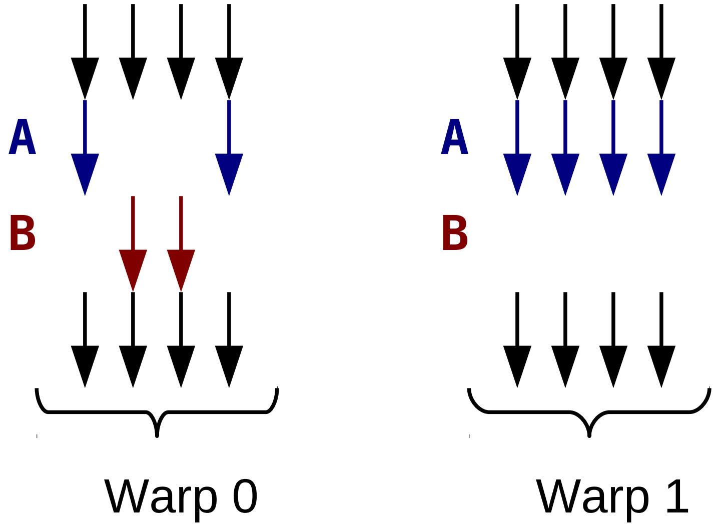
- When possible, re-schedule work to make non-divergent warps

```
// 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...)

Answer: ternary operator or predication

- 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 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) {
  ...
  }
}
```

Recap: instruction optimization

- Beware of local arrays
  use static indices and loop unrolling
- Use existing math functions
- Keep in mind branch divergence when writing algorithm
  - But do not end up managing divergence yourself

Code optimization tools

- IDE + profiler: nView
- Profile application: NVIDIA Visual Profiler
- Examine assembly code: cuobjdump
  - `cuobjdump --dump-sass`

OK, I lied

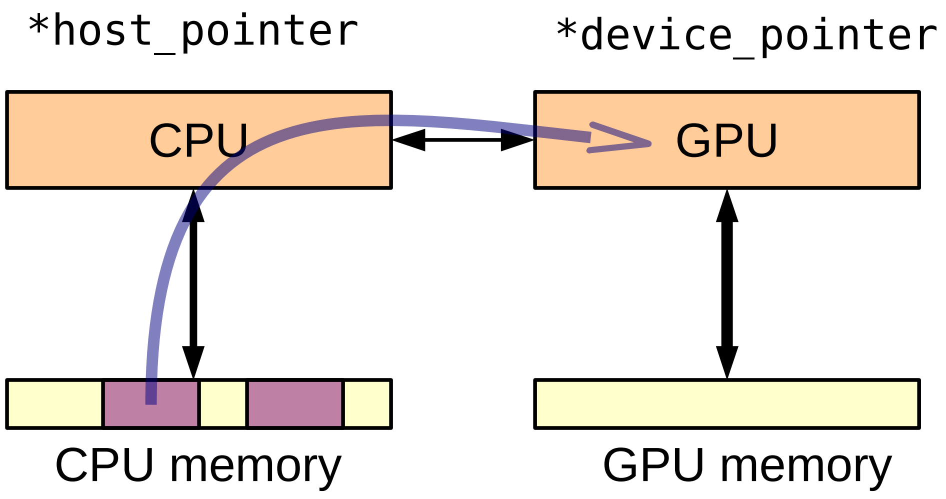
- You were told
  - CPU and GPU have distinct memory spaces
  - Blocks cannot communicate
  - You need to synchronize threads inside a block
- This is the least-common denominator across all CUDA GPUs
- This is not true (any more). We now have:
  - Device-mapped, Unified virtual address space, Unified memory
  - Global and shared memory atomics
  - Warp-synchronous programming, warp vote, shuffle
  - Dynamic parallelism
- Many features are still specific to CUDA and Nvidia GPUs
  - Should become available in OpenCL eventually...

Outline

- Instruction-level optimization
  - A few device code features
  - Warp divergence
- Advanced features
  - Device-mapped memory, Unified virtual address space, Unified memory
  - Global and shared memory atomics
  - Warp-synchronous programming, warp vote, shuffle

Device-mapped memory

- The GPU can access host memory directly,
  if explicitly mapped in the device address space
- **Lower bandwidth** but **lower latency** than DMA transfers
  - Useful for light or sparse memory accesses
- Advantageous for integrated GPUs
- Still different **address spaces**
  Device address is different than host address



Example

```
// Allocate device-mappable memory
float* a;
cudaHostAlloc((void**)&a, bytes, cudaHostAllocMapped);

// Get device-side pointer
float* da;
cudaHostGetDevicePointer((void**)&d_a, a, 0);

// Write using host pointer on CPU
a[0] = 42;

// Read/write using device pointer on GPU
mykernel<<<grid, block>>>(d_a);

// Wait for kernel completion
cudaDeviceSynchronize();

// Read using host pointer
printf("a=%f\n", a[0]);

// Free memory
cudaFreeHost(a)
```

Unified virtual address space (UVA)

- Coordinate allocation in CPU and GPU address space
  - Can tell if an address is on the device, host or both
- `cudaHostAlloc` returns an address valid in both host and device space
- No need to specify direction in `cudaMemcpy`
- Caveat: not a true unified address space
  - Having an address does not mean you can access the data it points to
  - Memory allocated with malloc still cannot be accessed by GPU
- Requirements: CUDA ≥ 4.0, 64-bit OS, CC ≥ 2.0
  - Not available on Windows Vista/7 under WDDM

Unified memory: the real thing

- Allocate memory using `cudaMallocManaged`
  - The CUDA runtime will take care of the transfers
  - No need to call `cudaMemcpy` any more
  - Behaves as if you had a single memory space
- Suboptimal performance: software-managed coherency
  - Still call `cudaMemcpy` when you can

- Requires CUDA ≥ 6.0, CC ≥ 3.0 (preferably 5.x), 64-bit Linux or Windows

VectorAdd using unified memory

```
int main()
{
    int numElements = 50000;
    size_t size = numElements * sizeof(float);

    float *A, *B, *C;
    cudaMallocManaged((void **)&A, size);
    cudaMallocManaged((void **)&B, size);
    cudaMallocManaged((void **)&C, size);
    Initialize(A, B);

    int blocks = numElements;
    vectorAdd2<<<blocks, 1>>>(A, B, C);
    cudaDeviceSynchronize();
    Display(C);

    cudaFree(A);
    cudaFree(B);
    cudaFree(C);
}
```

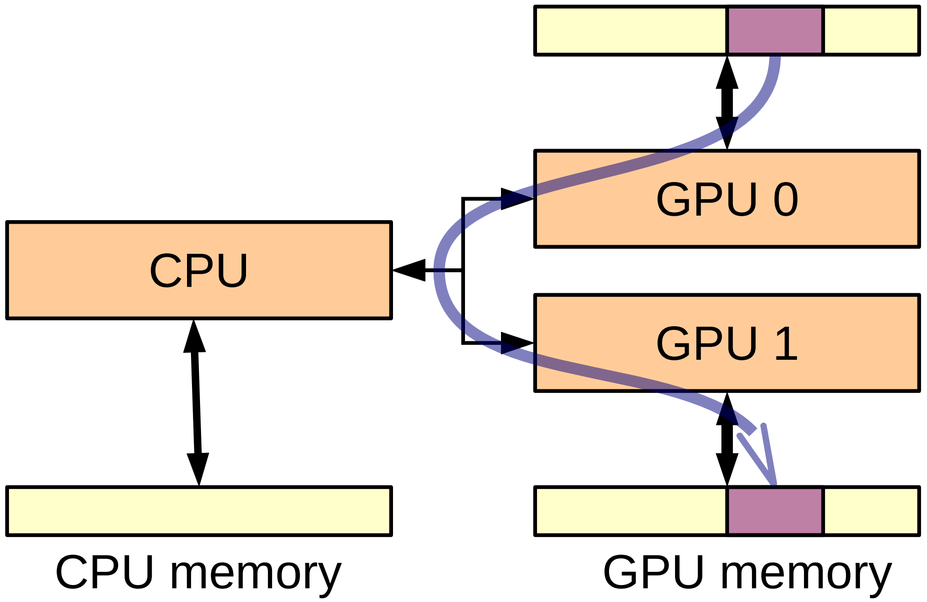
- Can I replace cudaMallocManaged by cudaHostAlloc (with UVA?)
  - What is the difference?

Multi-GPU programming

- What if I want to use multiple GPUs on the same machine?
- `cudaGetDeviceCount`, `cudaGetDeviceProperties` enumerate devices
- `cudaSetDevice(i)` selects the current device
  - All following CUDA calls in the thread will concern this device
- Streams are associated with devices
  - But cudaStreamWaitEvent can synchronize with events on other GPU: allow inter-GPU synchronization
- Host memory accessible from multiple devices: `cudaHostAlloc(..., cudaHostAllocPortable)`
  - Then call cudaHostGetDevicePointer for each GPU

Peer-to-peer memory copy/access

- Transfer memory between GPUs
  - cudaMemcpyDeviceToDevice will not work. Why?
  - Without UVA
    `cudaMemcpyPeer()`
    `cudaMemcpyPeerAsync()`
  - With UVA: just plain `cudaMemcpy`
- Direct access to other GPU's memory
  - Check result of `cudaDeviceCanAccessPeer`
  - Call `cudaDeviceEnablePeerAccess` from accessing GPU



Recap

- We have too many mallocs...
  - cudaMalloc
  - cudaMallocHost
  - cudaHostAlloc(..., cudaHostAlloc{Portable|Mapped})
  - cudaMallocManaged
- And too many memcpys
  - cudaMemcpy(dest, src,
    cudaMemcpy{HostToDevice, DeviceToHost, DeviceToDevice, Default})
  - cudaMemcpyAsync
  - cudaMemcpyPeer[Async]
- Quizz: do you remember what they do
  and when we should use them?

Recap: evolution of the memory model

New features: going away from the split memory model

- Device-mapped: GPU can map and access CPU memory
  - Lower bandwidth than DMA transfers
  - Higher latency than GPU memory access
- Uniform virtual addressing (CC 2.0):
  synchronize memory space between CPU and GPU
  - Addresses are unique across the system
  - No need to specify direction in `cudaMemcpy`
- Unified memory (CC 3.x):
  both CPU and GPU share a managed memory space
  - Driver manages transfers automatically
  - Only for memory allocated as managed
  - Unmanaged + cudaMemcpy still useful for optimized transfers

Outline

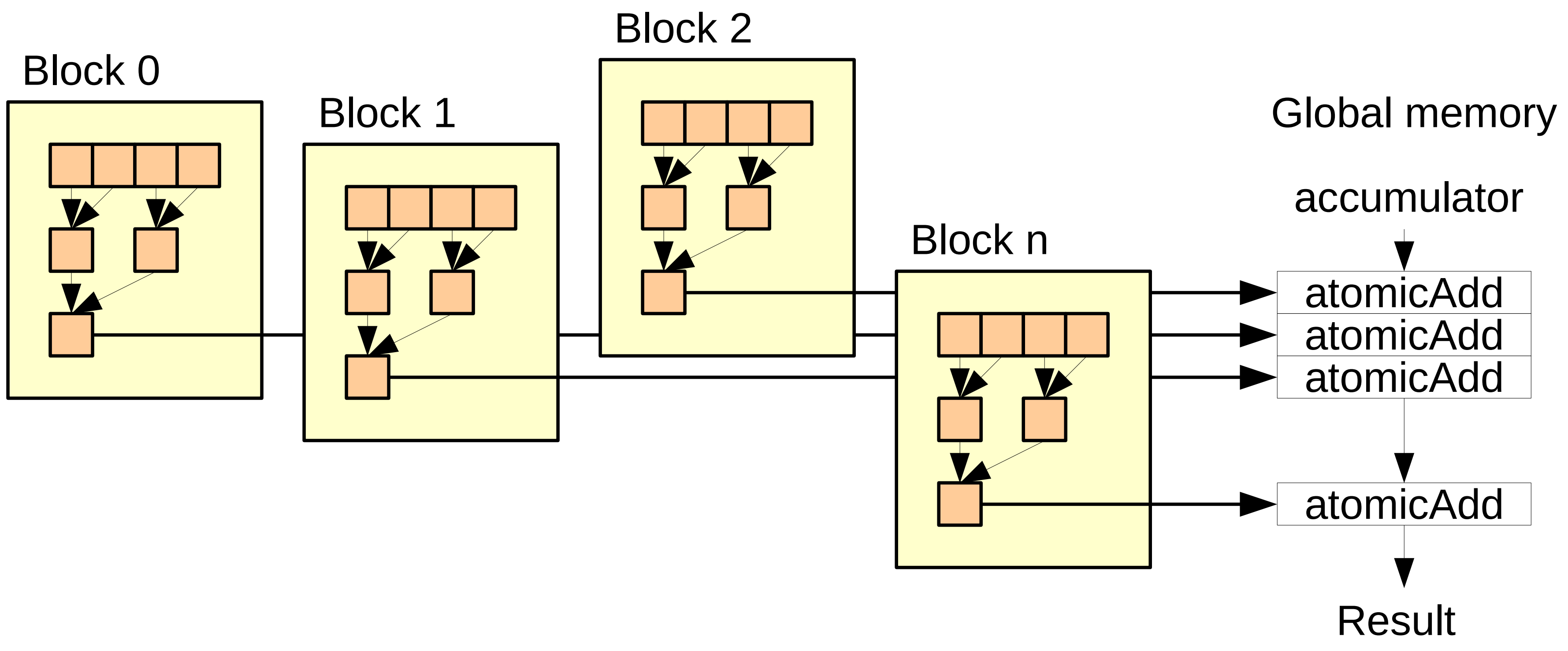
- Instruction-level optimization
  - A few device code features
  - Warp divergence
- Advanced features
  - Device-mapped memory, Unified virtual address space, Unified memory
  - Global and shared memory atomics
  - Warp-synchronous programming, warp vote, shuffle

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
  - From CC 1.1
- On shared memory
  - From CC 1.2
- Performance impact in case of contention

Example: reduction

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



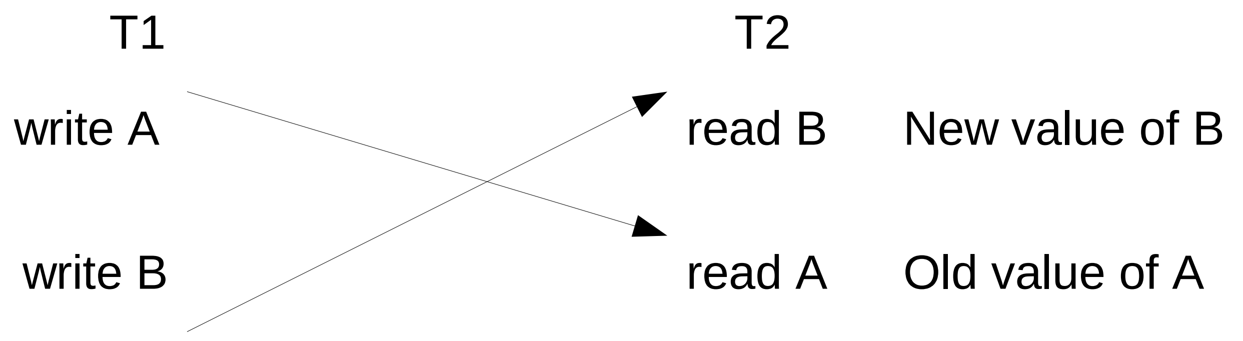
- Complexity?
- Time including kernel launch overhead?

Floating-point atomics

- atomicAdd supports single-precision floating-point (float) operands
- Remember floating-point addition is not associative
  - You will get a different answer depending on the scheduling

Memory consistency model

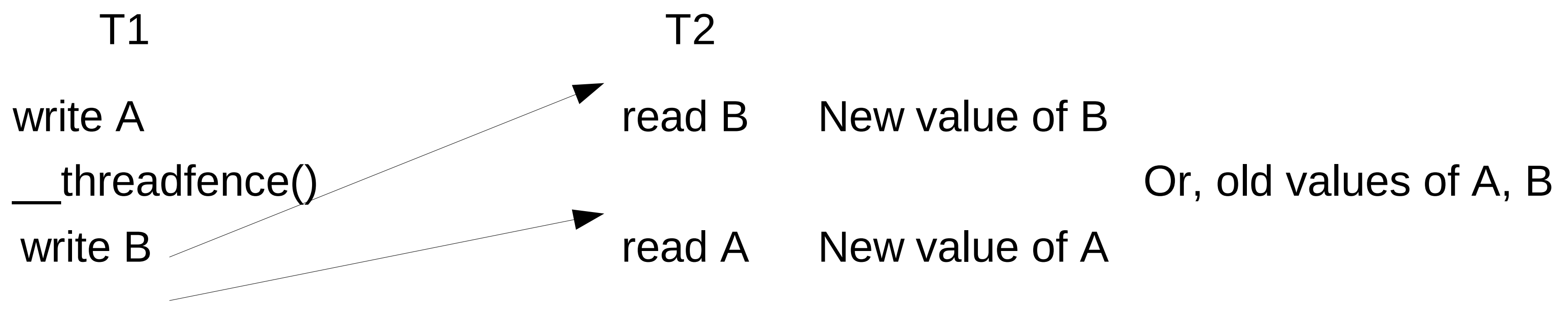
- x86 CPUs implement a strong consistency model
  - Pretend there is a global ordering between memory accesses
- Nvidia GPUs implement a relaxed consistency model
  - Threads may not see the writes/atomics in the same order



- Need to enforce explicit ordering

Memory consistency model

- `__threadfence_block`
  `__threadfence`
  `__threadfence_system`
  make all previous writes of the thread visible
  at the block / device / system level

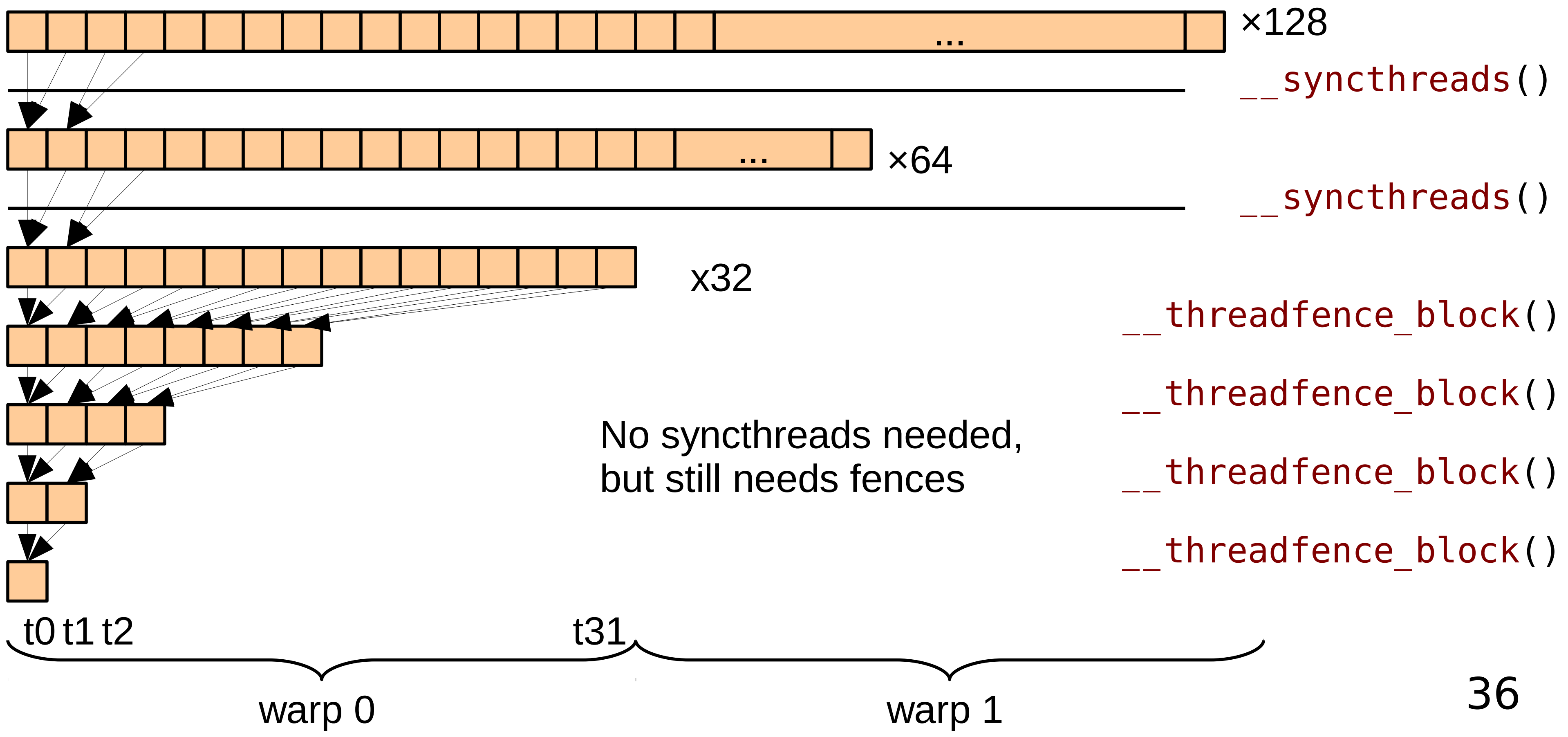


Outline

- Instruction-level optimization
  - A few device code features
  - Warp divergence
- Advanced features
  - Device-mapped memory, Unified virtual address space, Unified memory
  - Global and shared memory atomics
  - Warp-synchronous programming, warp vote, shuffle

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

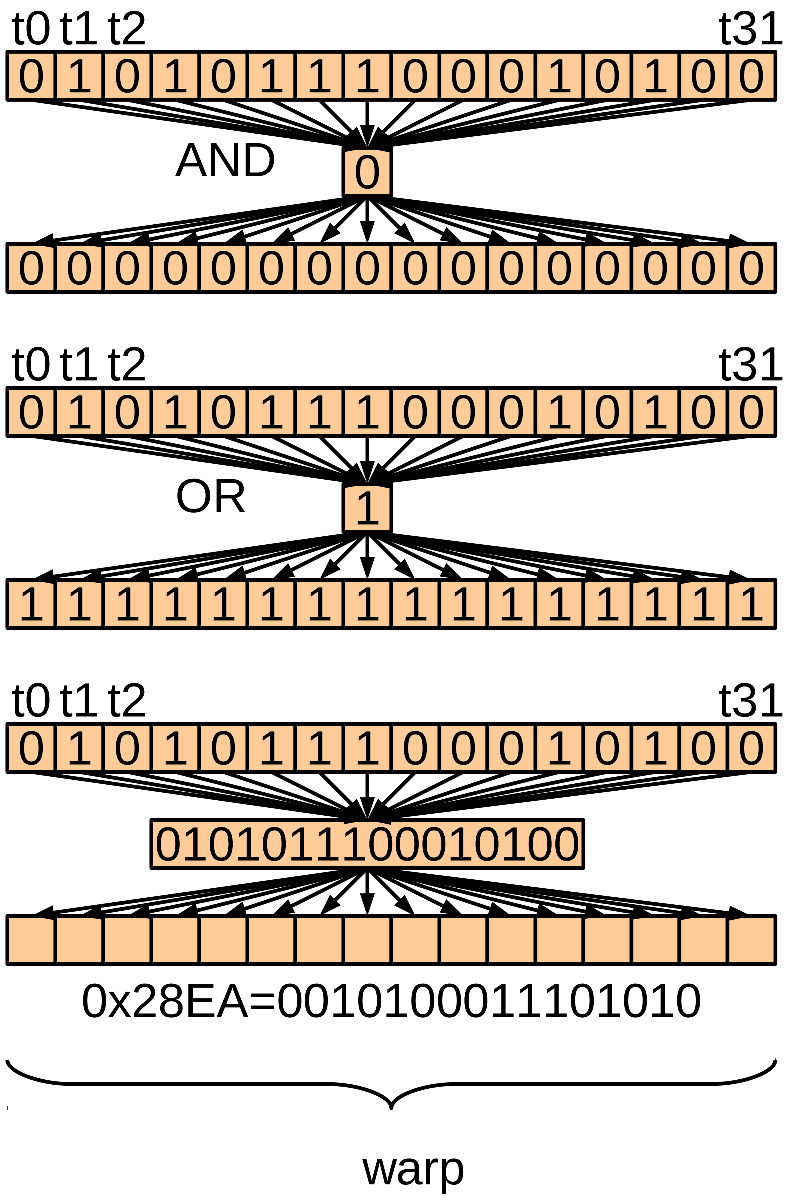


Warp-synchronous programming: tools

- We need warp size, and thread ID inside a warp: lane ID
- Official support
  - Predefined variable: `warpSize`
  - Lane ID exists in PTX, not in C for CUDA
    Needs to be computed:
    ```
    unsigned int laneId = threadIdx.x % warpSize;
    ```
- Note: as of CUDA 5.0, this is essentially useless
  - warpSize is a variable in PTX, only becomes a constant in SASS
  - PTX optimizer does not know warp size is a power of 2:
    does not turn % into shift
- Often use a WARP_SIZE constant hardcoded to 32…

Warp vote instructions

- `p2 = __all(p1)`
  horizontal AND between the 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
For CC ≥ 2.0

Manual divergence management

- How to write an if-then-else in warp-synchronous style?
  i.e. without breaking synchronization

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

  - Using predication:
    execute both sides always

```
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(); }
}
```

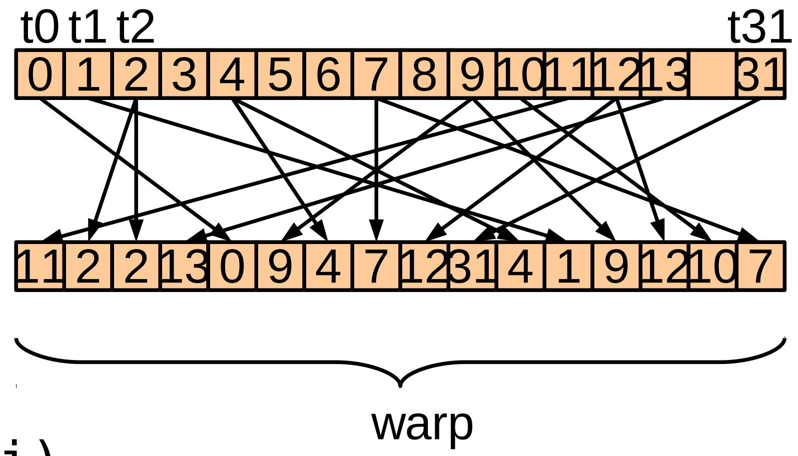
- How to write a while loop?

Shuffle

Exchange data between lanes

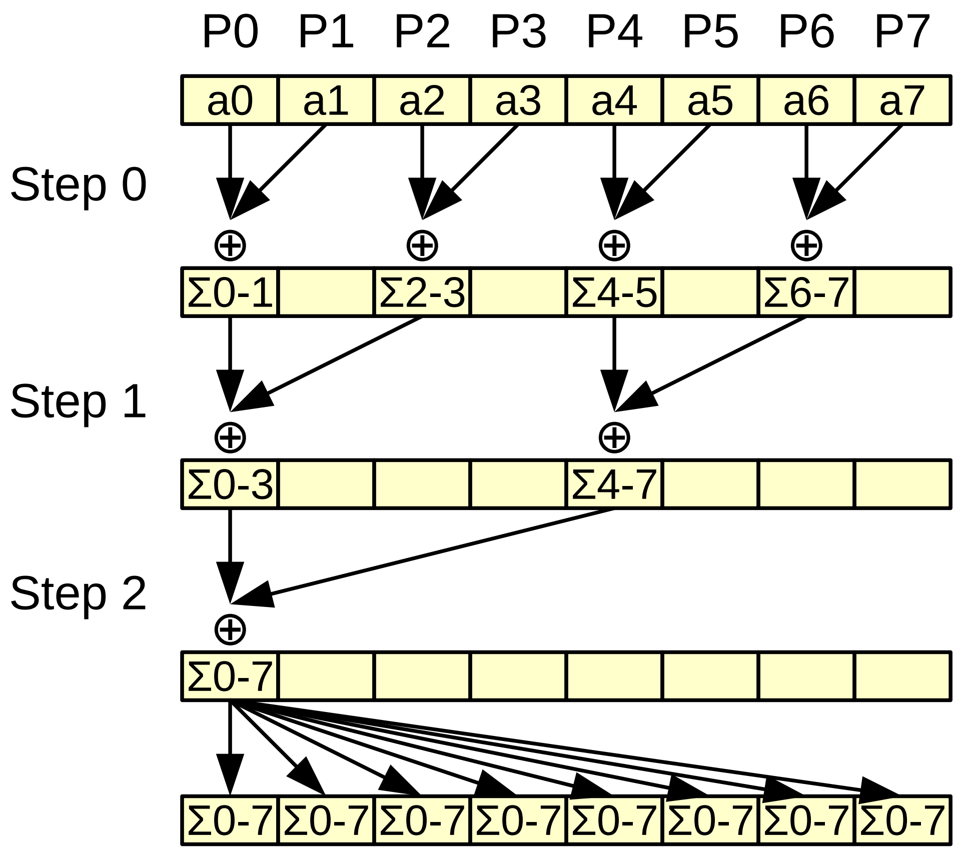
- `__shfl(v, i)`
  Get value of thread i in the warp
  - Use: 32 concurrent lookups in a 32-entry table
  - Use: arbitrary permutation...
- `__shfl_up(v, i) ≈ __shfl(v, tid-i),`
  `__shfl_down(v, i) ≈ __shfl(v, tid+i)`
  Same, indexing relative to current lane
  - Use: neighbor communication, shift
-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

For CC ≥ 3.0



Example: reduction + broadcast

- Naive algorithm

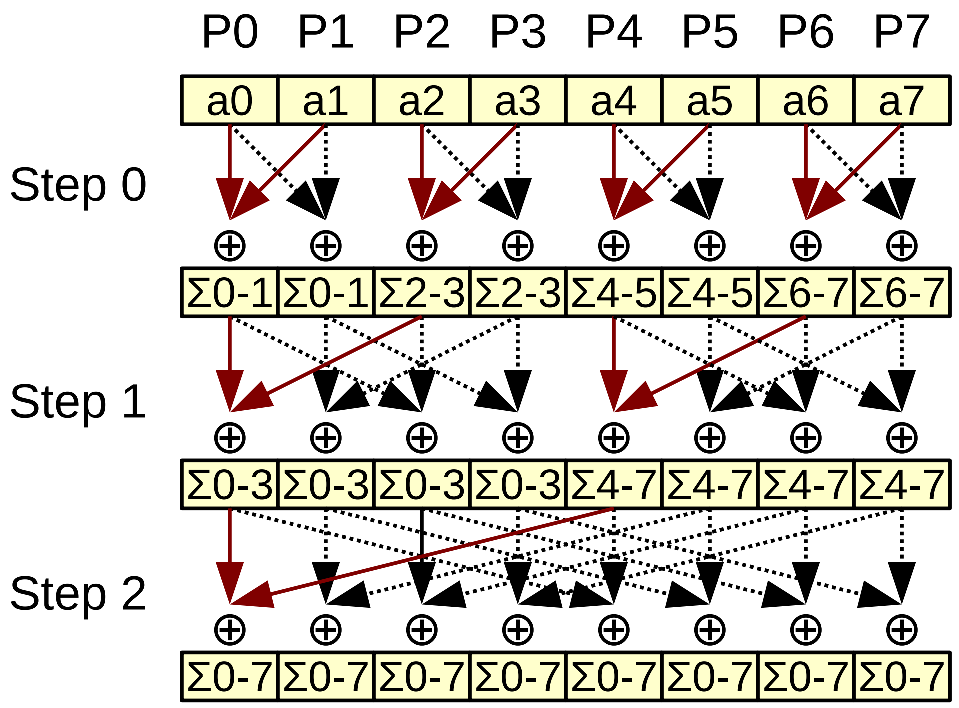


```
a[2*i]←a[2*i] + a[2*i+1]

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



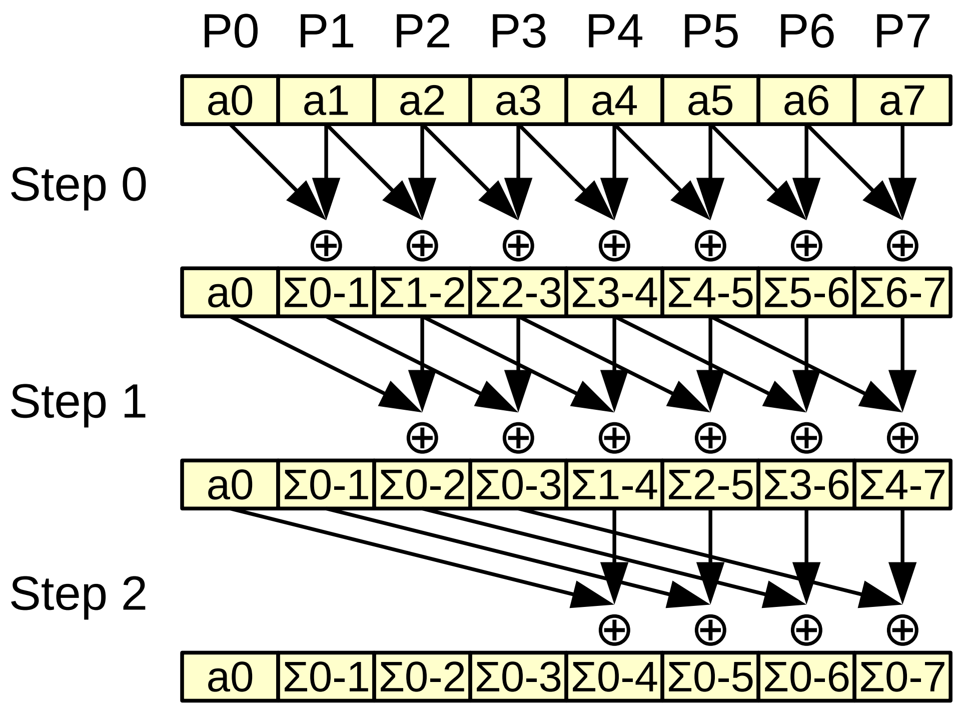
P0 P1 P2 P3 P4 P5 P6 P7
a0 a1 a2 a3 a4 a5 a6 a7
Step 0
Σ0-1 Σ0-1 Σ2-3 Σ2-3 Σ4-5 Σ4-5 Σ6-7 Σ6-7
Step 1
Σ0-3 Σ0-3 Σ0-3 Σ0-3 Σ4-7 Σ4-7 Σ4-7 Σ4-7
Step 2
Σ0-7 Σ0-7 Σ0-7 Σ0-7 Σ0-7 Σ0-7 Σ0-7 Σ0-7

```
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



P0 P1 P2 P3 P4 P5 P6 P7
a0 a1 a2 a3 a4 a5 a6 a7
Step 0
a0 Σ0-1 Σ1-2 Σ2-3 Σ3-4 Σ4-5 Σ5-6 Σ6-7
Step 1
a0 Σ0-1 Σ0-2 Σ0-3 Σ1-4 Σ2-5 Σ3-6 Σ4-7
Step 2
a0 Σ0-1 Σ0-2 Σ0-3 Σ0-4 Σ0-5 Σ0-6 Σ0-7

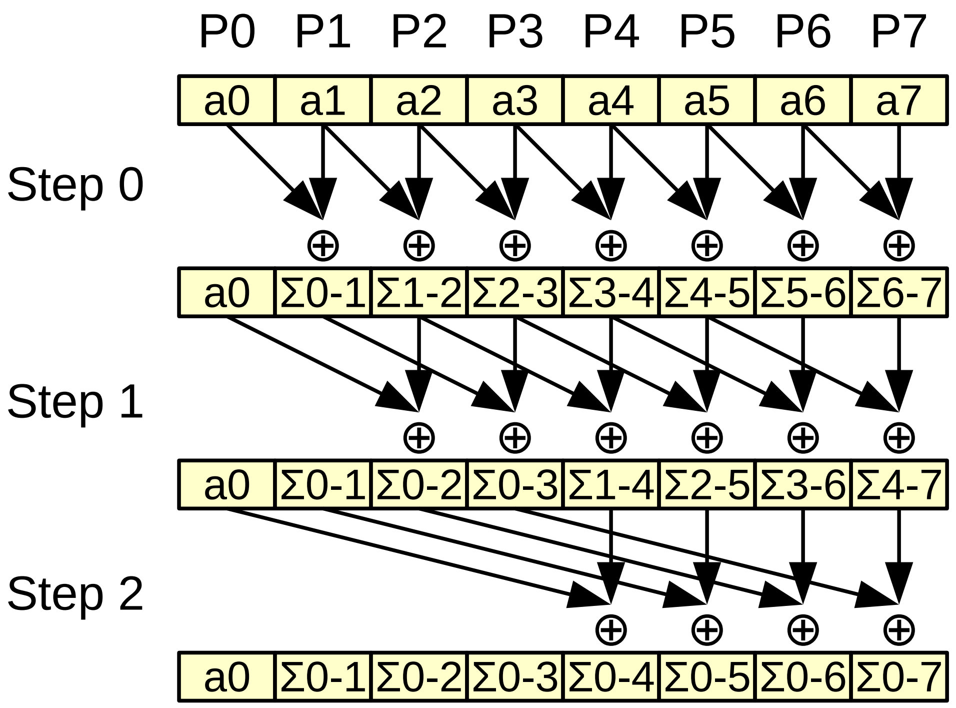
```
s[i] ← a[i]
if i ≥ 1 then
    s[i] ← s[i-1] + s[i]

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



P0 P1 P2 P3 P4 P5 P6 P7
a0 a1 a2 a3 a4 a5 a6 a7
Step 0
⊕ ⊕ ⊕ ⊕ ⊕ ⊕ ⊕
a0 Σ0-1 Σ1-2 Σ2-3 Σ3-4 Σ4-5 Σ5-6 Σ6-7
Step 1
⊕ ⊕ ⊕ ⊕ ⊕ ⊕
a0 Σ0-1 Σ0-2 Σ0-3 Σ1-4 Σ2-5 Σ3-6 Σ4-7
Step 2
⊕ ⊕ ⊕ ⊕
a0 Σ0-1 Σ0-2 Σ0-3 Σ0-4 Σ0-5 Σ0-6 Σ0-7

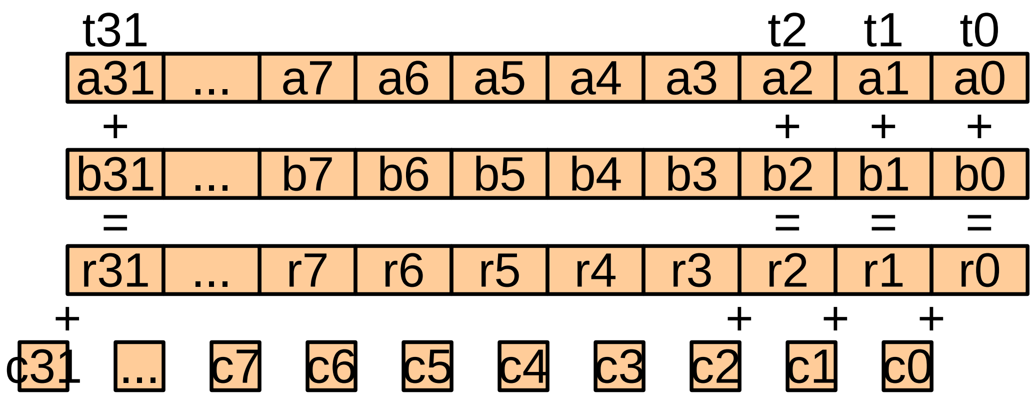
```
s = a;
n = __shfl_up(s, 1);
if(laneid >= 1)
    s += n;
n = __shfl_up(s, 2);
if(laneid >= 2)
    s += n;
n = __shfl_up(s, 4);
if(laneid >= 4)
    s += n;
```

Σi-j is the sum $\sum_{k=i}^{j} a_k$

```
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

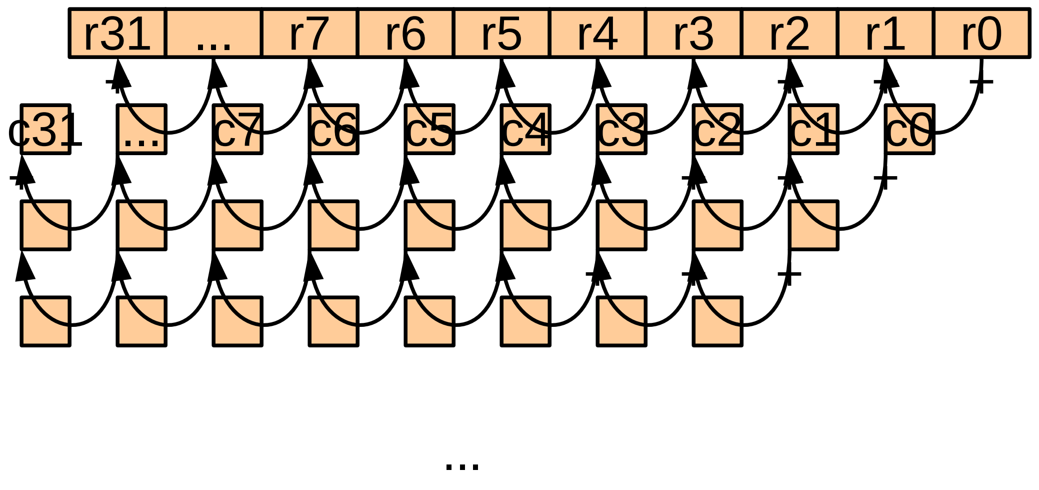


t31
t2
t1
t0
a31 ... a7 a6 a5 a4 a3 a2 a1 a0
+
b31 ... b7 b6 b5 b4 b3 b2 b1 b0
=
r31 ... r7 r6 r5 r4 r3 r2 r1 r0
+
c31 ... c7 c6 c5 c4 c3 c2 c1 c0

```
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;
```

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”, explicitely synchronizable
  - Threads in warp: implicitely synchronized

Things we have not talked about

- Constant memory
  - Memory-space that is read-only on the device
  - Being replaced by cached, read-only access to global memory on recent GPUs
- Texture memory
  - Cached, read-only memory space optimized for 2D locality
  - Can unpack compact integer and floating-point encoded data
  - Can perform filtering: interpolation between data points
- Dynamic parallelism
  - Starting from CC 3.5, kernels can launch kernels

Conclusion: trends

- GPU: rapidly evolving hardware and software
- Going towards CPU-GPU tighter coupling
  - On-chip integration
  - Shared physical memory
  - Shared virtual memory space
- Most development is going into mobile
  - Nvidia: Kepler GPUs in Tegra 4 support CUDA
  - Many GPUs supporting OpenCL:
    ARM Mali, Qualcomm Adreno, Imagination Technologies PowerVR Rogue...
- Still much work to do at the operating system level
  - GPU currently a second-class citizen
  - Need to move from CPU+devices model to heterogeneous compute model

Updated schedule

Week	Course: Tuesday 1:00pm room 2014	Lab: Thursday 1:00pm room 2011
06/10	Programming models	Parallel algorithms
13/10	GPU architecture 1	Know your GPU
20/10	GPU programming	Computing ln(2) the hard way
27/10	GPU optimization 1	
03/11	GPU optimization 2	Game of life
10/11	Advanced features	
17/11	Lecture by Fernando Pereira	
24/11	Exam	Project
01/12	Project presentations	

Project

- For next week: choose a CUDA programming project
- Work by yourself or in pair
- Ideas of subjects
  - Image processing: filtering, convolution
  - Cryptography: brute-force password “recovery”
  - Data compression

  - Any proposition?

References and further reading/watching

- CUDA C Programming Guide
- Mark Harris. Introduction to CUDA C. http://developer.nvidia.com/cuda-education
- David Luebke, John Owens. Intro to parallel programming. Online course. https://www.udacity.com/course/cs344
- Paulius Micikevicius. GPU Performance Analysis and Optimization. GTC 2012. http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf

Pitfalls and fallacies

GPUs are 100x faster than CPUs

GPUs are 100x faster than CPUs

- Wrong
  - The gap in peak performance (compute and memory) is 10x
  - In practice, 5x to 8x against optimized CPU code
- Right: you can get a 100x speedup
  when porting an application to GPU
  - You get 10x because of the higher GPU throughput
  - ... and 10x more because you optimized the application
    But spending the same effort on optimization for CPU
    would also result in 10x improvement
- Right: if you use GPU hardware features not available on CPUs
  - Texture filtering, elementary functions (exp, log, sin, cos)...

Victor W. Lee, et al. "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU." ISCA 2010.

GPUs have thousands of cores

GPUs have thousands of cores

- Official definition of CUDA Core: one execution unit
  - Right: GPUs have thousands of cores
  - Under this metric, CPUs have hundreds of cores !
- The closest to a CPU core is a GPU SM
  - Wrong: 15 to 20 SMs “only” on a GPU
  - Wide SIMD units inside
    But just x2 to 4x wider than on CPU
- Often-overlooked aspect:
  GPUs have 10s of thousands of threads
  - But: CPU thread ~ GPU warp
  - Still thousands of warps

I can avoid divergence with predication

I can avoid divergence with predication

Do not confuse predication with prediction !

- In a superscalar CPU, branches are predicted
  - Cost of misprediction ~ pipeline depth
    independent from length of mispredicted path
  - Inefficient on **short** ifs governed by **random** conditions
    Predication **might** help in this case
- In a GPU, instructions are predicated
  - Cost of divergence ~ divergent section size
  - Inefficient on **long** ifs/loops governed by divergent conditions
    Predication **does not** help in this case

Texture memory

- Primarily made for graphics
- Optimized for 2D accesses
    - Takes advantage of locality in space
- Read-only
- Initialization through specific host functions
    - `cudaCreateTextureObject()`, `cudaBindTexture2D()`...
- Access through specific device functions
    - `tex1D()`, `tex2D()`, `tex3D()`...

Constant memory

- Array or variable declared as __const__
- Initialized in place or from host code, then read-only
- Single-ported memory: good performance when all threads in a warp access the same word (broadcast)
  - Similar to shared memory with 1 bank
  - Generally not worth using with divergent accesses