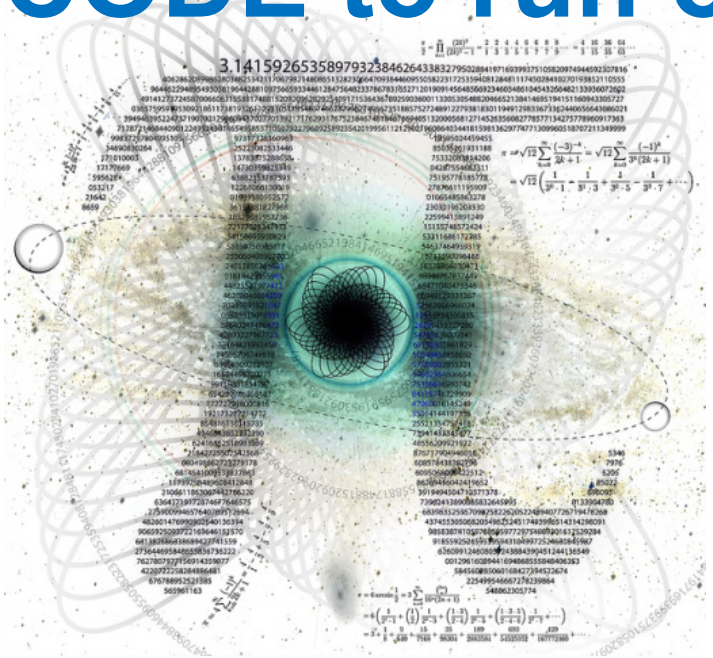
femto-st SCIENCES & TECHNOLOGIES

PARALLELIZING
a CPU CODE to run on GPU



gilles.perrot@univ-fcomte.fr

UFC UNIVERSITÉ DE FRANCHE-COMTÉ · cnrs · ensmm · utbm · UBFC

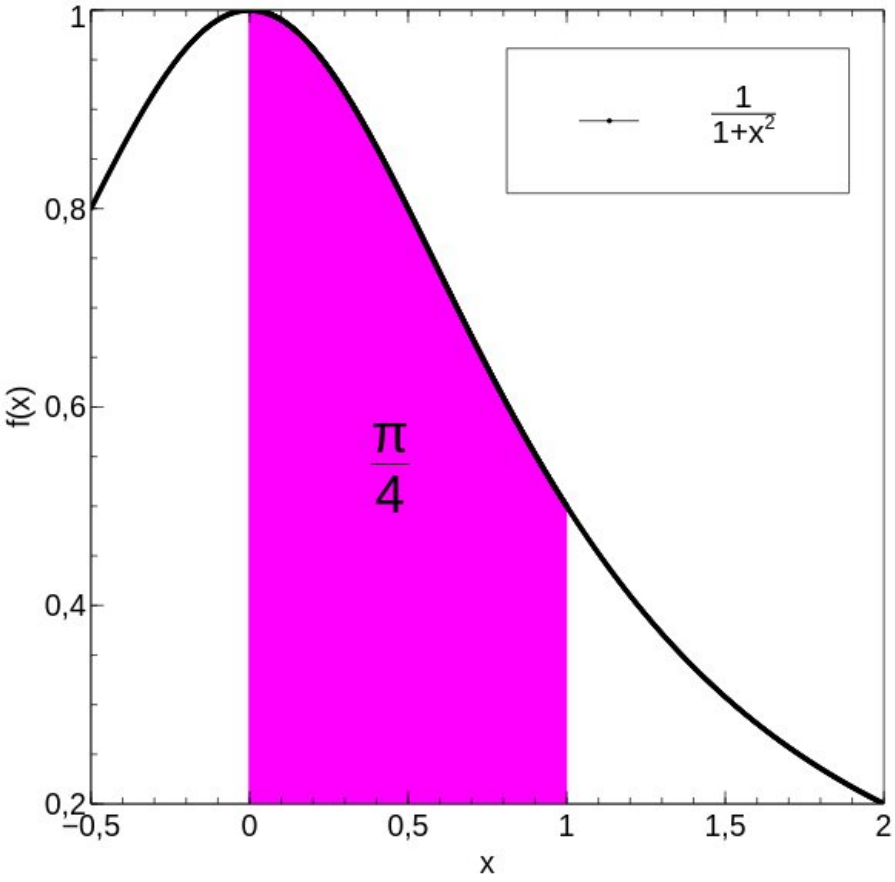
Summary

- Sequential code : compilation - profiling
- Simple automatic parallelization : openPGI + openACC
- Parallelization with CUDA : from naive to highly optimized
  - Naive reduction run by only one thread.
  - Reduction using shared memory.
  - Hybrid memory usage : registers/shared memory.
  - Intra-warp communication (shuffle instructions).

CPU code : algorithm

Simple algorithm, based on equation → $\int_0^1 \frac{1}{1+x^2}.dx = \frac{\pi}{4}$

- The outilne of the algorithm is to cut the interval [0 ; 1] into `nsteps` slices, considered thin enough.
- We assume that in each slice, the shape under the curve is rectanguar.
- Height is that of the middle point of the slice.
- The greater `nsteps`, the more accurate the result will be.
- In this case, the computation error is zero when `nsteps` tends towards infinity.



An estimation of π : the sequential version

Code c++ : pic.cpp

```
12  int main(int argc, char* argv[])
13  {
14    long i, nsteps;
15    double pi, step, sum = 0.0;
16    nsteps = 0;
17    if (argc > 1)
18      nsteps = atol(argv[1]);
19    if (nsteps <= 0)
20      nsteps = 100;
21    step = (1.0)/((double)nsteps);
22    for (i = 0; i < nsteps; ++i) {
23      double x = ((double)i+0.5)*step;
24      sum += 1.0 / (1.0 + x * x);
25    }
26    pi = 4.0 * step * sum;
27    cout << std::fixed;
28    cout << "pi is " << std::setprecision(17) << pi << "\n";
29  }
```

Compilation with gcc (option -Ofast)

```
$ g++ -Ofast -o pic pic.cpp
```

An estimation of π : the sequential version

Performance measurement : Execution with 10^8 and 10^9 intervals

via the `time` instruction

```
perrot@cluster2:~/samples-9/0_Simple/pic$ g++ -Ofast -o pic pic.cpp
perrot@cluster2:~/samples-9/0_Simple/pic$ time ./pic 1000000000
pi is 3.14159265358997075

real    0m1.446s
user    0m1.445s
sys     0m0.000s
perrot@cluster2:~/samples-9/0_Simple/pic$ time ./pic 100000000
pi is 3.14159265359042639

real    0m0.165s
user    0m0.165s
sys     0m0.000s
```

Via the Nvidia profiler `nvprof`

```
perrot@cluster2:~/samples-9/0_Simple/pic$ nvprof --cpu-profiling on --cpu-profiling-mode flat ./pic 1000000000
pi is 3.14159265358997075

======== CPU profiling result (flat):
Time(%)      Time  Name
 100.00%     1.52s  main

======== Data collected at 100Hz frequency
perrot@cluster2:~/samples-9/0_Simple/pic$ nvprof --cpu-profiling on --cpu-profiling-mode flat ./pic 100000000
pi is 3.14159265359042639

======== CPU profiling result (flat):
Time(%)      Time  Name
 100.00%  149.93ms  main
```

An estimation of π : the sequential version

Compilation with openPGI compiler : `pgc++`

```
$ pgc++ -fast -Minfo=all,intensity,ccff -Minline -o pic pic.cpp
```

The option -Minfo allows to get some useful feedback before trying to parallelize.

Performance measurement… Not all compilations are equivalents.

```
perrot@cluster2:~/samples-9/0_Simple/pic$ time ./pic 1000000000
pi is 3.14159265358976825

real    0m2.362s
user    0m2.358s
sys     0m0.004s
```

At least, results are equals with a precision of 10^{-12}.

Automatic parallelization: openACC

- OpenACC can generate hybrid executable code (CPU/GPU).
- Automatic generation is driven by a set of compilation directives.
- The compiler : openPGI (pgc++).

- Before trying any parallelization, it is mandatory to:
  - Use the profiler to identify the more time consuming sequences.
  - Find out, among that sequences, those that could be parallelized (on a GPU).
  - Determine an appropriate set of ACC directives for each code sequence.

In the sample code `pic.cpp`, the 'for' loop is a candidate for parallelization .

The typical directive to parallelize a C/C++ loop looks like

```
#pragma  acc parallel loop
```

And has to be placed just above the target loop

```
20  #pragma acc parallel loop
21    for (i = 0; i < nsteps; ++i) {
22      double x = ((double)i+0.5)*step;
23      sum += 1.0 / (1.0 + x * x);
24    }
```

Automatic parallelization: openACC

Compilation

```
$ pgc++ -fast -Minfo=all,intensity,ccff -Minline \
                                -ta=tesla:cuda9.2 -o pic pic.cpp
```

- The -ta option allows to specify a target for the parallel code.
- In our case, we will target a Nvidia Tesla family GPU, Pascal generation, Titan-X model.
- On the host computer, the sdk release version is cuda9.2.

Execution

```
perrot@cluster2:~/samples-9/0_Simple/pic$ time ./pic 1000000000
pi is 3.14159265358979312

real    0m0.265s
user    0m0.098s
sys     0m0.163s
```

Automatic parallelization: openACC

The compiler output shows that one reduction has been identified (on sum) and that compliant code has then been generated.

However, one can write the corresponding directive more explicitely

```
20  #pragma acc parallel loop reduction(+:sum)
21    for (i = 0; i < nsteps; ++i) {
22      double x = ((double)i+0.5)*step;
23      sum += 1.0 / (1.0 + x * x);
24    }
```

Execution

```
perrot@cluster2:~/samples-9/0_Simple/pic$ time ./pic 1000000000
pi is 3.14159265358979312

real    0m0.292s
user    0m0.079s
sys     0m0.210s
```

Speedup achieved :

- x30 against the sequential code and pgc++ compiler
- x18 against the sequential code and g++ compiler

WITHOUT ANY EFFORT (ALMOST)

Automatic parallelization: openACC

A few comments about openACC

- Even common cases can be difficult to solve.
- Managing memory transfers and persistence can be quite challenging.
- Speedups are more difficult to obtain and less impressive.
- A significant overhead is introduced by the compiler, when
  - Transferring data between CPU and GPU,
  - Calling fonctions / kernels,
  - Choosing non optimal grid dimensions.

- The official documentation of openACC directives
  https://www.openacc.org/sites/default/files/inline-files/OpenACC_2_0_specification.pdf

CUDA / C parallelization

Outlines

- Nvidia template as a starting point (0_Simple/template)
- General structure of a CUDA code:
  1. Resource allocations (CPU & GPU).
  2. Data loading into host's memory (CPU).
  3. Data copy from host memory into GPU memory.
  4. Computation of grid dimensions and kernel executions.
  5. Copy of the result data from GPU memory to host memory.

Estimation of π

- Simplified structure; no input data to transfer.
- Step by step design and successive refinements.
- Solutions to some performance limitation parameters.

CUDA : step #1

Idea

- One *kernel* computes the $1/(1+x^2)$ value associated to each slice of the [0 ; 1] interval.
- Each slice value is computed by one thread.
- The number of slices, *nbsteps,* can be great. We need to check the capacities of the GPU regarding the maximum number of concurrent threads.
- For this purpose, one can use the d*eviceQuery program*

```
Device 0: "GeForce GTX TITAN X"
  CUDA Driver Version / Runtime Version          9.2 / 9.0
  CUDA Capability Major/Minor version number:    5.2
  Total amount of global memory:                 12213 MBytes (12806062080 bytes)
  (24) Multiprocessors, (128) CUDA Cores/MP:     3072 CUDA Cores
  GPU Max Clock rate:                            1076 MHz (1.08 GHz)
  Memory Clock rate:                             3505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 3145728 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

- *nbsteps* max >2.10^9 x 1024 = 2.10^12.

CUDA : step #1

Kernel code

- The number of concurrent thread is a power of 2, multiple of 32.
- If *nbsteps* is not a power of 2, it should be passed to the kernel as a parameter.
- If *nbsteps* is a power of 2, its value can be guessed by k*ernels* at runtime.

```
 99  __global__ void
100  testKernel(DTYPE *out_data, unsigned int nsteps)
101  {
102      // thread id inside thread block and inside grid
103      const unsigned int tidb = threadIdx.x;
104      const unsigned int tid = blockIdx.x*blockDim.x + tidb;
105
106      // perform some computations
107      DTYPE step;
108      if (tid < nsteps){
109        step = (1.0)/((DTYPE)nsteps);
110        DTYPE x = ((DTYPE)tid+0.5)*step;
111        out_data[tid] = (DTYPE) 1.0 / (1.0 + x * x);
112      }
113  }
```

CUDA : step #1

Main.cu source code

- Memory allocation on GPU to store *nbsteps* real values.
- Definition of the computing grid dimensions.
- *Kernel execution*.

```
143     unsigned int threads_per_block = 1024;
144     unsigned int nbsteps = atoi(argv[1]);
145     unsigned int mem_size = sizeof(DTYPE) * nbsteps;
146
147     // allocate device memory
148     DTYPE *d_vector;
149     checkCudaErrors(cudaMalloc((void **) &d_vector, mem_size));
150
151     // setup execution parameters
152     dim3  grid((nbsteps+threads_per_block-1)/threads_per_block, 1, 1);
153     dim3  threads(threads_per_block, 1, 1);
154
155     // execute the kernel
156     testKernel<<< grid, threads, 0 >>>(d_vector, nbsteps);
```

CUDA : step #1

Results & performance

- The final sum is processed on the CPU.
- The data transfer GPU → CPU is highly time consuming (2300 ms).
- Processing time for slice values : # 150 ms.

```
perrot@cluster2:~/samples-9/0_Simple/ecmDemo_etape_1_vector$ ./template 1000000000
./template Starting...

GPU Device 0: "GeForce GTX TITAN X" with compute capability 5.2

nbsteps : 1000000000 - Memory size : 4000000000
Memory total : 12806062080 - Memory free : 12566986752
Processing time: 155.380997 (ms)
Pi ref CPU : 3.1415926538
Pi vector GPU : 3.1415926861
```

Conclusion

- The final sum must be computed on the GPU to avoid transferring a large amount of data (on only the overall sum has to be copied in this case).

CUDA : step #2

Idea

- Modifying the *kernel* code in order to compute the sum of all the slice values.
- As a starting point, one can add a sequence that allows thread 0 to add up the *nsteps* values previously computed by the kernel and stored in global memory.

Kernel code

```
43  __global__ void
44  testKernel(DTYPE *data, DTYPE *d_sum, unsigned int nsteps)
45  {
46      // thread id inside thread block and inside grid
47      const unsigned int tidb = threadIdx.x;
48      const unsigned int tid = blockIdx.x*blockDim.x + tidb;
49      // perform some computations
50      DTYPE step;
51      DTYPE sum = 0.0;
52      if (tid < nsteps){
53        step = (1.0)/((DTYPE)nsteps);
54        DTYPE x = ((DTYPE)tid+0.5)*step;
55        data[tid] = (DTYPE)1.0 / (1.0 + x*x);
56      }
57      __syncthreads();
58
59      if(tid == 0){
60        for(unsigned int i =0; i<nsteps; i++){
61          sum += data[i];
62        }
63        *d_sum = sum;
64      }
65  }
```

CUDA : step #2

Results & performance

- The sum is computed on the GPU.
- No more large vector to transfer GPU → CPU.
- Computing time **> 1 minute** for 10^9 slices !!!

```
perrot@cluster2:~/samples-9/0_Simple/ecmDemo_etape2_dummySum$ ./template 1000000000
./template Starting...

GPU Device 0: "GeForce GTX TITAN X" with compute capability 5.2

nbsteps : 1000000000 = 10^9 - Memory size : 8000000000
Memory total : 12806062080 - Memory free : 12566986752
Processing time: 61955.273438 (ms)
Pi GPU 3.1415926536
Pi ref CPU : 3.1415926536
```

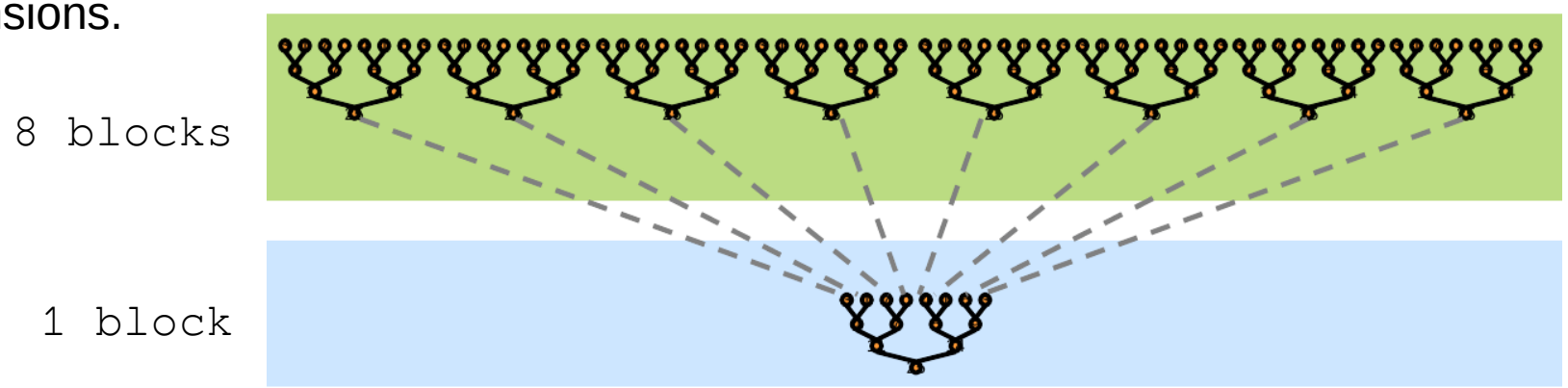
Conclusion

- Only one thread is computing the sum (tid #0, no parallelism).
- Thread #0 waits for all the other threads to complete their tasks before beginning to add up values (__syncthreads()). It is mandatory to avoid adding up wrong values.
- The slice values needed to compute the sum are first stored in global memory, then they are read from global memory by thread #0, one by one.
- Trying to have a GPU to work like a CPU always lead to a disaster.

CUDA : step #3

Idea

- Modifying the *kernel* in order to add up the values more cleverly (with parallelism).
- Storing the values to add up into shared memory, faster than global memory:
  - 48kB of memory shared by thread block are available.
  - Our sum requires 1 double precision value by thread, ie 8 Bytes/thread.
  - A max number of 1024 threads can be define for each block, ie 8 kB/block ⇒ OK (< 48kB).
  - No possible direct inter-block communication ⇒ partial sums in global memory.
- Two options :
  1. Reducing the overall global memory amount required + processing the first summing stage while computing the slice values.
  2. Storing all the slice values into global memory and summing afterwards.
- Option #1 is a bit more efficient but less common ⇒ we choose option #2.
- *We try to write a scalable kernel*, *ie*. that can be launched with various scales of grid dimensions.

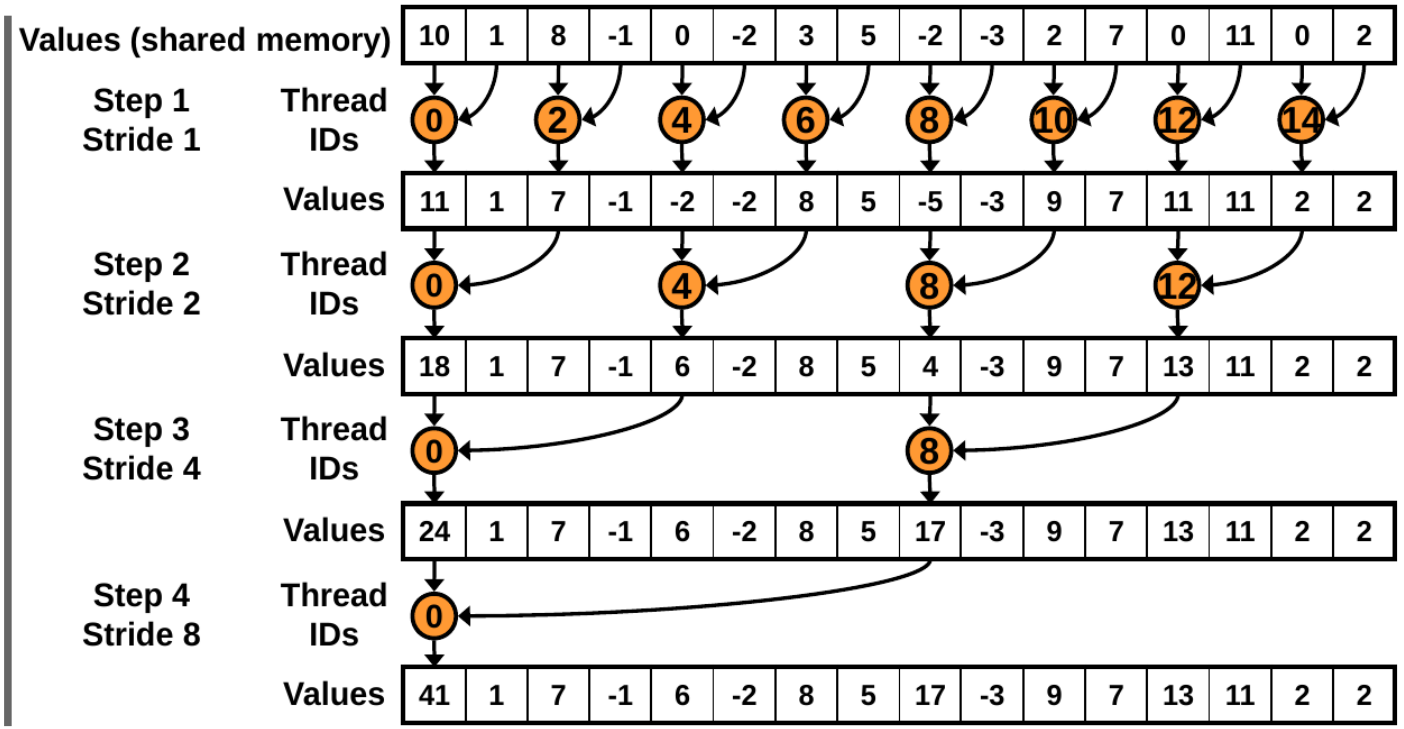


CUDA : step #3

log(n) summing algorithm

- Sums are processed inside each thread block (`threads_per_bloc`).
- Each block sum is then stored into global memory at the very index of the block inside the grid.
- Each kernel execution would divide the size of the vector by `threads_per_bloc`.
- Before each kernel call, suited grid dimensions have to be computed.

Sum inside a block of 16 threads



CUDA : step #3

kernel code (`summup<<<>>>`)

```
62  __global__ void
63  summup(DTYPE * data, unsigned int nvalues)
64  {
65      const unsigned int tidb = threadIdx.x;
66      const unsigned int tid = blockIdx.x*blockDim.x + tidb;
67
68      extern __shared__ DTYPE sdata[];
69
70      // mandatory if nvalues is not a power of 2
71      if (tid < nvalues)
72        sdata[tidb] = data[tid];
73      else
74        sdata[tidb] = 0.0;
75
76      __syncthreads();
77      // loop until all values have been summed up
78      for (int stride = 1; stride < blockDim.x; stride *=2) {
79          if(tidb %(2*stride) == 0)
80            sdata[tidb] += sdata[tidb + stride];
81          __syncthreads();
82      }
83      // writes the sum of the bloc
84      if(tidb == 0)
85        data[blockIdx.x] = sdata[tidb];
86  }
```

CUDA : step #3

Kernel calls `summup<<<>>>`

```
140     computeNstore<<< grid, threads, 0 >>> (d_vector, nbsteps);
141     printf("computeNstore terminated \n");
142
143     int nvalues = nbsteps;
144     int nblocks = (nvalues+threads_per_block-1)/threads_per_block;
145     while(nvalues > 1 ){
146       printf("vector : %d values - %d block(s) of %d threads\n", nvalues,
          nblocks, threads_per_block);
147       summup<<< nblocks, threads_per_block, sizeof(DTYPE)*threads_per_block >>>
          (d_vector, nvalues);
148       nvalues = nblocks;
149       nblocks = (nvalues+threads_per_block-1)/threads_per_block;
150     }
```

CUDA : step #3

Results & performance

- The final sum is stored at index #0 of `d_vector`.
- Overall computing time (memcpy included) # **0,40s** for 10^9 slices (block size =1024)
- Overall computing time (memcpy included) # **0,27s** for 10^9 slices (block size =128)

```
62  __global__ void
63  summup(DTYPE * data, unsigned int nvalues)
64  {
65      const unsigned int tidb = threadIdx.x;
66      const unsigned int tid = blockIdx.x*blockDim.x + tidb;
67
68      extern __shared__ DTYPE sdata[];
69
70      // mandatory if nvalues is not a power of 2
71      if (tid < nvalues)
72        sdata[tidb] = data[tid];
73      else
74        sdata[tidb] = 0.0;
75
76        __syncthreads();
77      // loop until all values have been summed up
78      for (int stride = 1; stride < blockDim.x; stride *=2) {
79          if(tidb %(2*stride) == 0)
80            sdata[tidb] += sdata[tidb + stride];
81          __syncthreads();
82      }
83      // writes the sum of the bloc
84      if(tidb == 0)
85        data[blockIdx.x] = sdata[tidb];
86  }
```

Divergent Warps
Modulo operator

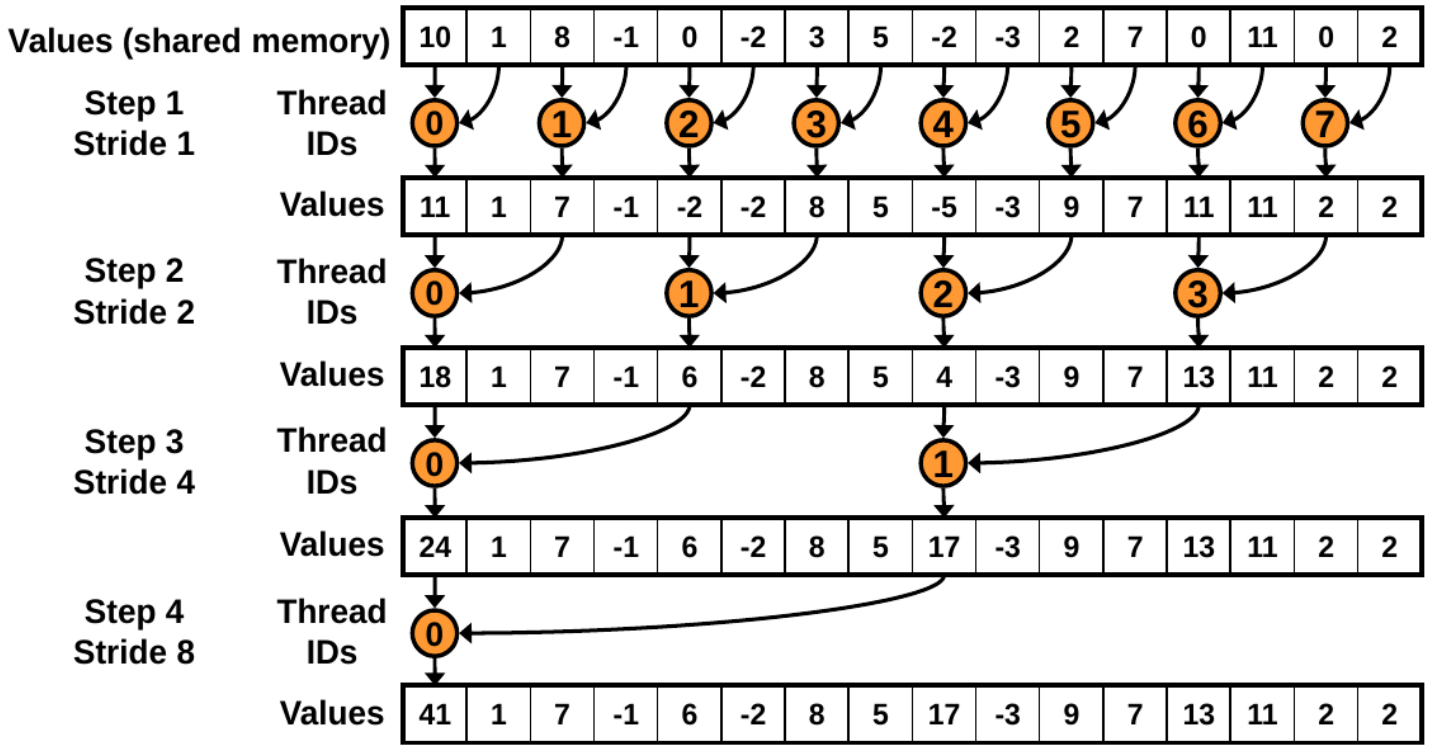
CUDA : step #4

Computing the sum inside thread blocks

- No more divergent warps, neither modulo operators

```
77      // loop until all values have been summed up
78      for (int stride = 1; stride < blockDim.x; stride *=2) {
79          int idx = 2*stride*tidb;
80          if(idx < blockDim.x)
81            sdata[idx] += sdata[idx + stride];
82          __syncthreads();
83      }
```

Sum inside one block of 16 threads



Performances summary

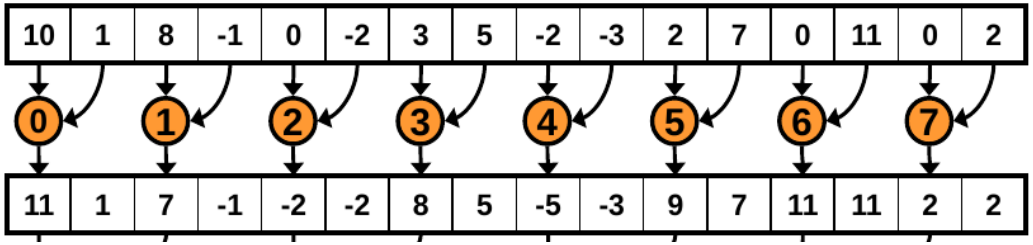
- Estimation of π over 10^9 slices;
- Double precision real computation;
- Memory throughput (D)
  - maximum Titan-X : bus width x clock frequency/2
  - Dmax = 384 x 3505/2 8 = **336 GB/s**
  - D = nbsteps x 8 Bytes / duration

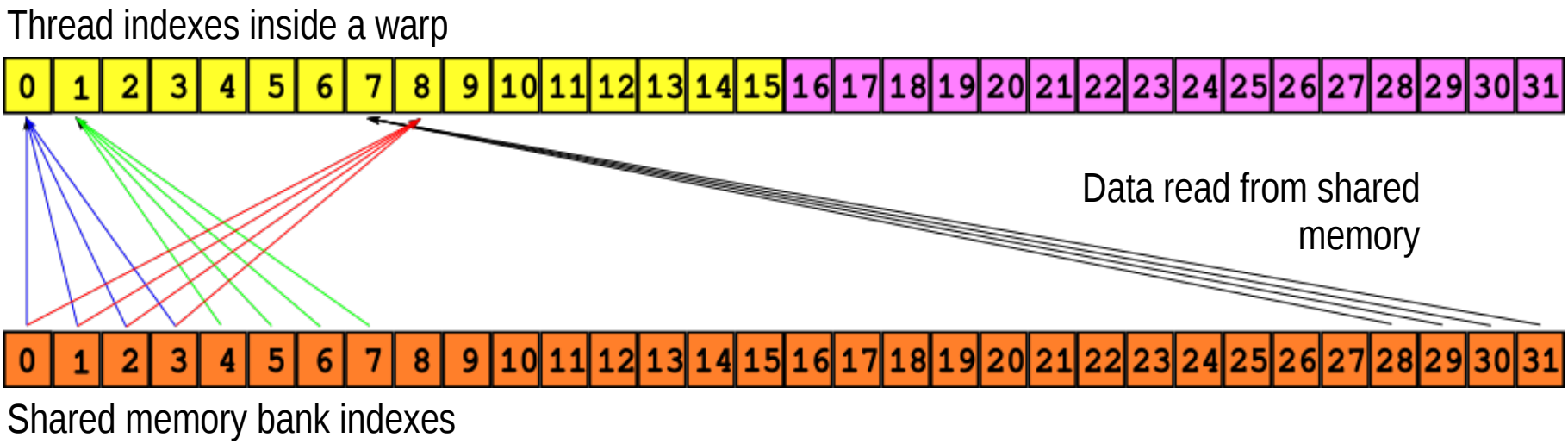
Version	Duration	Throughput
CPU gcc	1440 ms	5.5 GB/s
CPU openPGI	2358 ms	3.4 GB/s
GPU openPGI	270 ms	30 GB/s
GPU CUDA v3	270 ms	30 GB/s
GPU CUDA v4	220 ms	36.4 GB/s

CUDA : step #5

Bank conflicts in shared memory

- The shared memory is implemented through 32 4-Byte-width banks.
- Two threads belonging to two different half-warps (#O-15 et #16-31) accessing two different datas of the same bank will result in a bank conflict.
- During reading stage for `stride=1`, each thread reads 2 doubles (16 Bytes).
- Each double spreds out on 2 banks.
- 4-ways bank conflicts cannot be avoided.

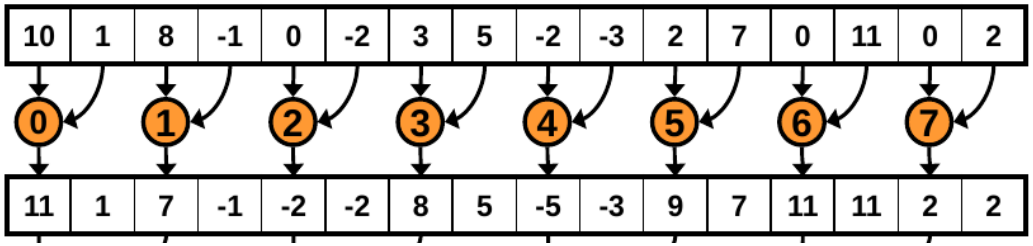


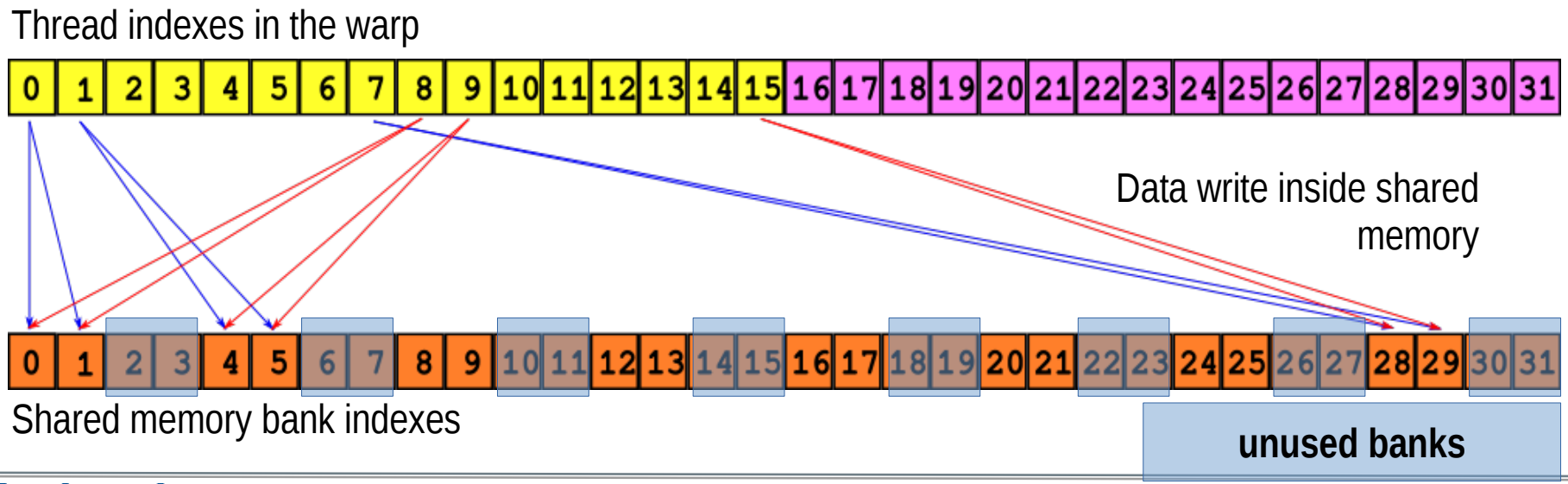


CUDA : step #5

Shared memory bank conflicts

- At writing stage for `stride=1`, each thread writes 1 double (8 Bytes).
- There's an unused memory space between each written value.
- 2-way bank conflict if nothing is done.

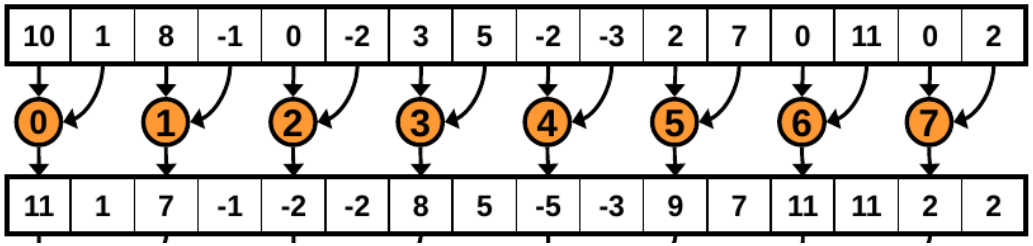


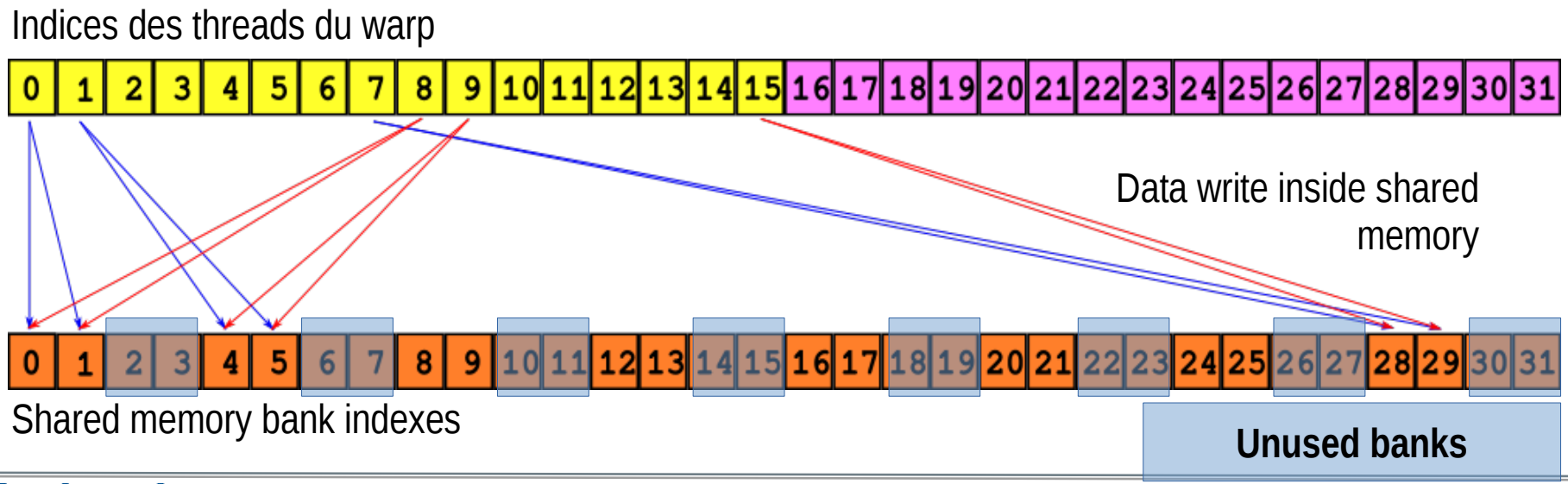


CUDA : step #5

Shared memory bank conflicts

- For threads #8-15, #24-31, etc. we add an offset of 1 to the memory address.
- New indexe computation
  - `shid = tidb + (tidb>>3)&0x1`

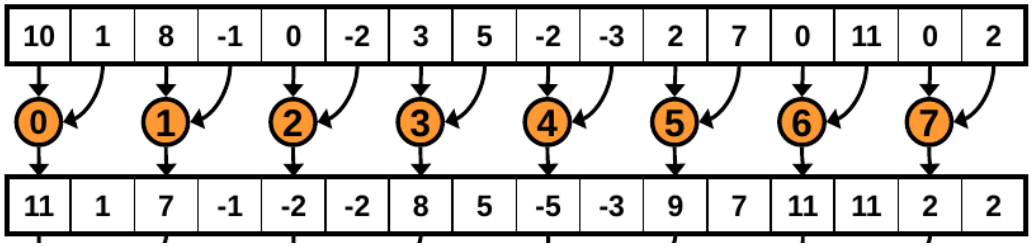


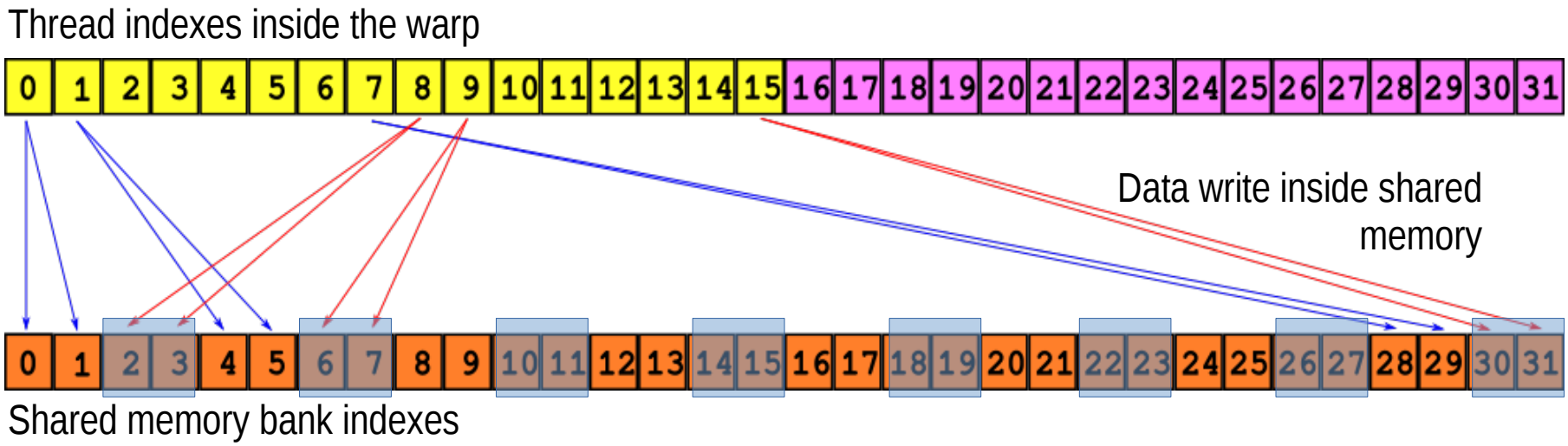


CUDA : step #5

Shared memory bank conflicts

- For threads #8-15, #24-31, etc. we add an offset of 1 to the memory address.
- New indexe computation
  - `shid = tidb + (tidb>>3)&0x1`

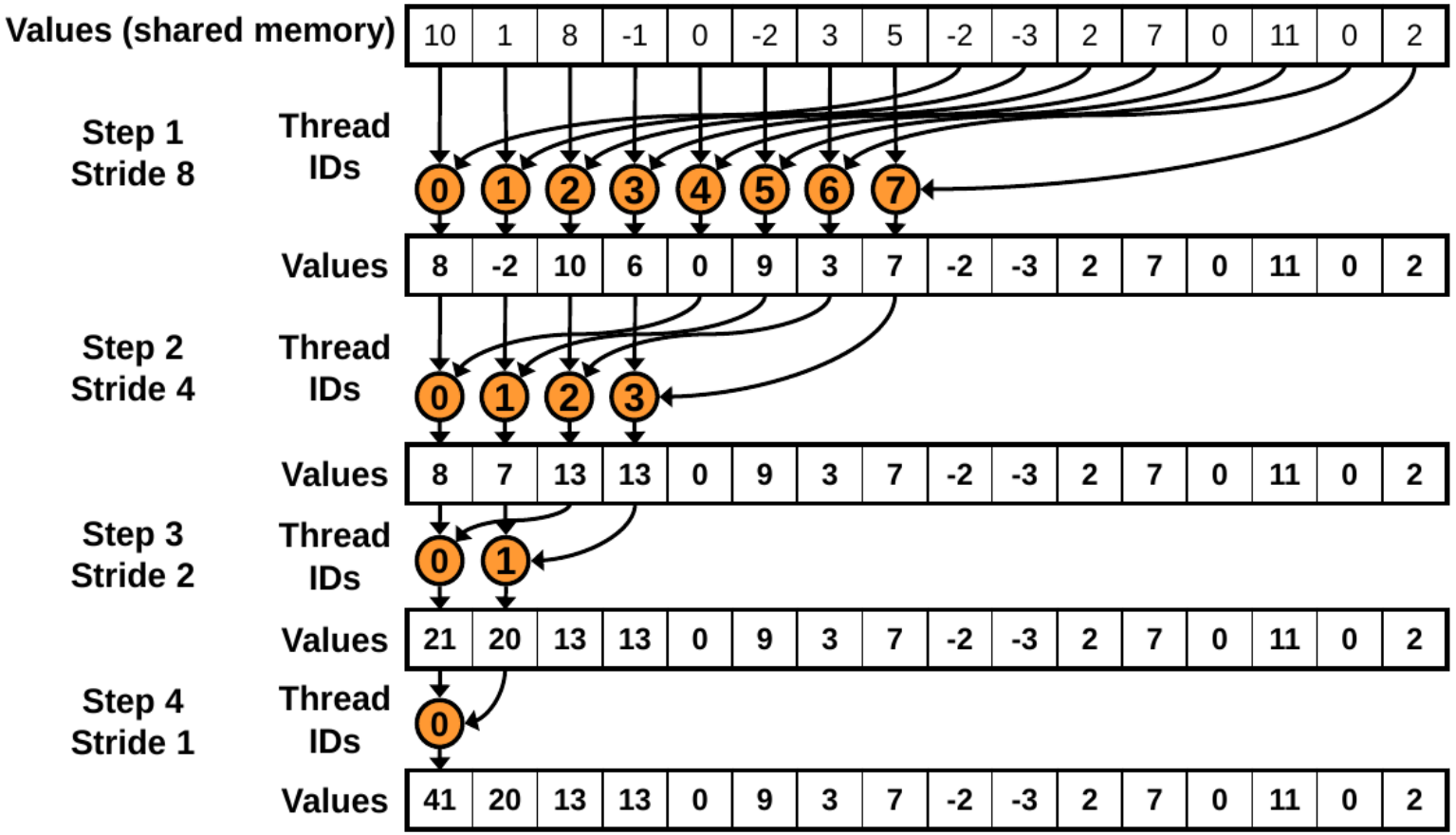




CUDA : step #5

Shared memory bank conflicts

- When computing in single precision, its relevant to address the shared memory in a contiguous way ⇒ no bank conflicts.
- In our case, it would not bring additional performance (v6).



Performance summary

- Estimation of π over 10^9 slices;
- Double precision real computation;
- Memory throughput (D)
  - maximum Titan-X : bus width x clock frequency/2
  - Dmax = 384 x 3505/2 8 = **336 GB/s**
  - D = nbsteps x 8 Bytes / duration

Version	Duration	Throughput
CPU gcc	1440 ms	5.5 GB/s
CPU openPGI	2358 ms	3.4 GB/s
GPU openPGI	270 ms	30 GB/s
GPU CUDA v3	270 ms	30 GB/s
GPU CUDA v4	220 ms	36.4 GB/s
GPU CUDA v5	200 ms	40,0 GB/s

CUDA : step #7

Idea

- With kernel v5, lot of threads are idle.
- Half of all the threads are idle as soon as the very first iteration.
- Combining the first addition with the data loading into global memory.
- Caution if *nbsteps* is not a multiple of 2.
- Padding the vector size to the next power of 2 (`nextPow2()` function).

Modified kernel code

```
67      const unsigned int tidb = threadIdx.x;
68      const unsigned int tid = blockIdx.x*blockDim.x + tidb;
69      const unsigned int offset = nvalues>>1;
70
71      extern __shared__ DTYPE sdata[];
72
73      if (tid < offset)
74        sdata[SHID(tidb)] = data[tid] + data[tid + offset];
75      else
76        sdata[SHID(tidb)] = 0.0;
```

CUDA : step #7

Modified kernel calls

```
148     int nvalues = nbsteps;
149     nblocks = (nvalues/2+threads_per_block-1)/(threads_per_block);
150     while(nvalues > 1 ){
151       printf("vector : %d values - %d block(s) of %d threads\n", nvalues,
  •          nblocks, threads_per_block);
152       summup<<< nblocks, threads_per_block,
  •          sizeof(DTYPE)*(threads_per_block+(threads_per_block>>3)) >>>
  •          (d_vector, nvalues);
153       nvalues = nblocks;
154       nblocks = (nvalues/2+threads_per_block-1)/threads_per_block;
155     }
```

Performance summary

- Estimation of π over 10^9 slices;
- Double precision real computation;
- Memory throughput (D)
  - maximum Titan-X : bus width x clock frequency/2
  - Dmax = 384 x 3505/2 8 = **336 GB/s**
  - D = nbsteps x 8 Bytes / duration

Version	Duration	Throughput
CPU gcc	1440 ms	5.5 GB/s
CPU openPGI	2358 ms	3.4 GB/s
GPU openPGI	270 ms	30 GB/s
GPU CUDA v3	270 ms	30 GB/s
GPU CUDA v4	220 ms	36.4 GB/s
GPU CUDA v5	200 ms	40,0 GB/s
GPU CUDA v7	190 ms	42,1 GB/s

CUDA : step #8

Idea

- Iterations after iteration, less and less threads are active.
- When stride ≤ 32, only one warp is still active.
- Inside a warp, as all instructions are executed synchronously
  - ⇒ `__syncthreads()` is now useless
  - ⇒ `if(tidb < stride)` is now useless

Modified kernel code

```
89      for (int stride = blockDim.x/2; stride >32; stride >>=1) {
90          if(tidb < stride)
91            sdata[SHID(tidb)] += sdata[SHID(tidb + stride)];
92          __syncthreads();
93      }
94      if (tidb < 32) warpReduce(sdata, tidb);
```

```
64  __device__ void warpReduce(volatile DTYPE * sdata, int tidb){
65    sdata[SHID(tidb)] += sdata[SHID(tidb +32)];
66    sdata[SHID(tidb)] += sdata[SHID(tidb +16)];
67    sdata[SHID(tidb)] += sdata[SHID(tidb + 8)];
68    sdata[SHID(tidb)] += sdata[SHID(tidb + 4)];
69    sdata[SHID(tidb)] += sdata[SHID(tidb + 2)];
70    sdata[SHID(tidb)] += sdata[SHID(tidb + 1)];
71  }
```

Performance summary

- Estimation of π over 10^9 slices;
- Double precision real computation;
- Memory throughput (D)
  - maximum Titan-X : bus width x clock frequency/2
  - Dmax = 384 x 3505/2 8 = **336 GB/s**
  - D = nbsteps x 8 Bytes / duration

Version	Duration	Throughput
CPU gcc	1440 ms	5.5 GB/s
CPU openPGI	2358 ms	3.4 GB/s
GPU openPGI	270 ms	30 GB/s
GPU CUDA v3	270 ms	30 GB/s
GPU CUDA v4	220 ms	36.4 GB/s
GPU CUDA v5	200 ms	40,0 GB/s
GPU CUDA v7/v8	190 ms	42,1 GB/s

CUDA : step #9

Idea

- Optimising the kernel `<<<computeNstore>>>` by using specific instructions.

```
 97  computeNstore(DTYPE *out_data, unsigned int nsteps, DTYPE step)
 98  {
 99      const unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;
100
101      if (tid < nsteps)
102        out_data[tid] = __drcp_rn(1.0 + __dmul_rn(threadIdx.x+0.5,step));
103  }
```

- Profiler ⇒ 93 ms with 512 threads/block (almost no impact).
- The fastest memory on a GPU chip is represented by its **registers**.
- GPU registers:
  - Limited amount : 255 per thread, 64K x 4Bytes max per block.
  - Maximal speed : no latency, no bank conflicts.
  - No possible indexing ⇒ no loops on array elements.
- Intra-warp register instructions:
  - `__shfl_sync,`
  - `__shfl_up_sync,`
  - `__shfl_down_sync,`
  - `__shfl_xor_sync`

CUDA : step #9

Warp level instructions

Example 1

```
96  __global__ void warpReduce_demo() {
97      int laneId = threadIdx.x & 0x1f;
98      // Seed starting value as inverse lane ID
99      int value = 31 - laneId;
100
101     int i=16;
102     //for (int i=warpSize/2; i>0; i/=2)
103         value += __shfl_down_sync(0xffffffff, value, i);
104
105     // "value" now contains the sum across all threads
106     printf("Thread %d final value = %d\n", threadIdx.x, value);
107 }
108
109 int main(int argc, char **argv)
110 {
111   warpReduce_demo<<< 1, 32 >>>();
112   cudaDeviceSynchronize();
113   return 0;
114 }
```

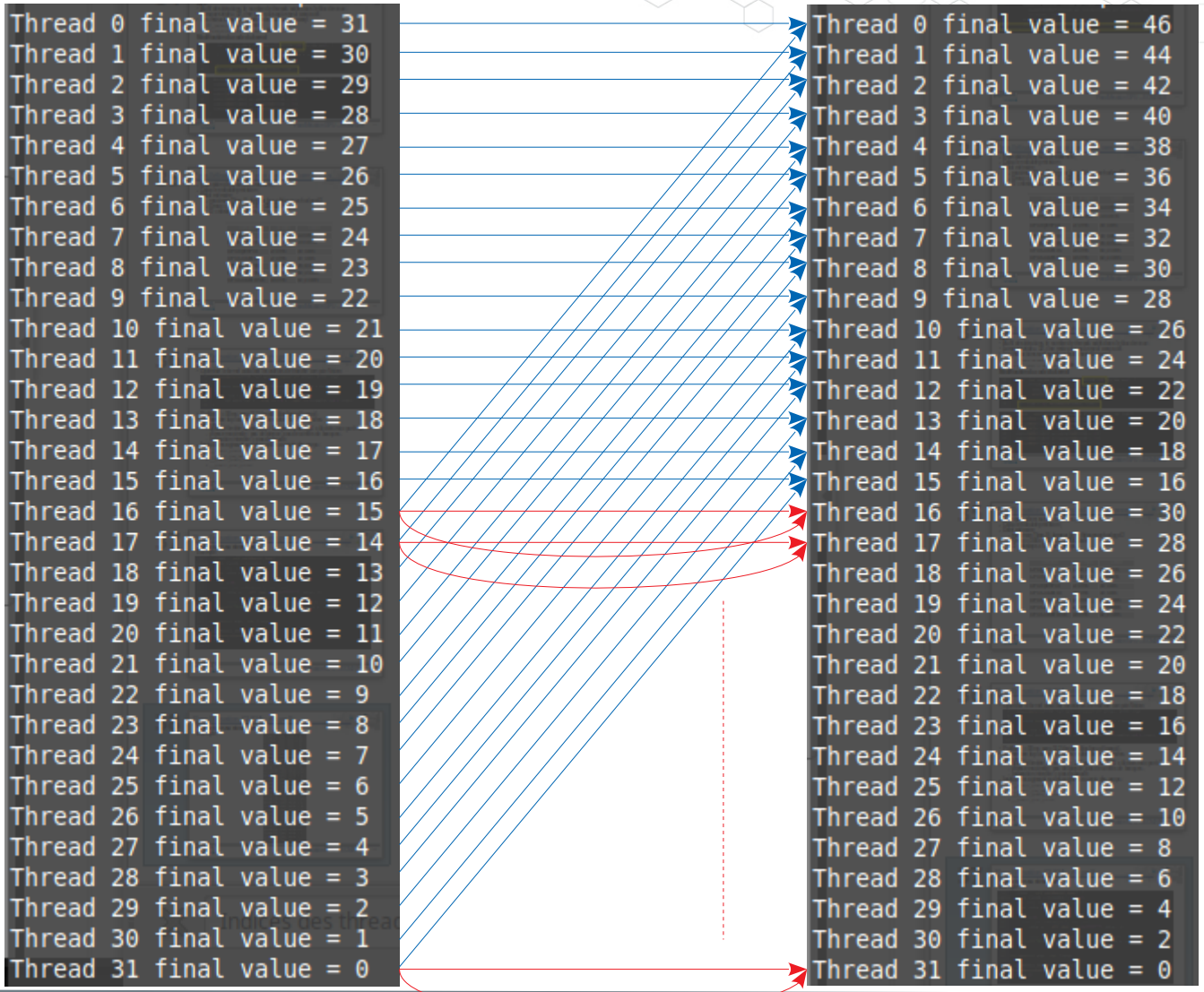
CUDA : step #9

Warp level instructions

Example1

`tid+offset<32`
⇒ returns
`tid+offset`

`tid+offset >31`
⇒ returns `tid`

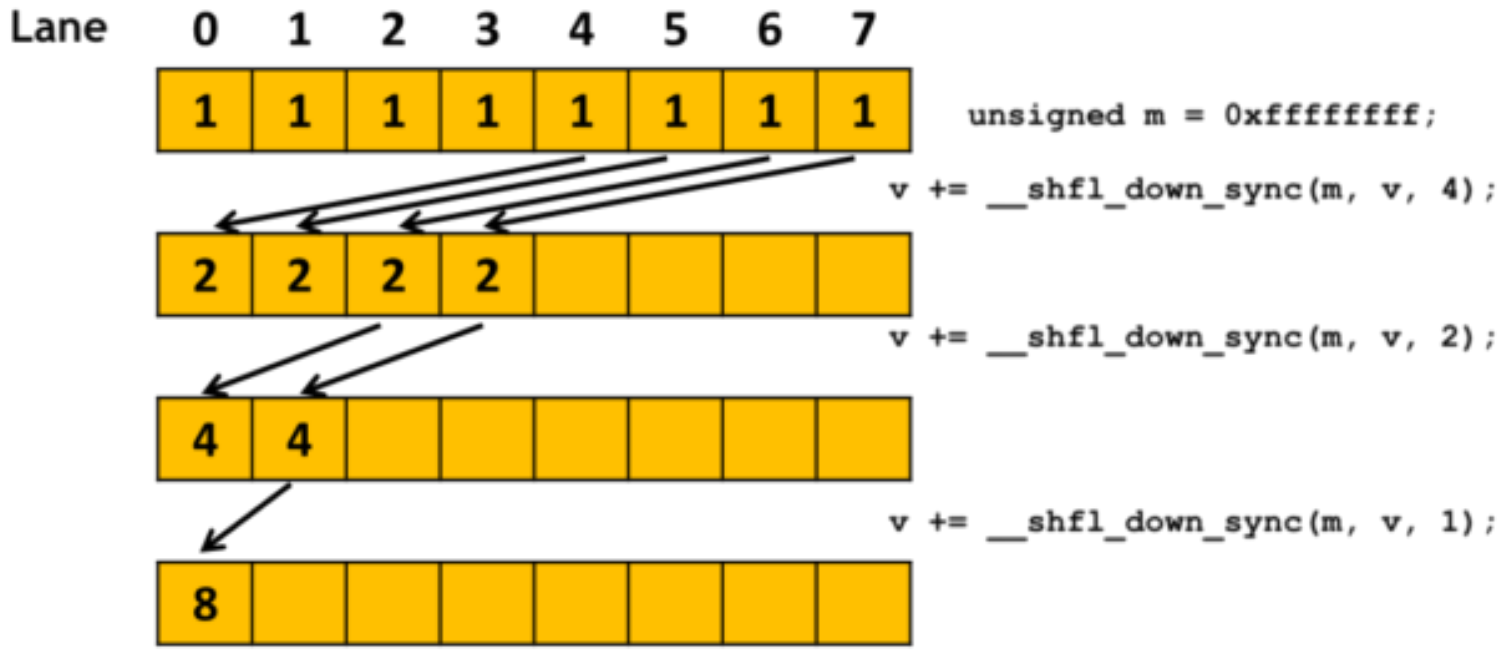


CUDA : step #9

Warp level instructions

Example 1 : sum over one warp

```
101     for (int i=warpSize/2; i>0; i/=2)
102         value += __shfl_down_sync(0xffffffff, value, i);
```



CUDA : step #9

Warp level instructions
Example 2 : sum over one block

__device__ function processing the sum over one warp

```
44  __inline__ __device__
45  DTYPE warpReduceSum(DTYPE val) {
46    for (int offset = warpSize/2; offset > 0; offset /= 2)
47     val += __shfl_down_sync(0xffffffff, val, offset);
48    return val;
49  }
```

Each thread calling this function would execute the `__shfl_down_sync()` instructions and then would return its own value stored in `val`.

CUDA : step #9

Warp level instructions

Example 2 : sum over one block

__device__ function processing the sum over one block

```
54  __inline__ __device__
55  DTYPE blockReduceSum(DTYPE val) {
56
57    static __shared__ DTYPE sdata[32];
58    int lane = threadIdx.x % warpSize;
59    int wid = threadIdx.x / warpSize;
60
61    val = threadIdx.x;
62    __syncthreads();
63
64    val = warpReduceSum(val);
65
66    if (lane==0)
67       sdata[wid]=val;
68    __syncthreads();
69
70    val = (threadIdx.x < blockDim.x / warpSize) ? sdata[lane] : 0;
71
72    if (wid==0)
73     val = warpReduceSum(val);
74
75    return val;
76  }
```

To store warp sums (max 32 in each block).

Warp id in the block =wid, Thread id in the warp =lane.

Just for demo, one puts values in val.

Sums inside warps.

Sum is held by thread #0 of the warp (lane #0).

Set-upvalues for second stage. If blockDim <1024, 0-padding.

Warp sum lane #0 = block sum

CUDA : step #9

Warp level instructions
Example 2 : sum over one block

Kernel processing the sum over one block

```
110  __global__ void blockReduce_demo() {
111    int sum = 0;
112    sum = blockReduceSum(sum);
113    if (threadIdx.x==0)
114     printf("Thread %d final value = %d\n", threadIdx.x, sum);
115  }
116
117  int main(int argc, char **argv)
118  {
119    blockReduce_demo<<< 1, 1024 >>>();
120    cudaDeviceSynchronize();
121    return 0;
122  }
```

Kernel calling the function.

kernel call with 1 block of 1024 threads.

Résultat

```
perrot@cluster2:~/samples-9/0_Simple/ecmDemo_shuffle$ ./template
Thread 0 final value = 523776
```

Correct result.

CUDA : step #9

Warp level instructions

Processing the sum over a vector of values

```
87  __global__ void deviceReduceKernel(DTYPE *in, DTYPE* out, int N) {
88    DTYPE sum = 0.0;
89
90    for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x *
      •    gridDim.x) {
91       sum += in[i];
92    }
93    sum = blockReduceSum(sum);
94    if (threadIdx.x==0)
95     out[blockIdx.x]=sum;
96  }
```

nb of values

If nb blocks > 1024, pre-summation

Thread #0 in each block holds the sum of its own block

La somme globale est obtenue en deux temps

```
82    int threads = 512;
83    int blocks = min((N + threads - 1) / threads, 1024);
84    deviceReduceKernel<<<blocks, threads>>>(in, out, N);
85    deviceReduceKernel<<<1, 1024>>>(out, out, blocks);
```

Max 1024 blocks

Block sums

Sum of sums

CUDA : step #9

Warp level instructions
Estimation of π

```
166     computeNstore<<< grid, threads, 0 >>> (d_vector, nbsteps, 1.0/nbsteps);
167
168     nblocks = min(nblocks, 1024);
169     checkCudaErrors(cudaMalloc((void **) &d_sums, sizeof(DTYPE)*nblocks));
170     deviceReduceKernel<<<nblocks,threads_per_block>>>(d_vector,
    •       d_sums,nbsteps);
171     deviceReduceKernel<<<1, 1024>>>(d_sums, d_sums, nblocks);
```

```
./template Starting...

GPU Device 0: "GeForce GTX TITAN X" with compute capability 5.2

nbsteps : 1073741824 = 10^9 - Memory size : 8589934592
Memory total : 12806062080 - Memory free : 12566986752
Processing time: 118.502998 (ms)
Pi     GPU : 3.1415926536
Pi ref CPU : 3.1415926536
TEST PASSED.... (err 0.000189)x10^-9
```

Performance summary

- Estimation of π over 10^9 slices;
- Double precision real computation;
- Memory throughput (D)
  - maximum Titan-X : bus width x clock frequency/2
  - Dmax = 384 x 3505/2 8 = **336 GB/s**
  - D = nbsteps x 8 Bytes / duration

Version	Duration	Throughput
CPU gcc	1440 ms	5.5 Go/s
CPU openPGI	2358 ms	3.4 Go/s
GPU openPGI	270 ms	30 Go/s
GPU CUDA v3	270 ms	30 Go/s
GPU CUDA v4	220 ms	36.4 Go/s
GPU CUDA v5	200 ms	40,0 Go/s
GPU CUDA v7/v8	190 ms	42,1 Go/s
GPU CUDA v9	**118 ms**	**67,8 Go/s**

Connecting to IUTBM cluster nodes

- `ssh login@cluster2` (or `cluster1` or `cluster3`)

If needed, first log in on slayer with

- `ssh login@slayer.iut-bm.univ-fcomte.fr`

For more convenience, edit your ~/.ssh/config and add some useful things:

```
Host *
     ForwardAgent yes
     ForwardX11 no
     ForwardX11Trusted yes
     Port 22
     Protocol 2
     ServerAliveInterval 60
     ServerAliveCountMax 30

Host cluster2
    Hostname cluster2
    User login
    ProxyCommand ssh -W %h:%p login@slayer
```