GPGPU Programming Fundamentals — 3

Salvatore Filippone, PhD

DICII
salvatore.filippone@uniroma2.it

Performance

In this section you will hear about...

- What are the most important factors affecting GPU codes performance
- How to handle CPU-GPU memory transfers with page-locked memory, Mapped memory, Unified memory;
- Global memory coalescing and optimising memory layout
- Shared memory bank conflicts
- Warp divergence

Performance Guidelines

Easy to learn, difficult to master

- It is relatively easy to write working GPU code, but. . .
- It is difficult to get it right!

Strategies for Optimisation (from CUDA Programming Guide)

- Maximise parallel execution to achieve maximum utilisation
- Optimise memory usage to achieve maximum memory throughput
- Optimise instruction usage to achieve maximum instruction throughput

Page-locked Host Memory

- Page-locked memory enables faster CPU-GPU transfers
- Pageable memory allocated by `malloc()` can be locked with `cudaHostRegister()`
- *Using page-locked memory reduces the amount of physical memory available to the operating system!*

```
float* h_vec, d_vec;
cudaHostAlloc((void**) &h_vec, N * sizeof(float),
                        cudaHostAllocDefault);
cudaMalloc((void**) &d_vec, N * sizeof(float));
// Faster copy from page-locked host memory
cudaMemcpy(d_vec, h_vec, N * sizeof(float),
                   cudaMemcpyHostToDevice);
cudaFreeHost(h_vec);
cudaFree(d_vec);
```

Mapped Memory

- Simplifies programming tasks, but hides where/when the transfers happen;
- The memory has dual pointers, from host and from device;
- Need to synchronize to access data;

```
__global__ void foo(float *vec, int n) {
  if (threadIdx.x < n) vec[threadIdx.x] = 1.0 + threadIdx.x;
}
....
  float* h_vec, d_vec;
  cudaHostAlloc((void**) &h_vec, N * sizeof(float),
                                cudaHostAllocMapped);
  cudaHostGetDevicePointer(&d_vec, h_vec,0);
  // Fill in d_vec with a kernel
  foo<<<128,128,0>>>(d_vec,N);
  cudaDeviceSynchronize();
  for (int i=0; i<N; i++) printf("v[%d]=%f \n",i,h_vec[i]);
  cudaFreeHost(h_vec); d_vec=NULL;
```

Unified Memory (from CUDA 6)

- Simplifies programming tasks, but hides where/when the transfers happen;
- The memory has a single pointer, from both host and device;
- Need to synchronize to access data;

```
__global__ void AplusB(int *ret, int a, int b) {
  ret[threadIdx.x] = a + b + threadIdx.x;
}
int main() {
  int *ret;
  cudaMallocManaged(&ret, 1000 * sizeof(int));
  AplusB<<< 1, 1000 >>>(ret, 10, 100);
  cudaDeviceSynchronize();
  for(int i=0; i<1000; i++)
    printf("%d: A+B = %d\n", i, ret[i]);
  cudaFree(ret);
  return 0;
}
```

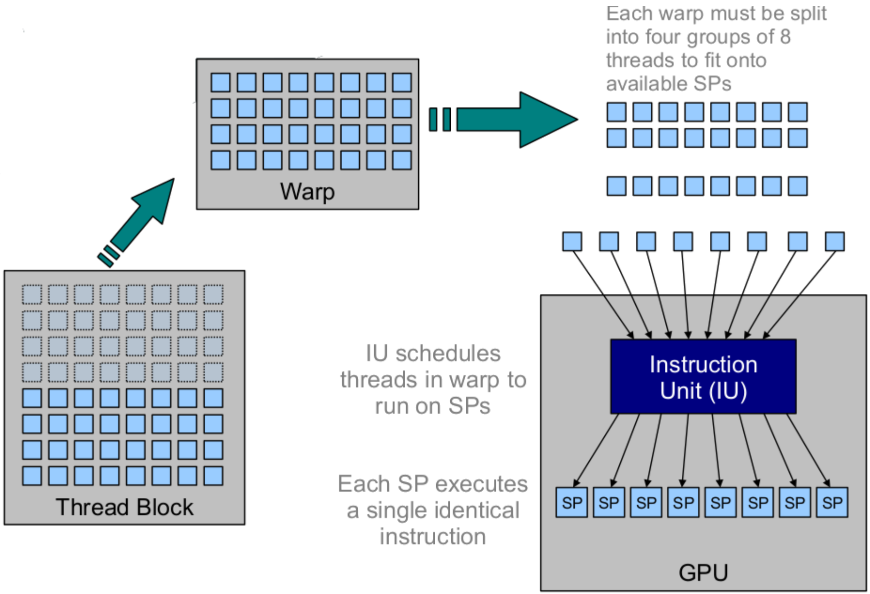
Maximise Utilisation

- Design your algorithms to use as many threads as possible
- Use scarce resources carefully (shared memory, registers)
- Overlap computation and communication:
  - CUDA streams
  - Asynchronous CUDA memory API (Async suffix)
  - Page-locked host memory
- Pay attention to pipeline utilisation and memory accesses!!

Warps

- A warp is a basic unit of execution on the GPU
- Each warp contains 32 threads
- Threads assigned to a warp according to contiguous thread IDs
- Represents the lowest level parallel granularity
- The threads in a warp execute a single instruction at a time: SIMT - Single Instruction, Multiple Threads
- Once all threads in the warp have completed the instruction, the next instruction is fetched

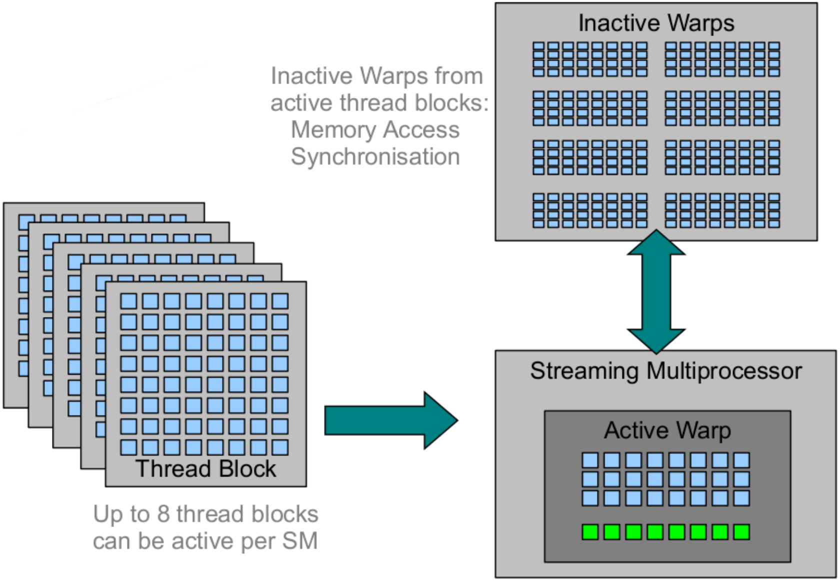
Warps


Each warp must be split into four groups of 8 threads to fit onto available SPs
Warp
Thread Block
IU schedules threads in warp to run on SPs
Each SP executes a single identical instruction
Instruction Unit (IU)
SP
GPU

Scheduling

- Four clock cycles needed to dispatch instruction to all the threads in a warp
- Each SM has 8 SPs
- To process an entire warp, the warp must be split into 4 subsets and the instruction sent to each and executed on each
- Hence, 32 threads / 8 SPs = 4 instruction fetches (1 cycle each)
- Warps can be “paged out” if threads are waiting on a memory request
- This allows other eligible warps to be scheduled for execution by the Instruction Unit $\Rightarrow$ improved performance!
- Local register values are preserved when threads are inactive

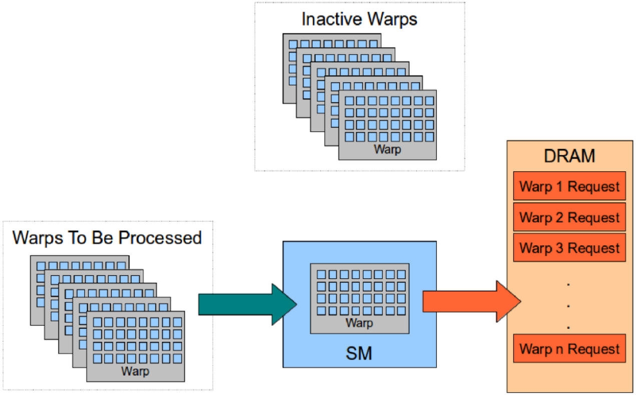
Scheduling


Inactive Warps
Inactive Warps from active thread blocks: Memory Access Synchronisation
Thread Block
Up to 8 thread blocks can be active per SM
Streaming Multiprocessor
Active Warp

Scheduling

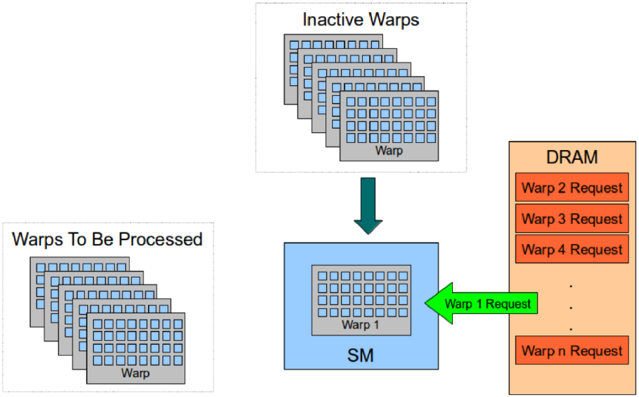
- To amortise latency due to DRAM access, many warps need to be active on an SM
- Each access requires 4 cycles to process the request
- Each request can take at least 200 cycles to complete
- To attain optimal performance, ensure the hardware is not lying dormant

Processing Global Memory Requests (1/2)



- Warps on SM which are active are dispatched for processing
- If any thread requests data from the global memory, the whole warp becomes inactive

Processing Global Memory Requests (2/2)



- When global memory request is fulfilled, warp becomes active
- *If all warps are inactive (waiting for data), the SM is idle!*

Global Memory Coalescing

- Global memory is accessed with a very high latency
- Solution: access global memory in larger transactions

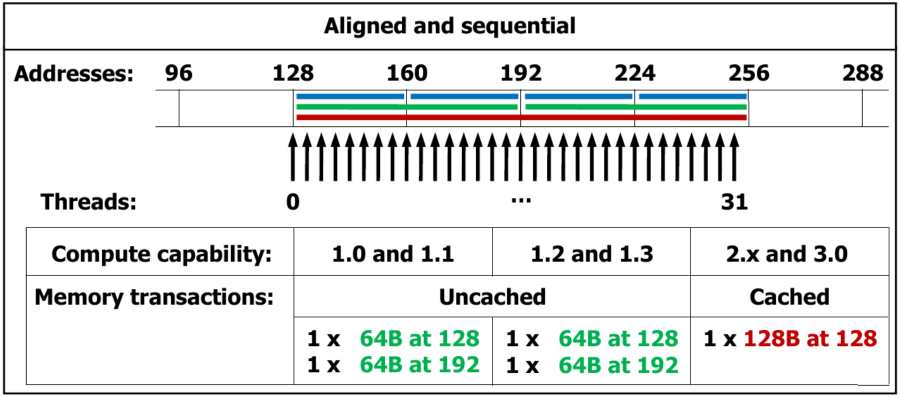
Global Memory Access

- Threads within a warp can access data in the global memory in a reduced number of transactions
- Memory cells accessed by threads in warp are grouped in transactions of 32-, 64- or 128-bytes
- Data in each transaction must be properly aligned!

To maximise global memory throughput:

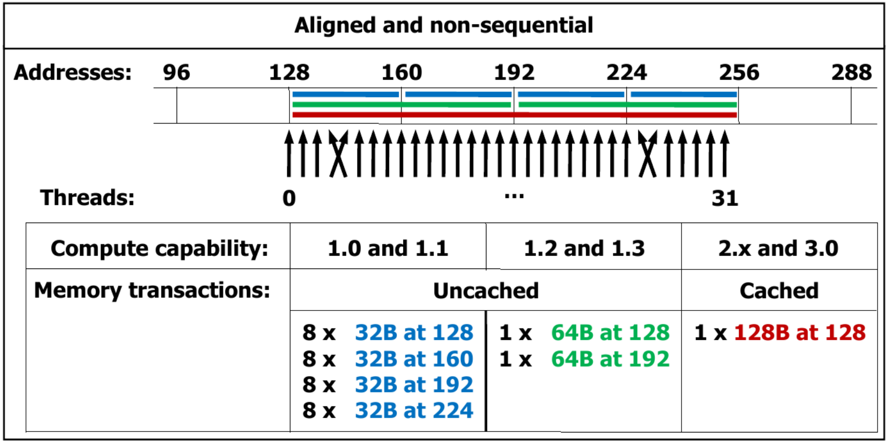
- follow optimal access patterns,
- pad data to meet the alignment requirements (e.g. 2D arrays),
- use data types of 1, 2, 4, 8 or 16 bytes (careful with struct!)

Global Memory Access Patterns (1/3)



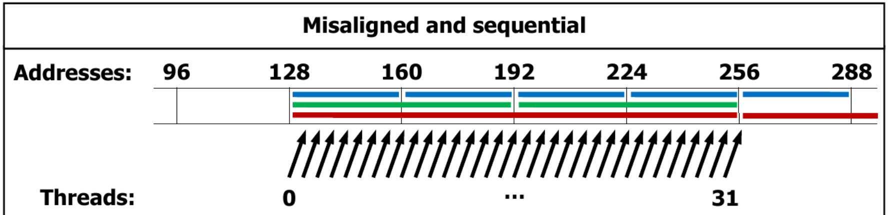
Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Global Memory Access Patterns (2/3)



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Global Memory Access Patterns (3/3)



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.0
Memory transactions:	Uncached		Cached
	7 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224 1 x 32B at 256	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

Choosing the Right Data Layout

Array of Structures

Vector	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4	5	5	5	5	6	6	6	6	7	7	7	7	8	8	8	8	9	9	9	9
Element	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4

- Vectors are processed one-by-one
- Cache-friendly on CPU-like architectures
- Significantly slower on GPUs (limited benefits from coalescing)

Structure of Arrays

Vector	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9
Element	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3	3	4	4	4	4	4	4	4	4	4

- All vectors are processed at the same time
- Benefits from full global memory coalescing on the GPU
- Likely to be slower on the CPU due to low cache hit ratio

A Detour: Multi-Dimensional Arrays

What is a multi-dimensional array?

An array is a collection of objects all of the same type, identified by *a set of one or more* integer indices.

Trouble is:

You can view a 2D array as a set of 1D array (which ones?)

but

That's defeating its purpose (to some extent)

Compiler view

An array is a collection of objects such that you can identify the address in memory of any entry given only the set of indices and the size(s)

A Detour: Multi-Dimensional Arrays

A 1-D example

Consider the following code:

```
double v[10];
i=4;
x=v[i];
```

To access `v[i]` the compiler needs to know the following:

- The starting address in memory of the vector `v`;
- The size of each element of the vector (for a `double`: 8 bytes);
- The starting index `xb` of the first element (in C: 0);

Then it can compute

```
   addr = start(v) + size * ( i - xb)
        = start(v)+ 8*(4-0)
```

A Detour: Multi-Dimensional Arrays

A 2-D example

Consider the following code:

```
double a[10][20];
i=4; j=3;
x=a[i][j];
```

How is the 2D array laid out in memory? In C, *storage by rows.*

To access `a[i][j]` the compiler needs to know the following:

- The starting address in memory of the array `a`;
- The size of each element of the vector (for a `double`: 8 bytes);
- The starting index `xb` in each dimension (in C: 0);
- How many entries `NC` there are in each row (in this case: 20)

Then it can compute

```
    addr = start(a) + size * ( (i - xb)*NC + (j-xb) )
```

A Detour: Multi-Dimensional Arrays

Note that there is another way to store arrays: *by columns*. And, what if the first index is 1 instead of 0?
This is what Matlab and Fortran (and Julia) do, in which case the formula requires the number of *rows* instead of columns, and you have

```
addr = start(a) + size * ( (j - xb)*NR + (i -xb))
     = start(a) + 8*( (3-1)*10+(4-1))
```

- Matlab *requires* the first address to be 1;
- Fortran allows the programmer to choose an arbitrary starting index `xb`, with 1 being the default.

A Detour: Multi-Dimensional Arrays

What happens if the number of rows/columns is not known at compile time?

For instance, you might have a function accepting the variable a: you might want to pass to the function the number of rows and columns, to interpret the variable properly. If you can do this, then your language supports 2D arrays in the most general sense

A Detour: Multi-Dimensional Arrays

What happens if the number of rows/columns is not known at compile time?

For instance, you might have a function accepting the variable a: you might want to pass to the function the number of rows and columns, to interpret the variable properly. If you can do this, then your language supports 2D arrays in the most general sense

Unfortunately, this is not what happens in C.

The default action in the C language is to interpret an expression like

```
x = mat[i][j];
```

in two completely different ways depending on whether the size of `mat` is known at compile time or not.

A Detour: Multi-Dimensional Arrays

When the size is only known at runtime. . .

. . . the compiler will interpret

```
x = mat[i][j];
```

as meaning that

- mat is a (pointer to an) array of pointers; so by mat[i] you are selecting the pointer at entry [i];
- Each pointer identifies a separate row array; entries in each row are addressed by [j];
- Each row is allocated independently and there is no obvious relationship between the locations in memory of two different rows.

A Detour: Multi-Dimensional Arrays

When the size is only known at runtime…

…This is different from Matlab and Fortran, where the size is *embedded* in the array object

```
real :: A(:,:)
x = A(i,j)
```

meaning that

- `A` contains details about the size in each dimension;
- The compiler always implements the formula we have seen previously at runtime;

A Detour: Multi-Dimensional Arrays

Do programming languages support multidimensional arrays?

A Detour: Multi-Dimensional Arrays

Do programming languages support multidimensional arrays?

Matlab	Yes (by columns)
Fortran	Yes (by columns)
Julia	Yes (by columns)

A Detour: Multi-Dimensional Arrays

Do programming languages support multidimensional arrays?

Matlab	Yes (by columns)
Fortran	Yes (by columns)
Julia	Yes (by columns)
Java	No (by rows)

A Detour: Multi-Dimensional Arrays

Do programming languages support multidimensional arrays?

Matlab	Yes (by columns)
Fortran	Yes (by columns)
Julia	Yes (by columns)
Java	No (by rows)
C	It's complicated (and it's by rows)
C++	It's complicated (and it's by rows)

In C/C++/Java you can have

`void compute(double a[][])`

but then a is interpreted as

A 1-D array of pointers (to 1-D arrays)

and of course those pointers (even if the rows all have the same length) can point to anything, i.e. to know where `mat[2][3]` is in memory, you need to examine the *pointer* contained in the location of `mat[2]`, then access its element at index 3. Knowing the start address of `mat` and its size(s) is not enough

A Detour: Multi-Dimensional Arrays

Possible solutions to have a 2-D data layout in memory

- If you are using Matlab, Fortran or Julia, you're all set;

A Detour: Multi-Dimensional Arrays

Possible solutions to have a 2-D data layout in memory

- If you are using Matlab, Fortran or Julia, you're all set;
- If you are using Java, you are out of luck, there's no way of forcing what you want (but see next item);

A Detour: Multi-Dimensional Arrays

Possible solutions to have a 2-D data layout in memory

- If you are using Matlab, Fortran or Julia, you're all set;
- If you are using Java, you are out of luck, there's no way of forcing what you want (but see next item);
- If you are using C or C++, you can declare the array *as if* it was 1-D, then implement the index calculation "by hand"

  ```
  void compute(int nr, int nc, double mat[])
  x = mat[my2Dindex(i,j,nr,nc)]

  int my2Dindex(i,j,nr,nc) { return(i*nc+j);}
  ```

  In this case you *really* want the compiler to do inlining. Especially if `my2Dindex` is declared as a class member function (or method) (See the STL).

But wait, there's more

A Detour: Multi-Dimensional Arrays

Possible solutions to have a 2-D data layout in memory

If you are using C, *and if* you can force the compiler to use the *C99* standard semantics, *then* you can write something like

```
void compute(int nr, int nc, double mat[nr][nc])
```

and the compiler will interpret the matrix as a 2D array correctly.
Note that you *must* put `nr,nc` *before* `mat` in the function argument list.

If you want to have fun

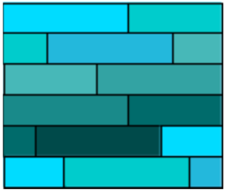
use a search engine and look for

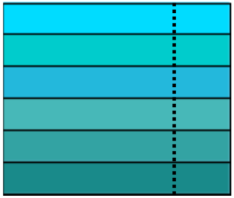
How do I declare a 2d array in C++ using new?

Two-dimensional Arrays (1/2)

Row alignment

- Data allocated with cudaMalloc is always properly aligned
- How about 2D array stored in linear memory (e.g. structure of arrays in the previous example)?


Misaligned rows, limited coalescing


Aligned rows, full coalescing

- If allocated with cudaMalloc, only the first row is guaranteed to be aligned
- Other rows must be padded for maximum performance

Two-dimensional Arrays (2/2)

cudaMallocPitch()

- Allocates 2D array in linear memory
- Each row is aligned to meet coalescing requirements
- pitch must be passed as parameter to the kernel

```
// Allocates 2D array with M rows and N columns on GPU
int* d_vec = 0; size_t pitch = 0;
cudaMallocPitch((void**) &d_vec, &pitch,
                             N * sizeof(int), M);
// Returned pitch is in bytes!
pitch /= sizeof(int); // Easier for pointer arithmetic
// d_vec[2*pitch + 1] is the 2nd element in the 3rd row
int* h_vec = new int[N * M];
cudaMemcpy2D(d_vec, pitch * sizeof(int), // destination
                  h_vec, N * sizeof(int), // source
                  N * sizeof(int), M, // width and height
                  cudaMemcpyHostToDevice);
```

Shared Memory Bank Conflicts

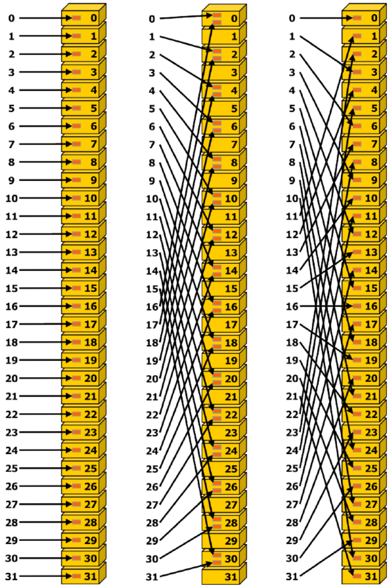
Shared Memory Implementation

- Organised in 32 memory banks
- Consecutive 32-bit words of shared memory are mapped to consecutive memory banks
- Threads within a warp can access data in shared memory simultaneously,
- *provided there are no bank conflicts*

Avoid bank conflicts to maximise shared memory throughput

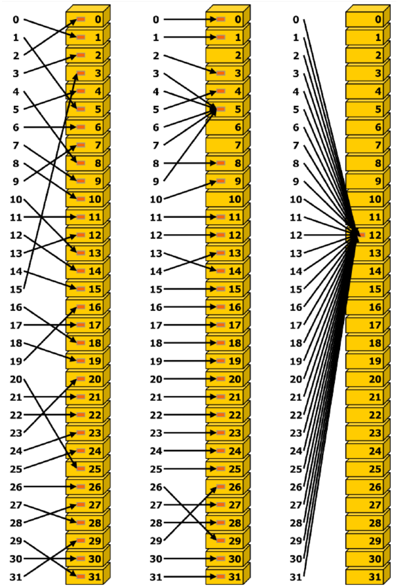
- Ideally, threads should access consecutive memory cells
- Stride of length $k \perp 32$ is also okay
- It is okay if all threads in warp read from the same address
- Use data types of 4 or 8 bytes (smaller require padding!)

Shared Memory Access Patterns (Strided)



- Left: Linear addressing with a stride of one 32-bit word (no bank conflict)
- Middle: Linear addressing with a stride of two 32-bit words (2-way bank conflicts)
- Right: Linear addressing with a stride of three 32-bit words (no bank conflict)

Shared Memory Access Patterns (Irregular)



- Left: Conflict-free access via random permutation
- Middle: Conflict-free access since threads 3, 4, 6, 7 and 9 access the same word within bank 5
- Right: Conflict-free broadcast access (all threads access the same word)

Divergent Execution Within Warps

`if, switch, do, for, while`

- Control instructions can significantly limit the throughput
- Entire warps are executed rather than particular threads
- All threads in a warp perform the same operation concurrently

```
if (threadIdx.x % 2 == 0) {
  ... // Do something (threads 0, 2, 4, ...)
} else {
  ... // Do something else (threads 1, 3, 5, ...)
```

Instruction throughput is reduced by a factor of two

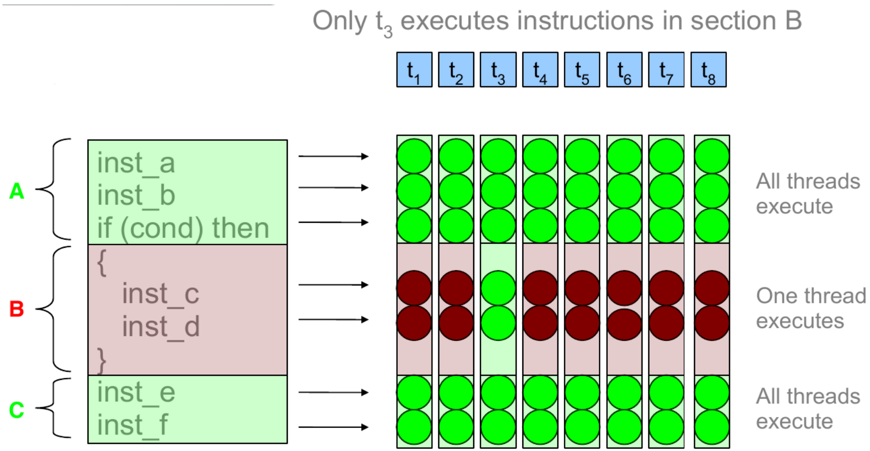
```
if (threadIdx.x < 128) { // Assuming block = 256 threads
  ... // Do something (first four warps)
} else {
  ... // Do something else (second four warps)
```

Often divergent warps can be avoided

Instruction Divergence

- Execution path divergence within a warp is penalised
- When the path of execution diverges between threads in a warp
  - Execution is serialised
  - In practice all threads perform the same instructions, but threads that do not need to follow the branch ignore their results
- Divergence between warps not so important

Thread Divergence



Other Instruction Throughput Optimisations

Minimise the use of low-throughput arithmetic instructions

- If precision is not crucial, intrinsic instructions can be used instead (e.g. faster but less accurate `__sinf`, `__cosf`)
- Single precision can be used instead of double precision

Remove excess `__syncthreads()` instructions

- Redesign your algorithm to minimise the number of synchronisation points
- Threads in a warp are executed in a lock-step, i.e. they are always synchronised

Case study: Implementation of GEMM

GEMM: General Matrix Multiply

$$C \leftarrow \alpha * A^X * B^X + \beta * C$$

where X can be `N` no transpose or `T` transpose. Essential kernel of the BLAS3, LAPACK and all dense linear algebra operations.
We have already seen some of the issues in conventional processors, and specifically:

- Unrolling the inner loops;
- Data reuse;
- Cache reuse: blocking.

Implementation of GEMM

Large tiling technique: computation proceeds by tiles

- Tile of matrix C of size 64×16 per thread block;

Implementation of GEMM

Large tiling technique: computation proceeds by tiles

- Tile of matrix C of size 64×16 per thread block;
- Each thread computes a column;

Implementation of GEMM

Large tiling technique: computation proceeds by tiles

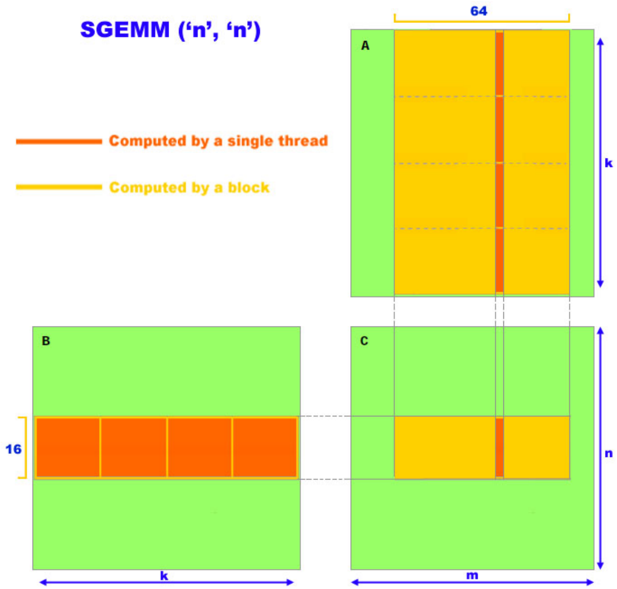
- Tile of matrix C of size 64×16 per thread block;
- Each thread computes a column;
- Shared memory used matrix B;

Implementation of GEMM

Large tiling technique: computation proceeds by tiles

- Tile of matrix C of size 64×16 per thread block;
- Each thread computes a column;
- Shared memory used matrix B;
- Matrix A from global memory.

Implementation of GEMM



Implementation of GEMM

Only 64 threads per block (theor. max: 512)

- Occupancy reduction, but
- Memory transaction hiding!
- Need for explicit pointer arithmetic (offset computation, reducing synchronizations)
- Leading dimension of buffer (bank conflict avoidance)
- Prefetching of A

Implementation of GEMM

Input size variance: need to employ padding (copy-in/copy-out) to smooth performance (multiples of tile size).
Transpositions: for TT variant

$$C \leftarrow A^T \cdot B^T$$

Implementation of GEMM

Input size variance: need to employ padding (copy-in/copy-out) to smooth performance (multiples of tile size).
Transpositions: for TT variant

$$C \leftarrow A^T \cdot B^T$$

exploit elementary identity

$$A^T \cdot B^T = (B \cdot A)^T$$

by transposing the output buffer during copy-out!
Large improvement over previous CUBLAS implementation, Up to 40 % peak performance on GTX 9800
Code is now included since CUBLAS 2.1

Applications: All-Pairs Shortest-Path

Given a graph

$$G = (V, E)$$

find the length of the shortest path between each pair of vertices. Basic formulation of "brute force" Floyd-Warshall algorithm:

> *very similar to a generalized matrix multiplication over* $(R, \min, +)$ *instead of* $(R, +, \times)$

```
$Ms \leftarrow Ma$
**for** $k = 1, \ldots$ **do**
    **for** $D(i,j) \in Ms$ **do**
        $D(i,j) \leftarrow \min(D(i,j), D(i,k) + D(k,j))$
    **end for**
**end for**
```

Bounded by outer loop;

Applications: All-Pairs Shortest-Path

Revised MM-based algorithm:

```
$Ms \leftarrow Ma$
**for** $p = \log(\min(|V|, |E|))$ **do**
    **for** $D(i,j) \in Ms$ (in parallel) **do**
        **for** $k = 1, \ldots$ **do**
            $D(i,j) \leftarrow \min\left(D(i,j), D(i,k) + D(k,j)\right)$
        **end for**
    **end for**
**end for**
```

If the graph is sufficiently dense the MM-based algorithm is better.

Applications: All-Pairs Shortest-Path

Results on random graphs on GTX 285 (msec):

	Floyd-Warshall			*Matrix-Multiply based*		
nodes	#edges/#nodes ratio			#edges/#nodes ratio		
	1/10	1/1	10/1	1/10	1/1	10/1
64	1.28	1.3	1.4	**0.49**	**0.72**	**0.74**
128	2.49	2.53	2.81	**0.75**	**1.13**	**1.27**
256	5.5	5.6	8	**1.45**	**2.12**	**3**
512	14	14	40	**5.73**	**8.45**	**15**
1024	41	**42.6**	275	**30**	43	**102**
2048	**136**	**157**	1525	191	262	**816**
4096	**508**	**659**	8514	1480	1896	**6877**
8192	**2067**	**3012**	**51949**	13901	19661	58736

Generated with GTgraph
`http://www.cse.psu.edu/~kxm85/software/GTgraph/`

Summary

- Design your algorithms to use as many threads as possible
- Use scarce resources carefully (shared memory, registers, page-locked memory)
- Organise your data to maximise global memory throughput (take advantage of coalescing mechanism)
- Avoid shared memory bank conflicts to maximise throughput
- Design control instructions to avoid divergent warps