Performance optimisation

Outline

- Occupancy
- Branching
- Memory optimisations
- Miscellaneous

Occupancy

Filling the GPU

- GPU devices are high latency, high bandwidth devices
- You need to have enough tasks to fill all memory/compute units on the devices
- You need to have enough tasks to exploit the high bandwidth and hide latency

AMD MI210

- 104 CU (Compute units)
- Each CU has 4 SIMD processors with 16 lanes
- Up to 8 wavefronts can be executed by a SIMD processor
- A wavefront is composed of 64 threads
- All the threads in a wavefront execute the same instruction

Occupancy

- Maximum number of threads on the GPU= 104 (CU) x 4 (SIMD processors) x 8 (Instructions per SIMD) x (64 threads per wavefront) = 212992
- Over 200k threads can be scheduled on the GPU
- The ratio between the number of threads executing on a GPU and the theoretical maximum number of threads scheduled on the GPU is called occupancy

Occupancy

- Some of the wavefronts on the device will be stalled, i.e. waiting for data to become available from memory, floating point operation units being busy …
- But the GPU can quickly switch between wavefronts that are stalled and wavefronts that are ready to execute, hiding latency
- Need occupancy large enough to hide latency
- Higher occupancy does not always equal higher performance.

Occupancy limiters

Theoretical maximum occupancy can be limited by

- Team size and number of teams
- Shared memory
- Registers

Occupancy: Teams size and number

- Each team is scheduled to a single CU
- Need at least as many teams as Computing Units to fill the GPU
- Up to 32 wavefronts (2048 threads) can be scheduled per CU
- If the team size is 17 wavefronts (1088 threads), only one team can fit on a CU at the same time. Max. possible occupancy is 1088/2048 = 53%

Occupancy: Teams size and number

- Use the `num_teams` clause to specify to place an upper bound on the number of teams
- Use `thread_lim` to place an upper bound on the number of threads per team

Occupancy: Teams size and number

- Make sure there is enough parallelism to fill the GPU
- Merge small loops into bigger loops
- In multidimensional loops use the collapse clause to spread inner loop operations across different threads

Occupancy: Register pressure

- Each team needs a certain number of registers
- Finite number of registers per computing units and SIMD, hardware dependent
- Can limit the number of teams that are executed concurrently on Computing Unit
- If more registers are needed than are available, the compiler can decide to spill the registers onto shared or global memory.
- Register memory is orders of magnitude faster than shared or global memory

Occupancy: Register pressure

- MI200 has **Vector Registers** (**VGPRs**) and **Scalar Registers** (**SGPRs**)

- A vector register holds 64 items, 4B wide
- Up to 512 vector registers per SIMD processor

- A scalar register holds the same value for all threads in a wavefront.
- Up to 800 per SIMD
- In practice, it is more common to run out of vector registers

Occupancy: Register pressure

- If a wavefront needs 128 vector registers, at most 512/128 = 4 wavefronts can reside on a SIMD, and so only 16 out of a possible 32 wavefronts can run on a CU.
- Occupancy is 50%

- N.B. a single wavefront cannot use more than 256 registers

Occupancy: Register pressure

- Split big loop bodies into multiple loops with smaller bodies. Smaller loop bodies might need smaller number of registers
- Smaller loops might decrease the amount of parallelism, so might not improve performance.
- May require additional temporary arrays to transmit values between loops
  - increases memory usage, can cause more pressure on cache space and/or memory bandwidth
- Use the `thread_limit` clause. If the compiler knows at compile time the maximum number of threads per team it can allocate registers more efficiently.

Occupancy: Shared Memory

- OpenMP tends to not to utilise shared memory very much.
- Shared memory is shared by all threads in a team.
- On MI200, there is 64KB of shared memory per CU.
- If a team requires 32KB of shared memory , at most two teams can fit on a single Computing Unit

Branching

Branching

- Threads in a team are grouped in wavefronts / warps
- The number of threads in a wavefront is architecture dependent
- For AMD GPUs a wavefront is usually 64 threads
  - For NVIDIA GPUs a warp is usually 32 threads
- All threads in a wavefront always execute the same instruction

Branching

- What if different threads in a wavefront execute different instructions ?

```
#pragma omp target teams
distribute parallel for
for ( int i=0; i<n; i++){
    if (i%2 == 0) {
        x[i] = a
    }
    else {
        x[i] = b;
    }
}
```

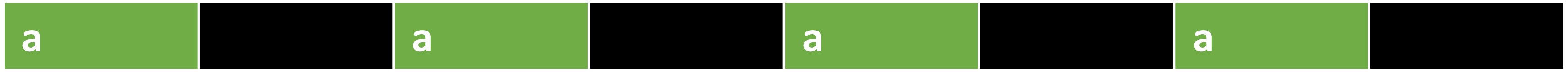
C/C++

```
!$omp target teams
        distribute parallel do
do i=1,n
        if (modulo(i,2) == 0) then
                x(i)= a
        else
                x(i)=b
        endif
end do
!$omp end target teams
```

F

Branching

- Set x to a on all threads, but mask odd threads

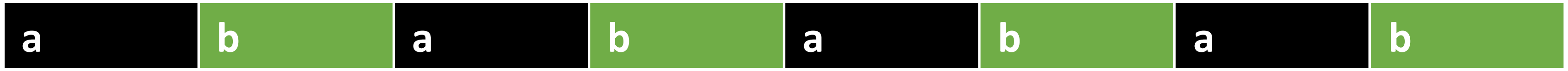


a		a		a		a	

```
    if (i%2 == 0)
    {
        x[i] = a
    }
```

C/C++

- Set x to b on all threads, but mask even threads



a	b	a	b	a	b	a	b

```
else
    {
        x[i] = b;
    }
```

C/C++

- Need to run two instructions per thread instead of one instruction per thread

Branching

- For each instruction, run the instruction on all threads, but mask the result of threads that do not need to run the instruction
- Each branch executed by all threads: leads to loss of parallelism
- Can strongly limit the performance of some scientific codes.
- An example of codes strongly affected are Monte Carlo codes.
- Monte Carlo codes often execute a random event per thread. All threads then execute different instructions at the same time, leading to serialization

Branching

- Avoid branching in GPU codes
- Might require re-thinking the parallel algorithm

Memory optimization

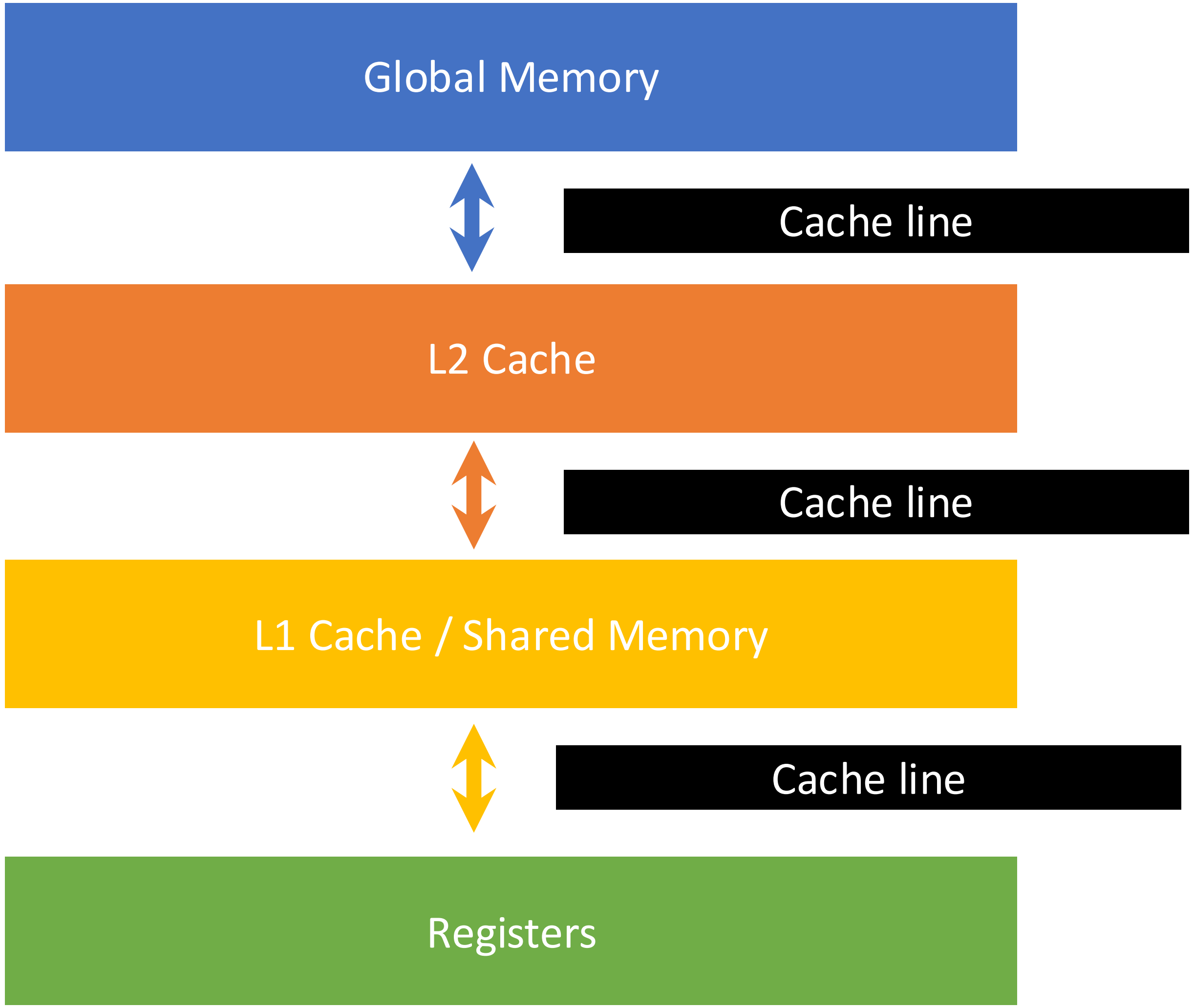
Memory usage

- Hierarchical memory structure
- Different types of Memory: L2 cache , L1 cache/Shared memory, Global Memory etc...
- Each layer is faster than the one above

Memory usage

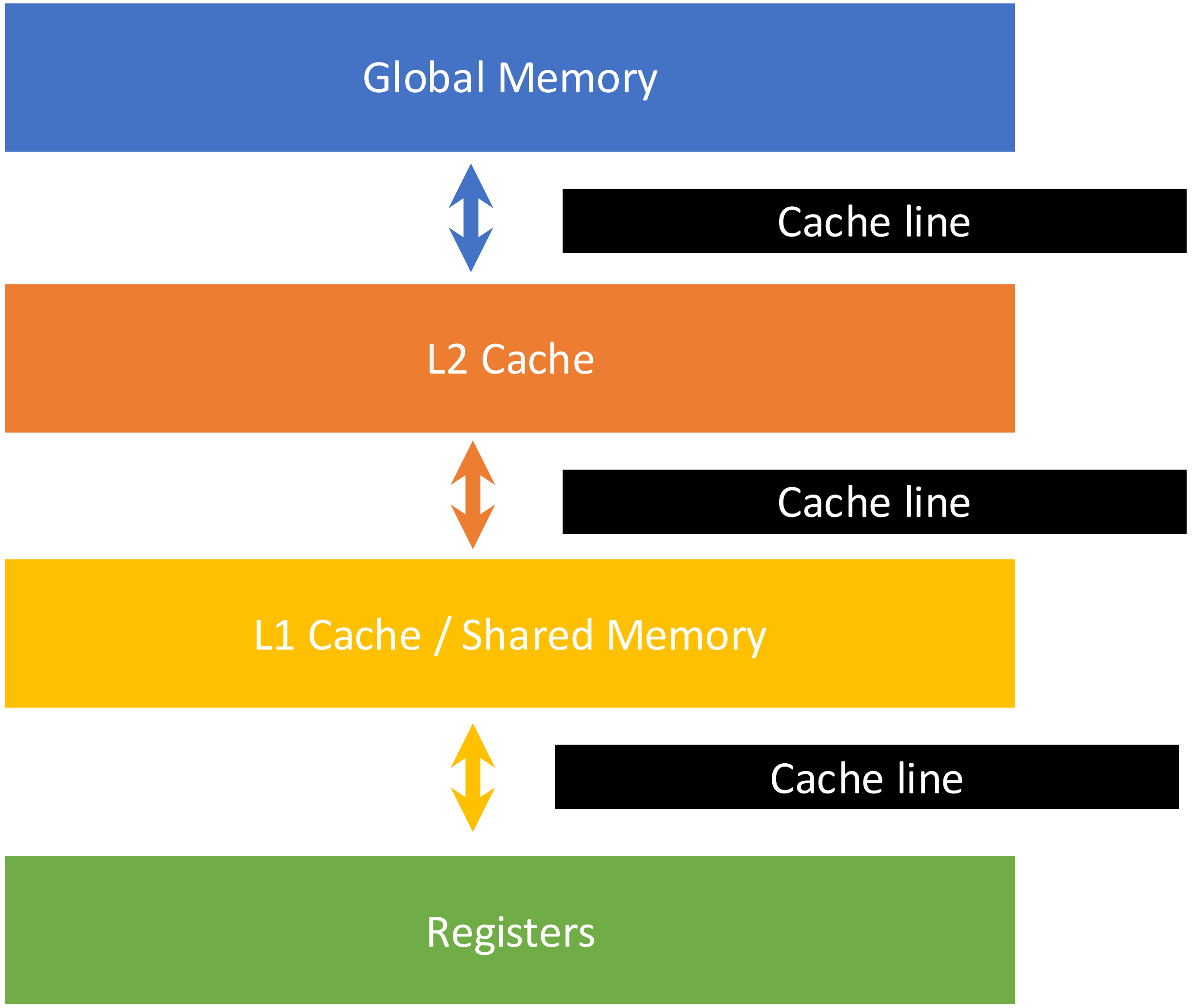
- Global Memory:
  - Visibility: all Compute Units
  - Capacity: ~ 100 GiB
- L2 cache :
  - Capacity: ~ 10 MiB per compute die
  - Visibility: a group of Compute Units on the same die
- L1 cache / shared memory :
  - Visibility: private to a Compute Unit
  - Capacity: ~ 60 KiB per Compute Unit

Memory usage



- All data needs to travel from global memory to registers across all layers
- If data is not found in a layer it needs to find the data in the upper layer
- As bottom layers have smaller capacity, some of the data might need to be overwritten

Memory usage



Global Memory

Cache line

L2 Cache

Cache line

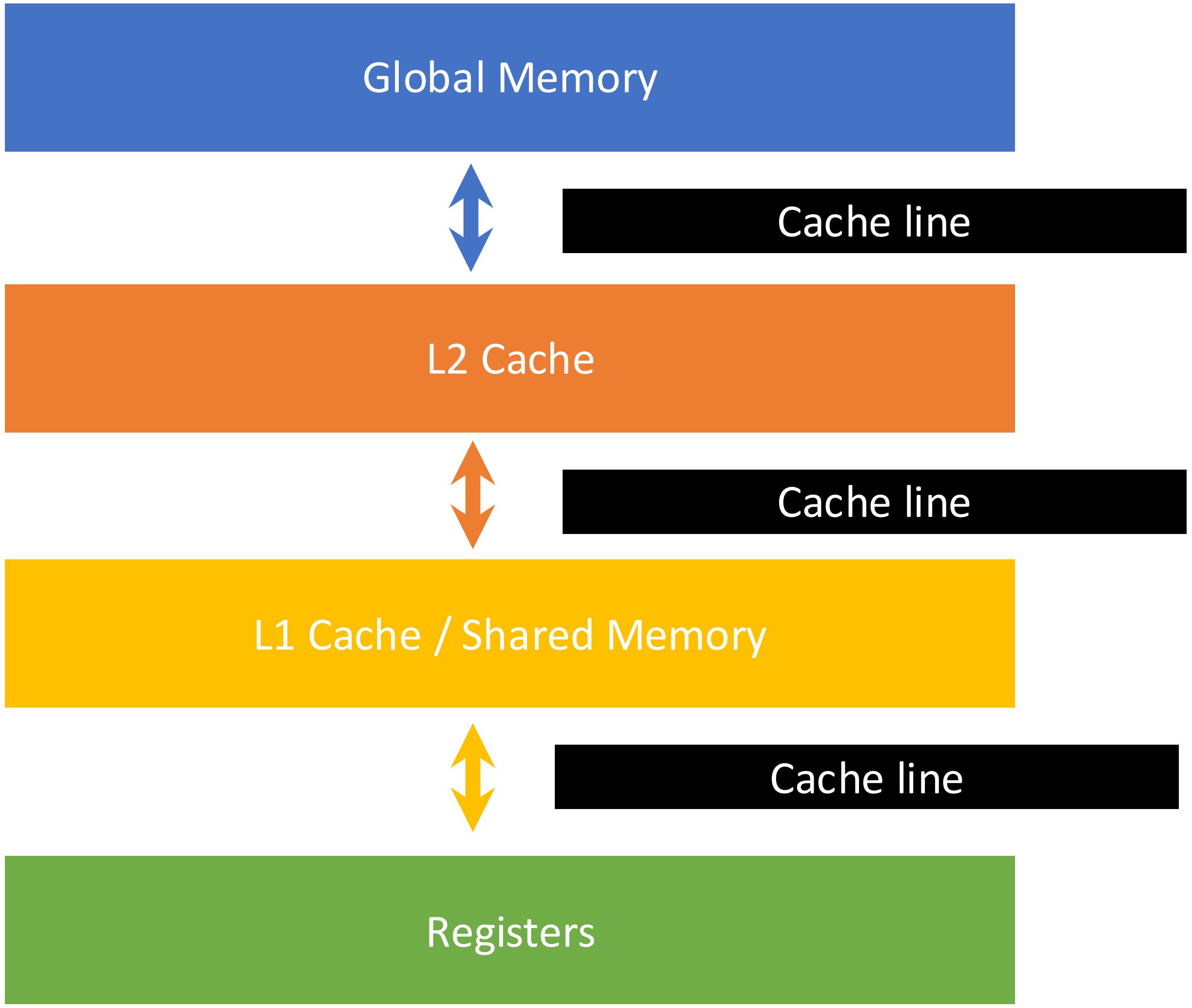
L1 Cache / Shared Memory

Cache line

Registers

- All data is transferred in bundles of a fixed size, called a cache line
- Cache line size depends on hardware. Usually 64 byte (i.e MI210) or 128 bytes (i.e. A100)
- If threads in the same wavefront access the same cache line, only one load instruction is needed: accesses are coalesced

Memory usage



- Aim to reduce the amount of data traffic between layers
- Try to reuse as much as possible data already in the layers

Coalesced memory access

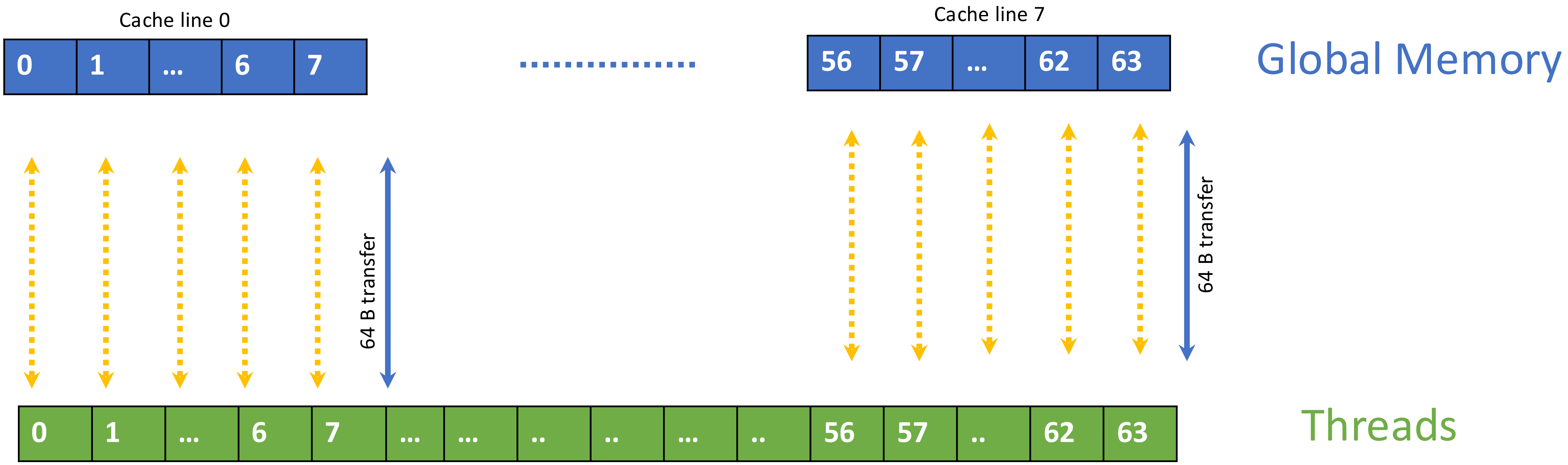
```
#pragma omp target teams distribute parallel for
for (int i=0; i<n; i++){
  a[i]= b[i] + c;
}
```

C/ C++

```
!$omp target teams distribute parallel do
   do i=1,n
       a(i) = b(i) + c
   end do
!$omp end target teams
```

F

Coalesced memory access



- 64 double precisions elements loaded from b
- 8 LOAD instructions per wavefront from b
- 64B x 8 = 512B transferred from b

Coalesced memory access

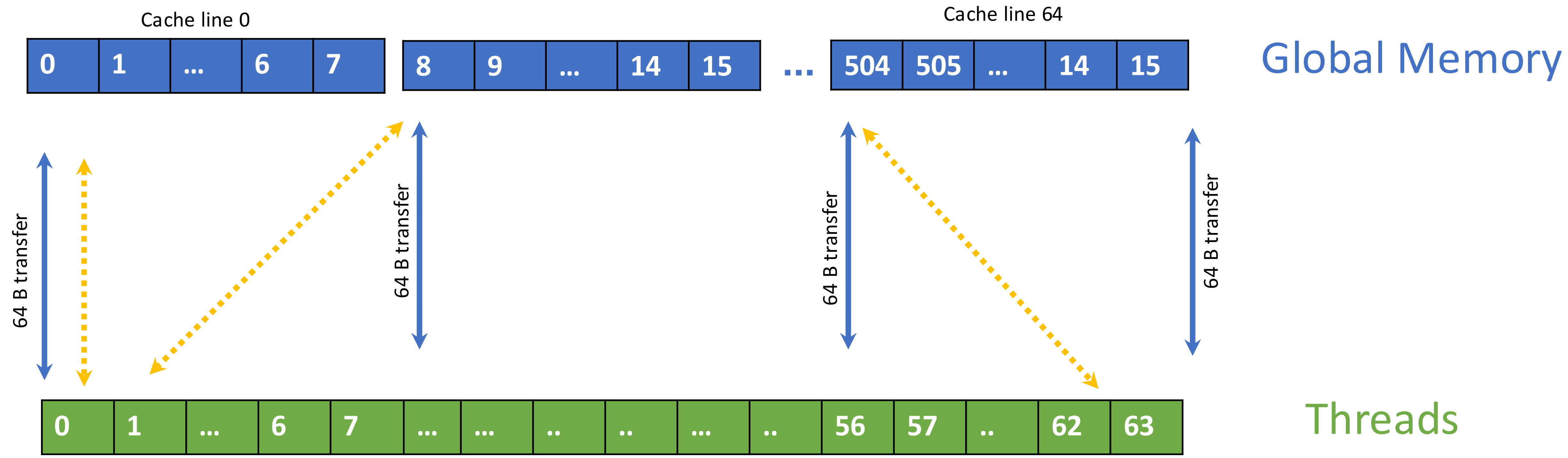
```
#pragma omp target teams distribute parallel for
for (int i=0; i<n; i++)
{
    a[i]= b[i*8] + c;
}
```

C/C++

```
!$omp target teams distribute parallel do
    do i=1,n
        a(i) = b(i*8) + c
    end do
!$omp end target teams
```

F

Coalesced memory access



- 64 double elements loaded from b
- 64 LOAD instructions per wavefront from b
- 64 B x 64 = 3.9 KiB transferred from b

Coalesced memory access
Loop ordering

- Inner loop iterations with the `collapse` clause are often assigned to neighbouring threads.
- Successive iterations of the inner loop more likely to be on the same thread

Coalesced memory access
Loop ordering

```
#pragma omp target teams distribute\
parallel for collapse(2)
for ( int i= 0; i<n; i++) {
  for ( int j= 0; j<n; j++) {
      a[i][j]= b[i][j] + c;
  }
}
```

C/C++

```
!$omp target teams distribute parallel do
  do j=1,n
    do i=1,n
       a(i,j) = b(i,j) + c
    end do
  end do
!$omp end target teams
```

F

- Inner loop threads are often assigned to neighbouring threads.
- Successive iterations of the inner loop more likely to be on the same thread

Coalesced memory access
Loop ordering

```
#pragma omp target teams distribute parallel for collapse(2)
for (int i= 0; i<n; i++) {
  for (int j= 0; j<n; j++) {
       a[i][j]= b[i][j] + c;
  }
}
```

C/C++

- In C/C++ rightmost array dimension is contiguous in memory
- j and j+1 element likely to be scheduled on the same wavefront
- Memory accesses are coalesced

Coalesced memory access
Loop ordering

```
#pragma omp target teams distribute parallel for collapse(2)
for (int j= 0; j<n; i++) {
  for (int i= 0; i<n; i++){
    a[i][j]= b[i][j] + c;
  }
}
```

C/C++

- Strided access between iterations with index i and i+1
- Memory accesses are not coalesced

Coalesced memory access
Loop ordering

```
!$omp target teams distribute parallel do collapse(2)
  do j=1,n
    do i=1,n
      a(i,j) = b(i,j) + c
    end do
  end do
!$omp end target teams
```

F

- In Fortran, leftmost array dimension is contiguous in memory
- i and i+1 elements likely to be scheduled on the same wavefront
- Memory accesses are coalesced

Coalesced memory access
Loop ordering

```
!$omp target teams distribute parallel do collapse(2)
  do i=1,n
    do j=1,n
       a(i,j) = b(i,j) + c
    end do
  end do
!$omp end target teams
```

F

- Strided access between iterations with index j and j+1
- Memory accesses are not coalesced

Miscellaneous

Kernel latency

- Scheduling a team on the device take times, regardless of how long it takes to execute
- Latency depends on size of the team, number of `firstprivate` variables used, etc.
- On MI210 a typical latency for offloading a computational kernel is about 5-10 microseconds

Kernel latency

- Try to have sufficiently long computation kernels, in order to hide kernel launch latency
- Merging small computational kernels into bigger computational kernels can help

Data movements

- Moving data between CPU and GPU is slow. Avoid un-necessary data transfers
- Use `enter data` / `exit data` to avoid transferring data to and from the GPU multiple times
- Use the `alloc` clause for temporary data only needed on the device
- Use the `update` directive to synchronize data between the CPU and GPU
- Try using `firstprivate` clause for scalar data, instead of mapping. Compiler might pass in the variable as a kernel launch argument, instead of having to map the variable to main memory

Use libraries

- Lots of available libraries for common scientific operations
- Some libraries are vendor independent (will work with AMD, NVIDIA, INTEL …) e.g. MAGMA
- Some are vendor dependent. But you can switch between implementations at build or run time, based on available hardware (e.g. rocBLAS, cuBLAS …)
- Might need `is_device_ptr`, `has_device_addr` clauses

Use asynchronous tasks

- If merging computational kernels is impossible or difficult, try to submit multiple kernels in parallel to the device
- Can use multiple threads for launching kernels. This is not always implemented efficiently in compilers.
- Use the `nowait` clause and `taskwait` construct
- Overlap computation on the device with other tasks on the CPU (e.g. communication)

Reduce OpenMP parallel overhead

- Try to always have
  `target teams distribute parallel for`
  in the same line
- (overhead of distributing teams) + (overhead of creating a parallel region in a team) > (overhead of distributing teams and creating a parallel region in one step)

Reduce OpenMP parallel overhead

```
#pragma omp target teams distribute
{
    // some code here
    #pragma omp parallel for
    {
    // some code here
    }
}
```

C/C++

- Larger overhead

```
#pragma omp target teams distribute parallel for
{
    // some code here
}
```

C/C++

- Smaller overhead

Reduce OpenMP parallel overhead

```
!$omp target teams distribute

! some code here
!$omp parallel do
! some code here

!$omp end target teams
```

F

- Larger overhead

```
!$omp target teams distribute parallel do
! all code here
!$omp end target teams
```

F

- Smaller overhead

Conclusions

- Ensure there is enough parallelism to fill all the compute units
- Avoid branches
- Optimize memory accesses, make sure neighbouring threads access neighbouring data elements in memory
- Have kernels sufficiently big to hide latency, but sufficiently simple not to run out of registers

Partners

Engineering and
Physical Sciences
Research Council

Natural
Environment
Research Council

epcc

THE UNIVERSITY
of EDINBURGH

Hewlett Packard
Enterprise