Programming paradigms for GPU devices

28th Summer School on Parallel Computing

1-12 July 2019

Sergio Orlandini
s.orlandini@cineca.it

SuperComputing Applications and Innovation

- Matrix Transpose
  - evaluating performance
  - using shared memory for coalesced accesses
  - avoiding bank-conflicts

Matrix Transpose

- Let's implement matrix transpose with the following simple design:
  - out-of-place buffers
  - square matrices with size modulo 32 elements
- This is a memory-bounded kernel
  - no computation on elements
  - just load and stores
- We will use effective bandwidth (GB/s) as a metric to measure the performance of such kernels

Simple Copy Kernel

```
__global__ void copy (float *idata, float *odata, int width, int height)
{
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

  int index_in  = width * yIndex + xIndex;
  int index_out = index_in;

  odata[index_out] = idata[index_in];
}
```

- We can use a simple copy kernel as a reference
  - **TILE_DIM** is the size of the square sub-matrix block
  - we map CUDA blocks to sub-matrix blocks

  matrix sub-block size == CUDA thread block size

Naive Transpose

```
__global__ void transposeNaive(float *idata, float *odata, int width, int height)
{
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

  int index_in  = width * yIndex + xIndex;
  int index_out = height * xIndex + yIndex;

  odata[index_out] = idata[index_in];
}
```

```
attributes(global) subroutine transposeNaive (idata, odata, width, height)
  integer, intent(in), value :: width, height
  real, intent(in) :: idata(width,height)
  real, intent(out) :: odata(height,width)

   i = ( blockIdx%x - 1 ) * TILE_DIM + threadIdx%x
   j = ( blockIdx%y - 1 ) * TILE_DIM + threadIdx%y

   odata(j,i) = idata(i,j)

end subroutine
```

SuperComputing Applications and Innovation

Measuring Performance

```
// take measurements for loop over kernel launches
cudaEventRecord(start);
for (int i=0; i < NUM_REPS; i++)
  kernel<<<grid, threads>>>(
        d_idata, d_odata, width, height);
cudaEventRecord(stop);
cudaEventSyncronize(stop);
float outerTime;
cudaEventElapsedTime(&outerTime, start, stop);


// take measurements for loop inside kernel
cudaEventRecord(start);
kernel<<<grid, threads>>>(
      d_idata, d_odata, width, height, NUM_REPS);
cudaEventRecord(stop);
cudaEventSyncronize(stop);
float innerTime;
cudaEventElapsedTime(&innerTime, start, stop);
```

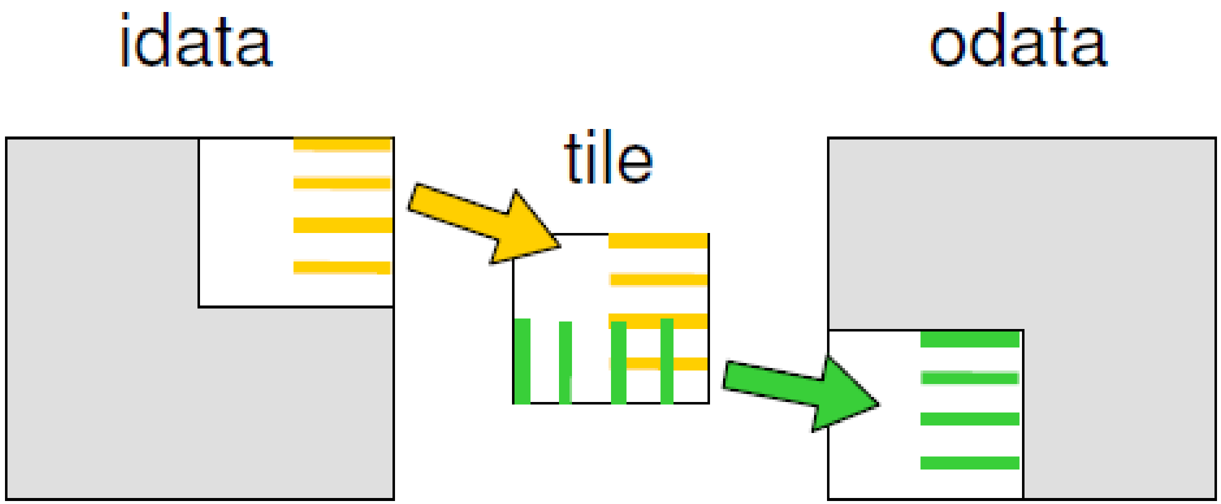
Effective Bandwidth (GB/s) on 2048x2048 S2050	
kernel	*Performance [GB/s]*
Copy	60.9
Naive	22.4

... what is happening?

Not coalesced Accesses

- All loads from input matrix are coalesced:
  - each *warp* reads a line of contiguous elements
    - 32 float belonging to the same cache line
- yet all stores into transposed matrix are not coalesced:
  - the copy kernel store by lines
  - the naive transpose kernel store by columns
    - threads in a warp write an elements into different segments
    - the matrix stride rules how distant those segments are
- the naive transpose kernel performs 32 different stores per row

Coalesced Transpose



- To avoid non-coalesced store we should store by row:
  - let's fill a tile in shared memory with data to be transposed
  - we don't get any penalty writing elements by columns into shared-memory
  - the transpose operation is now performed in shared-memory
  - once the tile is filled, we write back info global memory by rows

Coalesced Transpose

```
__global__ void transposeCoalesced(float *idata, float *odata, int
  width, int height)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in  = width * yIndex + xIndex;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = height * yIndex + xIndex;

  tile[threadIdx.y][threadIdx.x] = idata[index_in];

  __syncthreads();

  odata[index_out] = tile[threadIdx.x][threadIdx.y];
}
```

Coalesced Matrix Transpose

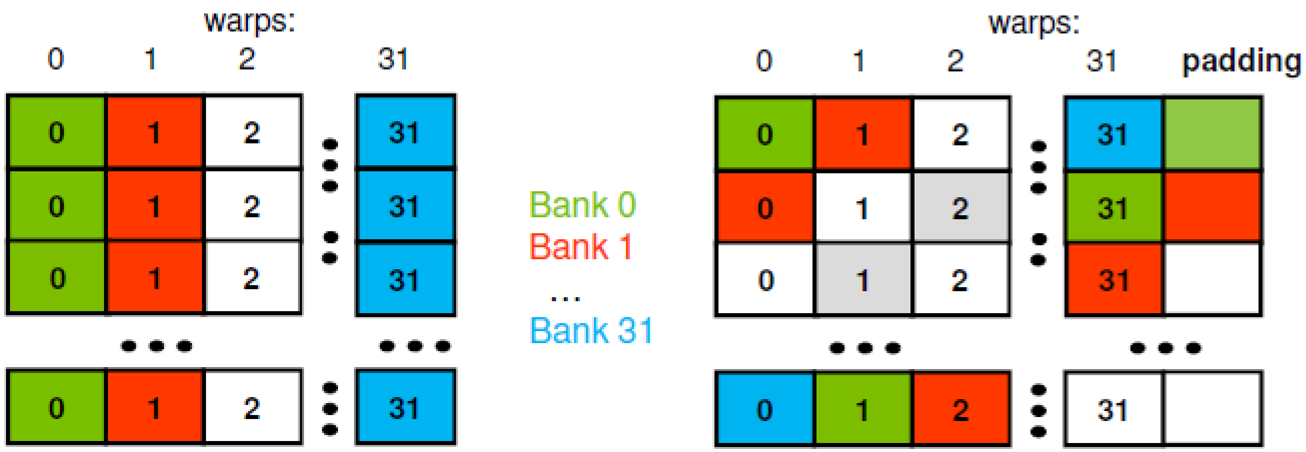
Effective Bandwidth (GB/s) on 2048x2048 S2050	
kernel	*Performance [GB/s]*
Copy	60.9
Naive	22.4
Coalesced	24.1

... mmm, we are still missing something

Avoiding Bank Conflict

- our new coalesced transpose use a shared memory tile of 32x32 float
  - each element resides on successive bank (4-byte)
  - accessing elements with a 32 size stride will fetch them **from the same bank**
  - any read/write access to this tile by column will get a 32 bank conflict
- use the trick! a new tile of 32x33 elements
  - element of the same tile column will resides on different banks
  - no more bank conflicts at all

```
__shared__ float tile[TILE_DIM][TILE_DIM+1];
```



Avoiding Bank Conflict

Effective Bandwidth (GB/s) on 2048x2048 S2050	
kernel	*Performance [GB/s]*
Copy	60.9
Naive	22.4
Coalesced	24.1
no Bank Conflicts	46.6

Rights & Credits

Slides and examples were authored by:

Isabella Baccarelli, Luca Ferraro, Sergio Orlandini