AN INTRODUCTION TO PARALLEL COMPUTING

CHAPTER 6: COALESCED MEMORY AND SPARSE MATRICES

BY FRED ANNEXSTEIN, PHD

6. COALESCED MEMORY AND SPARSE MATRICES

CHAPTER OBJECTIVES

In this chapter we will be covering two major memory bandwidth issues including techniques to increase the speed of block memory access and algorithmic techniques for addressing problems associated with sparse data. We will look at the concept of coalesced memory access for improving the memory bandwidth, and we will look at parallel algorithms and programming techniques for addressing sparse data. By the end of this chapter you will be able to...

1. Understand the performance advantages of parallel processing with higher granularity and coalesced memory access.
2. Understand the performance issues involved in compacting operations

utilizing scans and the generalized operation of segmented scans.
3. Understand the performance issues resulting from working with dense and sparse matrices.
4. Understand and apply an efficient coding technique called CSR format and use it in a parallel algorithm for sparse matrix vector multiplication.

DATA TILES

In the last chapter we looked at the problem of dividing up a matrix of numbers into a set of data tiles so that we could map the data matrix on to a set of independent thread blocks. Tiling is a very general issue since it is associated with the mapping problem in parallel computing.

When designing a parallel program for CUDA we need to determine the appropriate size of the data tiles. In designing CUDA applications tile sizes need to account for several system and program parameters. The number of tiles must be sufficiently many so that groups of thread blocks fill up the memory bus. Tile sizes must be set large enough so that the average latency is as low

as possible. Finally, tile sizes must be set small enough so that shared/register memory limits are not violated.

Larger tiles have the potential to increase the main memory bandwidth and hide latency, since it is possible to make better use of sequential global memory access. However larger tiles may make using shared memory more difficult because of strict limits on shared memory. Larger tile sizes have the advantage that it does not take any longer to synchronize threads, since this takes place at the block level.

WARPS AND STALLS

For a CUDA program a *grid* is composed of thread blocks which are scheduled to execute completely independently. Each block is composed of many threads (usually 512 or 1024) which can communicate within their own block through shared memory. Thread blocks execute instructions that are issued within an SM or symmetric multiprocessor on a per warp basis -- typically all threads in the block are divided into groups of 32 threads which define each warp. Warps are the unit of execution scheduling.

If any operand is not ready at the time the warp is scheduled to be executed, then the entire warp will *stall*, stopping progress on that warp until all the operands are ready.

At the time of a stall, a context switch must occur between warps so that a new warp is ready to execute. Context switching must be very fast so that performance is not significantly impacted.

FAST CONTEXT SWITCHING

CUDA is designed for fast context switching. This is accomplished by having registers and shared memory allocated for an entire block, and this memory is allocated for as long as that block is active. Once a block is active it will stay active until all threads in that block have completed all their work. Context switching is very fast because registers and shared memory do not need to be saved and restored.

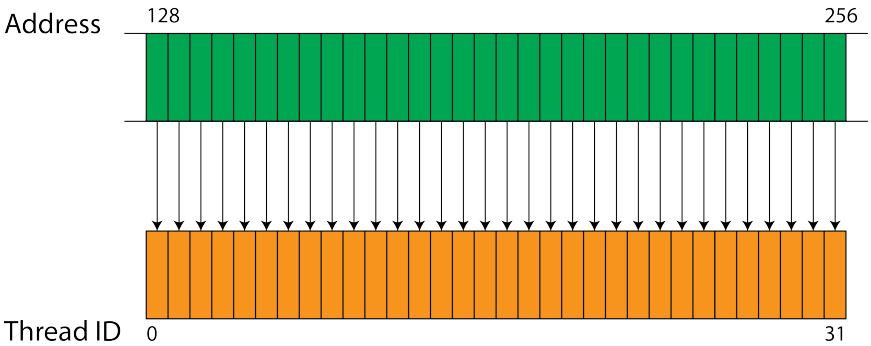
GRANULARITY

Granularity refers to the ratio of program computation vs communication time as measured on a per warp basis. Increasing granularity gives the memory system opportunities to optimize use of pipelining

and other performance features. We will address the design issues related to granularity later in this chapter.

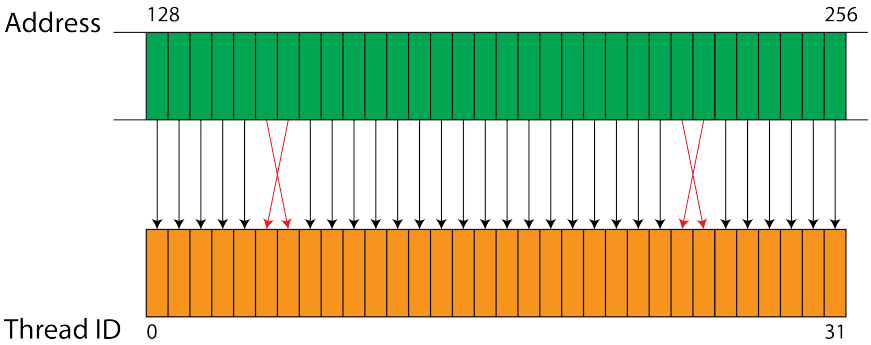
MEMORY COALESCING

Coalesced memory access refers to combining multiple memory accesses into a single transaction. As an example, we find that on a popular GPU every successive 128 bytes, or equivalently, 32 single precision words of memory can be accessed by a single warp of 32 consecutive threads in a single transaction. However, under certain conditions, coalesced memory access is not possible and memory access then becomes serialized resulting in poorer performance.

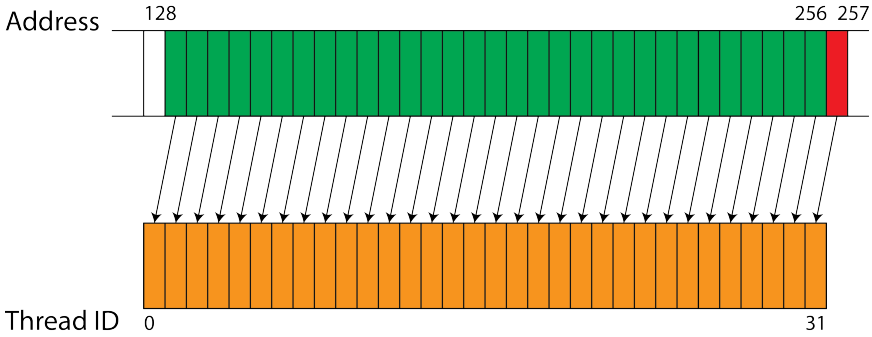


Coalesced memory access is possible when memory is *aligned* as illustrated in the figure above. The figure shows how 32 consecutive threads can access 32

consecutive words of memory. In such a case we say that the memory access is sequential and aligned, and this access pattern will yield the best memory performance.



In the figure above the memory is aligned but is not sequential. To achieve this memory pattern there will need to be multiple transactions or read cycles to account for the non-sequential nature of the access pattern.



In the figure above the memory is sequential but is misaligned. Therefore, it requires one transaction to load the first 31 words, and another transaction to load the last word. Two transactions are required in this case.

CODE EXAMPLE

A coalesced memory access instruction typically looks like the following within a kernel.

```
shmem[threadIdx.x]=
   gmem[blockIdx.x*blockDim.x + threadIdx.x];
```

This assignment statement executes a transfer from an array of global memory (gmem), to an array of shared memory (shmem), a transfer which is sequential and aligned.

The following similar instruction, but here includes a stride value is not a coalesced memory access, since the transfer is neither sequential nor aligned.

```
stride=4;
shmem[threadIdx.x]=
  gmem[stride*blockIdx.x*blockDim.x +
     threadIdx.x*stride];
```

If you have an global array of floats called f_data and you want to read a tile or block of data from this array starting at offset n, then thread 0 in the warp must read f_data[n], thread 1 must read f_data[n+1], and so on. Those 32-bit reads of floats, which are issued simultaneously, are merged into several 1024=32x32 bit reads in order to efficiently use the memory bus.

Coalesced memory access is a warp-level activity, and not a thread-level activity. CUDA memory throughput is maximized when 32 or 64-bit reads are coalesced. 128-bit coalescing has half the throughput of 64-bit reads. Note that 8 and 16 bit reads are not coalesced in CUDA.

COMPACTING SPARSE DATA SETS

If an array of size N has litte-oh o(N) non-zero items we call it *sparse array.* For example, an N=mxm matrix with only O(m) non-zero entries is a sparse matrix array, since the number of non-zero entries is the square root of the total number of entries.

Compacting is a common parallel computing and communication operation for processing sparse data sets, since this will reduce the data size by more than an order of magnitude. A compact operation applies a predicate filter to each item in an array sequence and returns a compacted array consisting of only those items from the array for which the predicate is true when applied to each of the items. For example, to compute a list of some small prime numbers we may use the following code in python which applies a filter to remove all multiples of 2 and 3.

```
>>> import numpy as np
>>> my_range =
        np.arange(start=1, stop=24, step=1)
>>> pred = lambda x: (x % 2 != 0 and x % 3 != 0)
>>> compact(pred, my_range)
>>> my_range
array([5, 7, 11, 13, 17, 19, 23])
```

IMPLEMENTING COMPACT IN PARALLEL

It is possible to frame the compacting operation as a parallel scatter operation. To do this we determine for each of the filtered (True valued) items x the computation of a reference index to x's final location in the compacted array.

We assume that the predicate filter can be applied to each element in the list in parallel in constant time, resulting in a potentially sparse bit vector of Boolean values. See the example below.

```
Here is an example array data:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Filtering with pred = lambda x: (x % 2 != 0 and x % 3
!= 0), produces array of bools:

0 0 0 1 0 1 0 0  0  1  0  1  0  0  0  1  0  1

Apply a prefix scan (or running sum) produces array:

0 0 0 1 1 2 2 2. 2. 3. 3. 4. 4. 4. 4. 5. 5. 6.
```

We can use these scan values as scatter addresses for the final array index positions of those original values selected by filter. That is the index of each element in the filtered or compacted list is the running sum, or more precisely, the exclusive sum scan. For example, we see that 17 is in the 4th list position since its running sum value is 5 which is 1 greater than its left neighbor which is 4.

SPARSE MATRIX PROCESSING

We now look at the problem of effectively process sparse matrices so that we can compute a fast matrix vector product, for example. To this end we build a data format for each sparse matrix using 3 dense vectors called the CSR format. The CSR format will use an amount of space at most 3 times the number of non-zero entries (NNZ).

```
V = Value vector = all nonzero data compacted in one array

COL_INDEX = Column pointers which identifies column index for each value stored in V

ROW_INDEX = Row_index value is the total number of nonzeros above row. This value also encodes the index in V and COL_INDEX where the given row starts.
```

Here is an example of 4×6 matrix with total 24 entries and only 8 is NNZ.

$$\begin{pmatrix} 10 & 20 & 0 & 0 & 0 & 0 \\ 0 & 30 & 0 & 40 & 0 & 0 \\ 0 & 0 & 50 & 60 & 70 & 0 \\ 0 & 0 & 0 & 0 & 0 & 80 \end{pmatrix}$$

```
CSR format for matrix above.

V  = [10 20 30 40 50 60 70 80]

COL_INDEX = [0  1  1  3  2  3  4  5]

ROW_INDEX = [0  2  4  7  8]
```

The whole matrix is stored as 21 entries: 8 in V, 8 in $\mathrm{COL\_INDEX}$, and 5 in $\mathrm{ROW\_INDEX}$. $\mathrm{ROW\_INDEX}$ splits the array V into rows: (10,

20) (30, 40) (50, 60, 70) (80), indicating the index of V (and COL_INDEX) where each row starts and ends; COL_INDEX aligns values in columns: (10, 20, ...) (0, 30, 0, 40, ...)(0, 0, 50, 60, 70, 0) (0, 0, 0, 0, 0, 80). Note that in this format, the first value of ROW_INDEX is always zero and the last is always the number of non-zero entries.

MULTIPLYING SPARSE MATRICES WITH VECTORS

We can use the sparse matrix encoding given in CSR format defined above to multiply any matrix M by any vector X= (x0,x1,x2, …., xn) as follows.

We will reduce the multiplication problem to that of component-wise multiplication with the dense value vector V along with another vector with values coming from X. This vector can be constructed from the COL_INDEX vector indicating which x_i value is to be multiplied with each non-zero value in V. Finally, to compute the final values of the matrix vector product, we only need to sum-reduce the partial products for each row. This last step can be done with a segmented reduction scan, described as follows.

SEGMENTED REDUCTIONS

Sparse matrix-vector multiplication algorithm describe above requires a final step for a sum reduction for each row of the matrix. We can do all these reductions in parallel as follows. We will modify a standard sum reduction scan operations, in such a way that we introduce a special symbol indicating the row segments within the array VX of component wise multiplies. All reductions are then done on a per segment basis.

We are able to account for segments using a more generalized operation called a segmented scan. A segmented scan is a modification of the prefix sum operation with an equal-sized array of flag bits to denote segment boundaries on which the scan should be performed. Here is an example:

1	2	3	4	5	6	input
1	0	0	1	0	1	flag bits
1	3	6	4	9	6	segmented scan +

The flag bits can be used to zero-out the prefix outcomes so that no previous values are added to the segmented sums.

Segmented reductions or segmented scans can run with the same work and step complexity as ordinary scan operations.

IMPACT OF LOAD BALANCING

To effectively process space matrices in parallel, we can first encode the matrices with dense vectors in a CSR format. To compute a product of a sparse-matrix with a vector, we can apply a segmented scan operations. As we have seen these scan operation will operate on the block level, and so we must consider the granularity when we map tiles to blocks.

There are two basic agglomerations we can consider: agglomerating data so that we assign one thread per array element, or one thread per row of the matrix. In order to address this issue we must consider the load imbalance given the assumption that we have no prior knowledge of the number of non-zero values in each row. This suggests that one thread per row agglomeration may give us better load balancing properties.