Optimization Examples: Fusion and Coarsening

Soumyajit Dey, Assistant Professor,
CSE, IIT Kharagpur

December 25, 2019

Course Organization

Topic	Week	Hours
Review of basic COA w.r.t. performance	1	2
Intro to GPU architectures	2	3
Intro to CUDA programming	3	2
Multi-dimensional data and synchronization	4	2
Warp Scheduling and Divergence	5	2
Memory Access Coalescing	6	2
Optimizing Reduction Kernels	7	3
Kernel Fusion, Thread and Block Coarsening	8	3
OpenCL - runtime system	9	3
OpenCL - heterogeneous computing	10	2
Efficient Neural Network Training/Inferencing	11-12	6

Recap

- Multi-dimensional mapping of dataspace
- Synchronization
- Warp scheduling
- Divergence

Recap

- Memory Access Coalescing
- Tiling
- Matrix Multiply, Convolution
- GPU optimization techniques

GPU Optimization techniques

Some examples of GPU Optimization techniques-

- Reduction
- **Fusion**
- Coarsening

Loop Fusion

- Classical compiler optimization in programming
- Improve performance by reducing off-chip memory traffic
  - reduction of cache miss
  - better control of multiple instruction
  - reduces branching condition
- Operates by fusing iterations of different loops when those iterations reference the same data

Loop Fusion

```
//Before Fusion
for (i = 0; i < 300; i++)
        a[i] = a[i] + 3;
for (i = 0; i < 300; i++)
        b[i] = b[i] + 4;

//After Fusion
for (i = 0; i < 300; i++)
{
        a[i] = a[i] + 3;
        b[i] = b[i] + 4;
}
```


Kernel Fusion

- An optimization technique applied to a group of GPU kernels to increase efficiency by decreasing execution time, power consumption
- GPU kernels cannot be scheduled once launched in device
- Kernel fusion can rearrange and schedule the kernels from the host side
- Kernels using the same or different data array(s) can be replaced with a single kernel call
- The new kernel aggregates the code segments of the separate kernels

Advantages

Increase efficiency by -

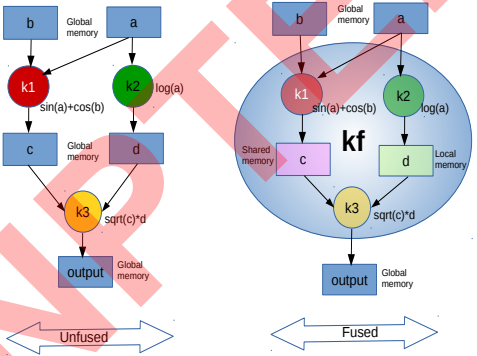
- data reuse using on-chip memory improves performance
- reducing off-chip memory data traffic
- reducing global memory data transfers
- reducing kernel launch overhead
- utilising maximum threads in GPU

Limitations

Kernel fusion does not always result in performance improvement:

- architectural resources like the capacity of on-chip memory and registers are limited
- reduction of off-chip memory data traffic is feasible upto a certain limit
- overhead in identifying fusable kernels
- overhead in defining a scalable method to search for the optimal rearrangement of fusable kernels
- may introduce divergence

Kernel Fusion example



Code snippet for Kernel 1

```
__global__ void
process_kernel1(float *a, float *b, float *c, int datasize) {
int blockNum=blockIdx.z*(gridDim.x*gridDim.y)+blockIdx.y*gridDim.x+blockIdx.x;
int threadNum=threadIdx.z*(blockDim.x*blockDim.y)+threadIdx.y*blockDim.x+
    threadIdx.x;
int i=blockNum*(blockDim.x*blockDim.y*blockDim.z)+threadNum;
if (i<datasize)
        c[i]=sin(a[i])+cos(b[i]);
}
```


Code snippet for Kernel 2

```
__global__ void
process_kernel2(float *a, float *d, int datasize) {

        int blockNum=...
        int threadNum=...
        int i=...

        if (i<datasize)
                d[i]=log(a[i]);
}
```

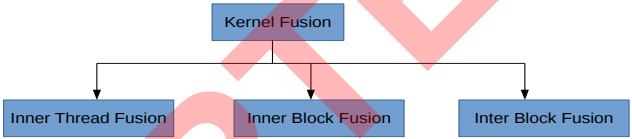

Code snippet for Kernel 3

```
__global__ void
process_kernel3(float *c, float *d, float *output, int datasize){

        int blockNum=...
        int threadNum=...
        int i=...

        if (i<datasize)
                output[i]=sqrt(c[i])*d[i];
}
```

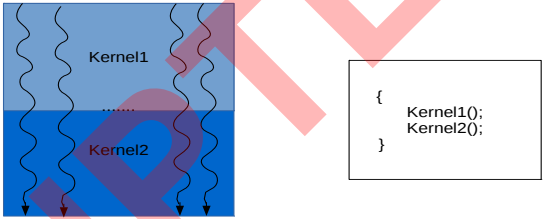

Code snippet for fused kernel

```
__global__ void
process_fused_kernel(float *a, float *b, float *output, int datasize) {

        __shared__ float c[blockDim.x * blockDim.y * blockDim.z];
        float d;
        int blockNum=...
        int threadNum=...
        int i=...
        if (i<datasize) {
                c[threadNum]=sin(a[i])+cos(b[i]);
                d=log(a[i];
                output[i]=sqrt(c[threadNum])*d;
        }
}
```


Types


Kernel Fusion
Inner Thread Fusion
Inner Block Fusion
Inter Block Fusion

Inner Thread Fusion



Inner Thread Fusion

- Combines computation of two kernel into single thread
- Suitable for both dependent and independent kernels if dataspace size is same
- Let, $S_{th,i}$ represents the number of threads in a thread block;
  $S_{bk,i}$ represents the number of thread blocks in kernel i($i = 1, 2$)
- For fused kernel -
  - $S_{th} = \max(S_{th,1}, S_{th,2})$
  - $S_{bk} = \max(S_{bk,1}, S_{bk,2})$
- Not suitable if -
  - Kernels not having equal number of threads/block
  - Results in unbalanced workload across threads

Inner Thread Fusion Example

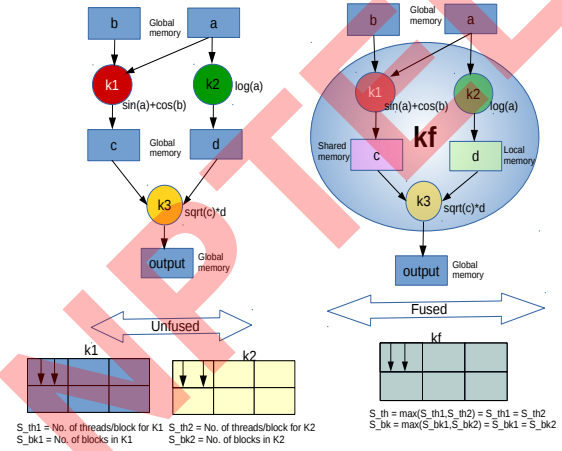


Figure: Inner Thread Fusion of dependent kernels with same data space size and thread/block

Inner Thread Fusion Example

Fusion of dependent kernels with same dataspace size and thread/block size

```
//Unfused kernels
k1(a, b, c, n):
        i = global threadId
        c[i]=sin(a[i])+cos(b[i])
k2(a, d, n) :
        i = global threadId
        d[i]=log(a[i])
//Fused kernel
kf(a, b, out , n):
        local c,d
        i = global threadId
        if(i<n)
                c=sin(a[i])+cos(b[i]);
                d=log(a[i];
                output[i]=sqrt(c)*d);
```


Inner Thread Fusion Example

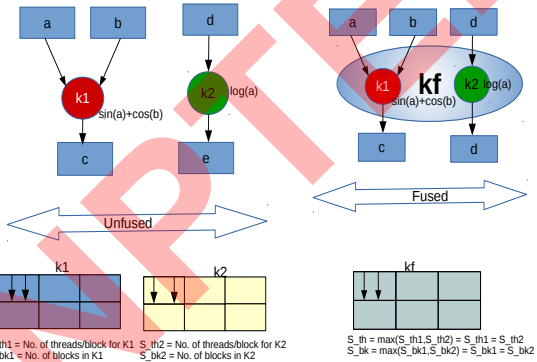


Figure: Inner Thread Fusion of independent kernels with same datasource size and thread/block

Inner Thread Fusion Example

Fusion of independent kernels with same dataspace size and thread/block size

```
//Unfused kernels
k1(a, b, c, n):
        i = global threadId
        c[i]=sin(a[i])+cos(b[i])
k2(d, e, n) :
        i = global threadId
        e[i]=log(d[i])

//Fused kernel
kf(a, b, c, d, e, n):
i = global threadId
if(i<n)
        c[i]=sin(a[i])+cos(b[i])
        e[i]=log(d[i])
```


Inner Thread Fusion Example

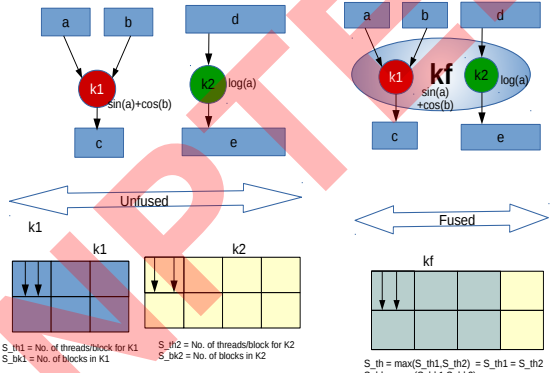


Figure: Inner Thread Fusion of independent kernels with different data size but same

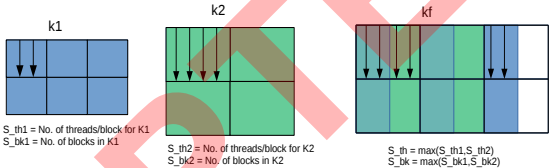
Inner Thread Fusion

Fusion of independent kernels with different data size but same thread/block size:

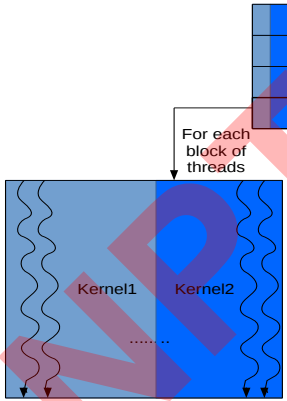
```
//Fused kernel
//Let n2>n1
kf(a, b, c, d, e, n1, n2):
i = global threadId
if(i<n1)
        c[i]=sin(a[i])+cos(b[i])
        e[i]=log(d[i])
else if(i<n2)
        e[i]=log(d[i])
```


Inner Thread Fusion Limitation

Fusion of independent kernels with different data size and threads/block :
NOT SUITABLE: Due to unbalanced workloads between threads



Inner Block Fusion

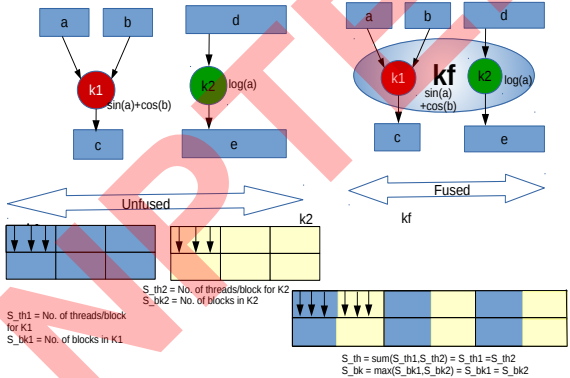



```
{
If (localThreadId < X)
        Kernel1();
    Else
        Kernel2();
}
```

Inner Block Fusion

- Distribute computation of two different kernels among threads in single block
- For independent kernels with small block size(threads/block)
- Let, $S_{th,i}$ represents the number of threads in a thread block;
  $S_{bk,i}$ represents the number of blocks in kernel i($i = 1, 2$)
- For fused kernel -
  - $S_{th} = \text{sum}(S_{th,1}, S_{th,2})$
  - $S_{bk} = \max(S_{bk,1}, S_{bk,2})$
- Not suitable if -
  - S_{th} of fused kernel exceed upper bound of threads/block
  - kernels with synchronization statement

Inner Block Fusion Example



Inner Block Fusion Example

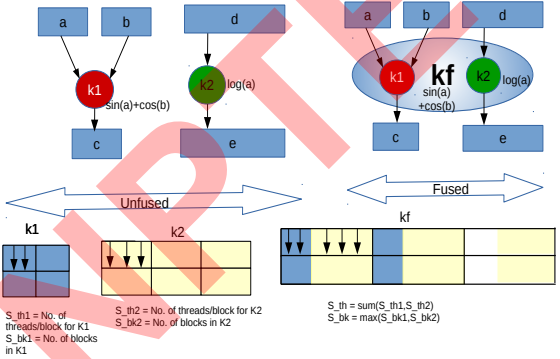
Fusion of independent kernels with same dataspace size

```
//Unfused kernels
k1(a, b, c, n):
        i = global threadId
        c[i]=sin(a[i])+cos(b[i])

k2(d, e, n) :
        i = global threadId
        e[i]=log(d[i])

//Fused kernel
//S_th1 = No. of threads/block for K1
//S_th2 = No. of threads/block for K2
//S_th = sum(S_th1,S_th2)
```


Inner Block Fusion Example

```
kf(a, b, c, d, e, n):
        b = blockId
        t = threadId

        if(t<S_th1)
                c[t]=sin(a[t])+cos(b[t])
        else if(t<S_th)
                e[t-S_th1]=log(d[t-S_th1])
```


Inner Block Fusion Example



Inner Block Fusion Example

Fusion of independent kernels with different dataspace size

```
//Unfused kernels
k1(a, b, c, n1):
        i = global threadId
        c[i]=sin(a[i])+cos(b[i])

k2(d, e, n2) :
        i = global threadId
        e[i]=log(d[i])

//Fused kernel
//S_th = sum(S_th1,S_th2)
//S_bk = max(S_bk1,S_bk2)
//Let S_th2 > S_th1
//Let S_bk2 > S_bk1
```


Inner Block Fusion Example

```
kf(a, b, c, d, e, n1, n2, X):
        b = blockId
        t = threadId
        if(b<S_bk)
                if(t<S_th1 AND b<S_bk1)
                        c[t]=sin(a[t])+cos(b[t])
                else if(t<S_th)
                        e[t-S_th1]=log(d[t-S_th1])
```


Inner Block Fusion Limitation

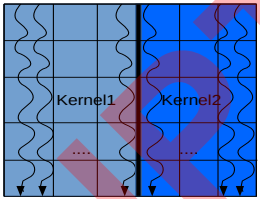
Upper bound for threads/block is architecture dependent

- S_{th} of fused kernel must not exceed this upper bound

CUDA does not support synchronization for partial threads in a block

- kernels with sync() statement like reduction kernel not applicable for this type of fushion

Inter block Fusion

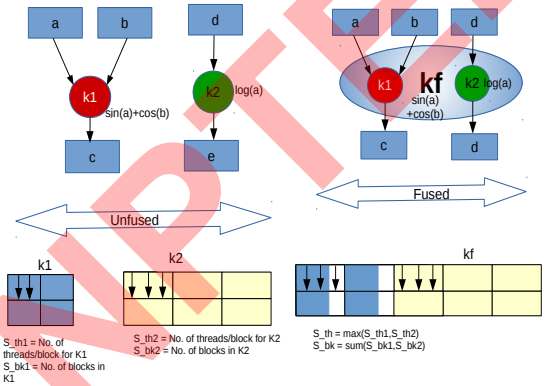



```
{
  If (blockId < X)
     Kernel1();
  Else
     Kernel2();
}
```


Inter block Fusion

- Distribute computation of two different kernels among different blocks
- For independent kernels with similar computation time
- Let, $S_{th,i}$ represents the number of threads in a thread block;
  $S_{bk,i}$ represents the number of blocks in kernel i(i = 1, 2)
- For fused kernel -
  - $S_{th} = \max(S_{th,1}, S_{th,2})$
  - $S_{bk} = \text{sum}(S_{bk,1}, S_{bk,2})$
- Not suitable if -
  - Workload of different thread blocks differs a lot

Inter Block Fusion Example



Inter Block Fusion Example

Fusion of independent kernels with different dataspace size

```
//Unfused kernels
k1(a, b, c, n1):
        i = global threadId
        c[i]=sin(a[i])+cos(b[i])
k2(d, e, n2) :
        i = global threadId
        e[i]=log(d[i])

//Fused kernel
//S_th = max(S_th1,S_th2)
//S_bk = sum(S_bk1,S_bk2)
//Let S_th2 > S_th1
//Let S_bk2 > S_bk1
```


Inter Block Fusion Example

```
kf(a, b, c, d, e, n1, n2):
b = blockId
t = threadId
i = global threadId

if(b<S_bk1)
        if(t<S_th1)
                c[t]=sin(a[t])+cos(b[t])
else if(b<S_bk)
        if(t<S_th2)
                e[t]=log(d[t])
```


Inter Block Fusion Limitation

Workload of different thread blocks differs a lot. For example, below two kernels are not suitable for this type of fusion.

```
k1(a, b, c, n1):
        i = global threadId
        c[i]=sqrt(sin(a[i])+cos(b[i]))
k2(d, e, f, n2) :
        i = global threadId
        f[i]=d[i]+e[i]
```


GPU Optimization techniques

Some examples of GPU Optimization techniques-

- Reduction
- Fusion
- **Coarsening**

Thread Coarsening

- Parallel execution sometimes requires doing -
  - Redundant memory accesses
  - Redundant calculations
- Merging multiple threads into one
- To allow re-use of result, avoiding redundant work
- Similar to loop unrolling, but applied across parallel threads rather than across serial loop iterations.

Effects of Thread Coarsening

Thread Coarsening results in reduction in parallelism.

- Beneficial effects:
  - Coarsening requires only less number of threads to be launched
  - Barrier execution is reduced
  - Increased number of instructions increases scope for exploiting hardware instruction-level parallelism

Effects of Thread Coarsening

Thread Coarsening results in reduction in parallelism.

- Detrimental effects:
  - Raises a kernel's resource consumption like registers, eventually resulting in reduced occupancy
    (Occupancy is the ratio of the number of active threads per SM and maximum numberof active threads allowed per SM)
  - Also increase the pressure on the cache in some kernels

Simple Example

Without coarsening:

```
__global__ void square(int *
    g_idata, int *g_odata,
    unsigned int n)
{
  unsigned int gid = threadIdx.
      x+ blockDim.x*blockIdx.x;
  if(gid<n)
    g_odata[gid] = g_idata[gid]
  *g_idata[gid] ;
}
```

With coarsening:

```
__global__ void square(int *g_idata, int *
    g_odata, unsigned int n)
{
    unsigned int gid = threadIdx.x +
    blockDim.x*blockIdx.x;
    if(gid<n) {
        int tid0 = 2 * gid + 0;
        int tid1 = 2 * gid + 1;
        g_odata[tid0] = g_idata[tid0]*g_idata[tid0];
        g_odata[tid1] = g_idata[tid1]*g_idata[tid1];
    }
}
```


Outline of Technique

- Merge multiple threads so each resulting thread calculates multiple output elements
  - Perform the redundant work once
  - Save result into registers
  - Use register result to calculate multiple output elements

Outline of Technique

- Merged kernel code will use more registers
  - May reduce the number of threads allowed on single SM
  - Increased efficiency may outweigh reduced parallelism for a given hardware

Register Tiling

With thread coarsening, computation from merged threads can now share registers.
Properties of registers:

- extremely fast (short latency)
- do not require memory access instructions (high throughput)
- private to each thread
- threads cannot share computation results or loaded memory data through registers

Coarsening Factor

Coarsening Factor -

- is the number of times the body of the thread is replicated,
- gives the best performance depending on the program and on the hardware.

Coarsening Factor for any problem in a hardware can be decided -

- Through profiling
- Using static analysis

Types of Coarsening

- **Thread-level coarsening:** Increase granularity within a single block of threads
- **Block-level coarsening:** Increase granularity across multiple blocks

Thread-level Coarsening

- Applies coarsening at the level of individual threads
- Combine two or more threads from the same block
- Each thread block performs the same amount of work but with fewer threads
- But each SM has limitations in terms of registers, shared memory, and concurrently runnable thread blocks
- These hardware constraints bound how much to coarsen.

Stride Length

- Acts as an offset between the IDs of threads that are to be combined
- Max stride length $\leq$ (Number of thread per block in the dimension where coarsening is applied)/ Coarsening Factor
- Minimum stride length $\geq$ Warp size to ensure memory coalescing

Thread-level Coarsening Example

```
__global__ void
thread_coarsened_reduce3(float *g_idata, float *g_odata, unsigned int n)
{
//Apply thread coarsening factor 2 and stride 32
// The coarsening factor dictates how many replicas of the local thread id and
     global thread id will be in the program
// Since we have coarsening 2, we will have two instances of tid (local thread
     id) and i (global thread id)

unsigned int tid0 = (threadIdx.x/32)*32*2 + threadIdx.x%32;
unsigned int tid1 = tid0+32;
unsigned int i0 = blockIdx.x*2*blockDim.x + tid0;
unsigned int i1 = blockIdx.x*2*blockDim.x + tid1;
```


Thread-level Coarsening Example

```
// load shared mem
__shared__ float sdata[BLOCK_SIZE]
sdata[tid0] = (i0 < n) ? g_idata[i0] : 0;
sdata[tid1] = (i1 < n) ? g_idata[i1] : 0;
__syncthreads();

// do reduction in shared mem
for (unsigned int s=2*blockDim.x/2; s>0; s>>=1) //Note every instance of
    blockDim.x gets multiplied by the coarsening factor
{
        if (tid0 < s)
                sdata[tid0] += sdata[tid0 + s];
        if (tid1 < s)
                sdata[tid1] += sdata[tid1 + s];
        __syncthreads();
}
// Note in the for loop, sdata is updated by both tid0 and tid1
```


Thread-level Coarsening Example

```
// only one write - the condition of the second will never be true
if (tid0 == 0)
        g_odata[blockIdx.x] = sdata[0];
if (tid1 == 0)
        g_odata[blockIdx.x] = sdata[0];

}
```


Block-level Coarsening

- Combines the work of several thread blocks into one block
- Number of threads per block remains unchanged so the number of executed thread blocks is reduced
- Each block has to handle an increased workload
- Resource requirements per block, in terms of register and shared memory usage, will typically increase

Stride Length

Stride Length

- Acts as an offset between the IDs of blocks that are to be merged
- Max stride length $\leq$ (Number of blocks in the dimension where coarsening is applied)/ Coarsening Factor
- Minimum stride length ≥ 1
  (Stride has no influence on memory coalescing as the memory access pattern within each block are preserved)

Block-level Coarsening Example

```
__global__ void
block_coarsened_reduce3(float *g_idata, float *g_odata, unsigned int n)
{
//Local Thread Id remains unchanged
//The coarsening factor dictates how many replicas of global thread id will be
     there in the program

unsigned int tid = threadIdx.x;
unsigned int i0 = 2*blockIdx.x*blockDim.x + threadIdx.x;
unsigned int i1 = (2*blockIdx.x+1)*blockDim.x + threadIdx.x;
//Apply block coarsening factor 2 and stride 1
```


Block-level Coarsening Example

```
// Shared memory requirements increase for block coarsening. Here since
    coarsening factor is 2, it is doubled

__shared__ float sdata0[BLOCK_SIZE];
__shared__ float sdata1[BLOCK_SIZE];
sdata0[tid] = (i0 < n) ? g_idata[i0] : 0;
sdata1[tid] = (i1 < n) ? g_idata[i1] : 0;
__syncthreads();

// do reduction in shared mem
for (unsigned int s=blockDim.x/2; s>0; s>>=1)
{
        if (tid < s)
        {
                sdata0[tid] += sdata0[tid + s];
                sdata1[tid] += sdata1[tid + s];
        }
        __syncthreads();
}
```


Block-level Coarsening Example

```
        // write result for this block to global mem
        if (tid == 0)
        {
                g_odata[2*blockIdx.x] = sdata0[0];
                g_odata[2*blockIdx.x+1] = sdata1[0];
        }
}
```


Recap Reduction

Kernel	Optimization
Reduce1	Interleaved addressing (using modulo arithmetic) with divergent branching
Reduce2	Interleaved addressing (using contiguous threads) with bank conflicts
Reduce3	Sequential addressing, no divergence or bank conflicts
Reduce4	Uses n/2 threads, performs first level during global load
Reduce5	Unrolled loop for last warp, intra-warp synchronisation barriers removed
Reduce6	Completely unrolled, using template parameter to assert whether the number of threads is a power of two
Reduce7	Multiple elements per thread, small constant number of thread blocks launched. Requires very few synchronisation barriers

Coarsening Analysis on Reduction

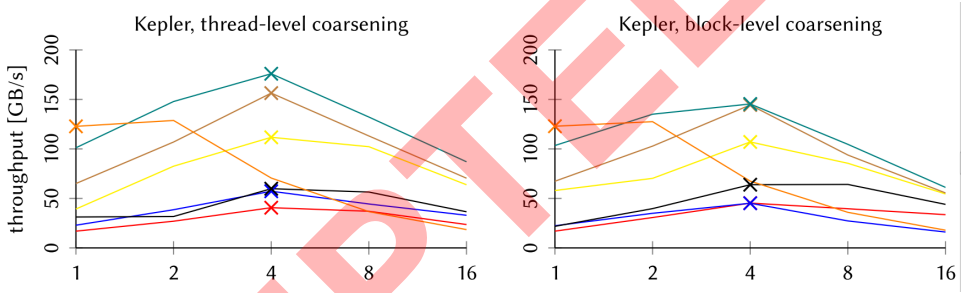


Figure: Reduction benchmarks' Throughput vs coarsening factor (Figure from reference[3])

For reduce 7, there is loop for global load sequence -> already significant per-thread activity

Target Architecture and Test Data Set

Device	GeForce GTX TITAN Black
GPU	NVidia Kepler
Streaming Multiprocessors (SM)	15
Max. active threads per SM	2048
Max. thread blocks per SM	16
Registers per SM	64K
Shared Memory per SM	48KB
Max. Shared Memory per block	48KB
Test data set size	512MB
Data-type for test data set	Single-precision floating point (4 byte)
Block Size used for uncoarsened code	512
Shared Memory requirement per block	512*4=2048 byte

Coarsening Factor Calculation

Thread-level Coarsening:

Let x be the coarsening factor.
Block size = 512/x
Active blocks per SM = 2048/(512/x)=4*x
For thread-level coarsening, shared memory requirement per block is same.
Shared memory requirement per block = 512*4=2048 byte
Shared memory requirement per SM = (4*x) * 2048 byte
Max allowable shared memory per SM = 48KB.
(4*x) * 2048 byte = 48 * 1024 byte
$\therefore$ x = 6 $\simeq$ **4** (since $4 = 2^2 \leq 6 \leq 2^3$)

Coarsening Factor Calculation

Block-level Coarsening:
Let x be the coarsening factor.
Block size = 512
Active blocks per SM = 2048/512=4
Shared memory requirement per block = 2048*x byte
Shared memory requirement per SM = 4 * (2048*x) byte
Max allowable shared memory per SM = 48KB.
4 * (2048*x) byte = 48 * 1024 byte
$\therefore x = 6 \simeq \mathbf{4}$ $(2^2 \leq 6 \leq 2^3)$

For the initial thread block size 512,

- In the general case we have 2048 active thread contexts, but the shared memory is not fully utilized by these threads inside the SM
- After thread/block level coarsening, per thread shared memory requirement is increased so that the 2048 active thread contexts fully utilize the entire shared memory of each SM

Coarsening Analysis on Reduction

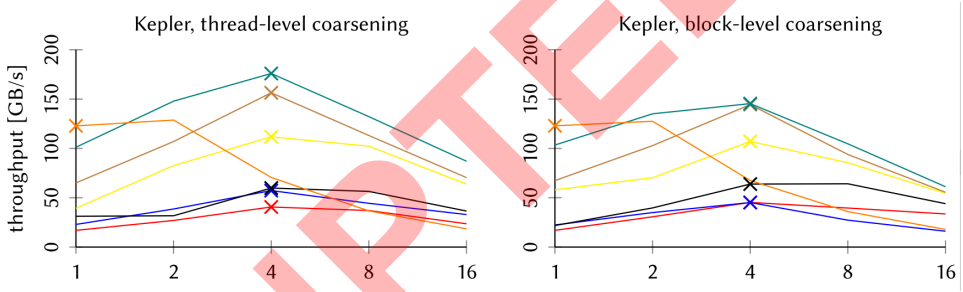


Figure: Reduction benchmarks' Throughput vs coarsening factor (Figure from reference[3])

Detrimental Effects of Thread Coarsening

- Raises a kernel's resource consumption like registers, eventually resulting in reduced occupancy
- **Also increase the pressure on the cache in some kernels**

Cache Pressure

- Overworking memory cache with too many or wasteful memory access
- Cache line re-use happens when data from same cache line are frequently accessed
- Cache line re-use (on conventional serial architectures improves performance) is a potential problem for thread coarsening for a GPU.
- In GPU, cache pressure is, however, not an issue when data is accessed in a streaming manner, where there is no data re-use or cache line re-use.

Cache Pressure Due To Coarsening

- With coarsening, in the worst case, the same number of active threads per SM will be executing as in the original kernel, but with each thread doing more work.
- In this worst case, the number of cache lines being accessed at any time will thus scale with the coarsening factor increasing the pressure on the cache.
- Higher coarsening factors usually yield lower occupancy and result in fewer active threads, the effect is not always linear.
- However, in the presence of cache line re-use, the cache pressure frequently increases with the coarsening factor. This typically outweighs the benefits that coarsening might otherwise bring.

Simple Matrix Transpose

- Each thread reads a single element from source array and writes to target array at its transposed index
- Cache lines are read in coalesced fashion by threads of single warp
- But written to different cache lines in an uncoalesced way from multiple warps
- Data cached for read will not be accessed again
- Cache line for write ideally should be present till written by each warp (cache line re-use)

(Figures from reference[3])

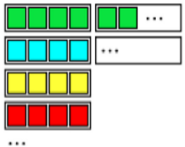


Figure: Coalesced Read

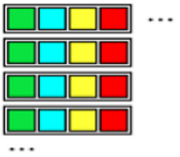


Figure: Uncoalesced Write

Simple Matrix Transpose

- As writes come from different warps, cache lines will not be evicted until the data is written by all the warps responsible for writing,
- For a coarsening factor of 2, twice as many instructions (in worst case) may be in flight at any time,
- Average time for completing writes to cache lines will increase.

Figure: Coalesced Read

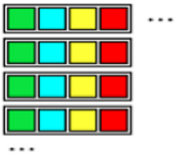


Figure: Uncoalesced Write

(Figures from reference[3])

Matrix Transpose Kernel (un-coarsened)

```
__global__ void matrixTranspose(__global float *output, __global const float*
    input, int width, int height) {

unsigned int row = blockIdx.y* blockDim.y+ threadIdx.y;
unsigned int column = blockIdx.x* blockDim.x+ threadIdx.x;

unsigned int indexIn  = row * width + column;
unsigned int indexOut = column * height + row;

output[indexOut] = input[indexIn];

}
```


Effects of Coarsening on Matrix Transpose Kernel

- Primarily depends on how threads are scheduled on SM and memory system
- If we coarsen by a factor 2, there will be two read-write statements, so twice as many read and write instructions are required
- So as we increase the coarsing factor, the number of cache line access will also increase resulting in cache pressure
- So, the average time taken to complete write to a given cache line increases leading to decreased performance

Effects of Coarsening on Matrix Transpose Kernel

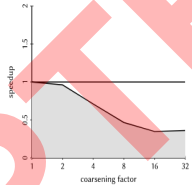


Figure: Effects of Coarsening on Matrix Transpose Kernel (Figure from refernece[3])

Reference

1. Wang, Guibin, YiSong Lin, and Wei Yi. "Kernel fusion: An effective method for better power efficiency on multithreaded GPU." 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing. IEEE, 2010.
2. Filipovič, Jiří, Matúš Madzin, Jan Fousek, and Luděk Matyska. "Optimizing CUDA code by kernel fusion: application on BLAS." The Journal of Supercomputing 71, no. 10 (2015): 3934-3957.
3. Stawinoga, Nicolai, and Tony Field. "Predictable Thread Coarsening." ACM Transactions on Architecture and Code Optimization (TACO) 15.2 (2018): 23.
4. Magni, Alberto, Christophe Dubach, and Michael O'Boyle. "Automatic optimization of thread-coarsening for graphics processors." Proceedings of the 23rd international conference on Parallel architectures and compilation. ACM, 2014.