EECS ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF MICHIGAN

A GPGPU Compiler for Memory Optimization and Parallelism Management

Group 5 - Allison Okimoto, Armaan Randhawa, and Danny Samuel

Background

High-Level Overview

How can we utilize memory optimization and parallelism in GPUs at the compiler level?

Motivation:

- GPU programming is hard
- GPU programmers spend lots of time on low-level optimizations
- GPU hardware is rapidly evolving - optimizing code for one GPU doesn't translate to future models

GPU Overview

Image Credit: Hwu and Kirk

GPU Memory Model

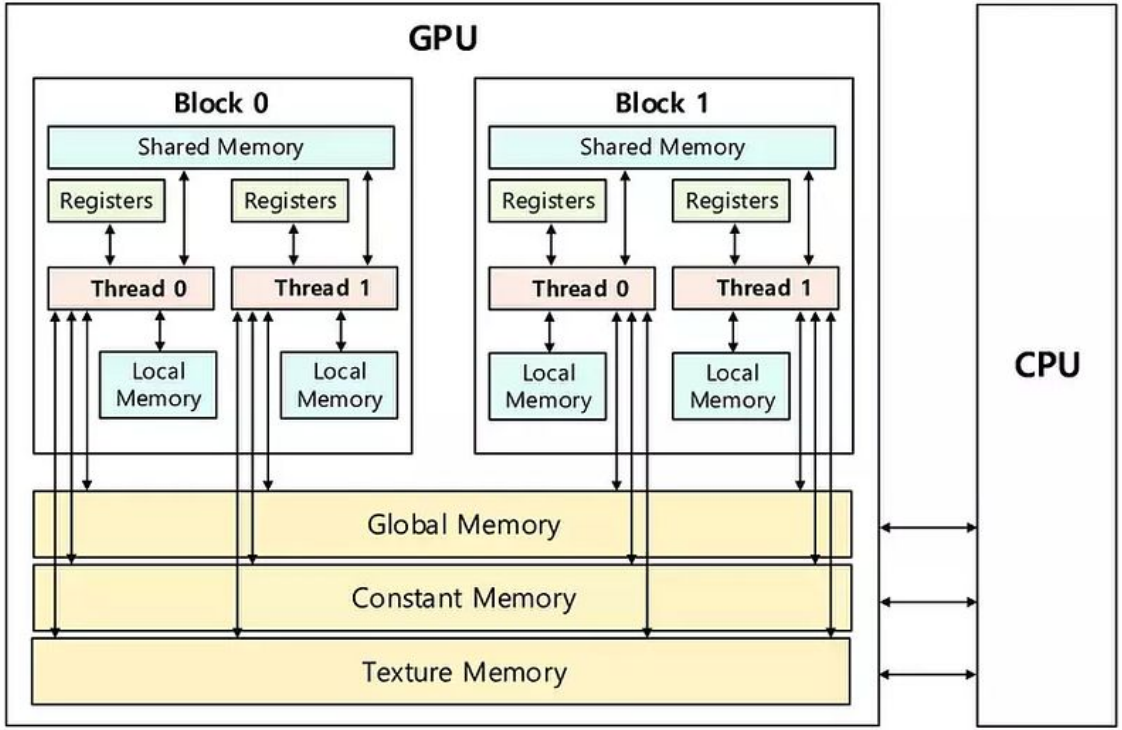


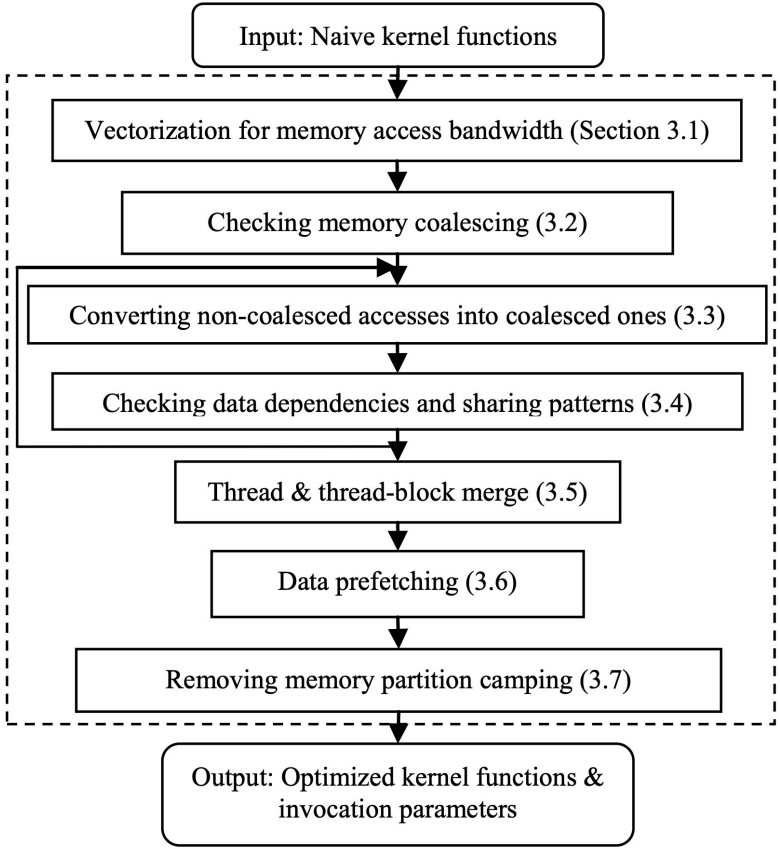
Image Credit: Parallel Implementations of ARIA on ARM Processors and Graphics Processing Unit

Example Kernel

```
float sum = 0;
for (int i=0; i<w; i++)
        sum+=a[idy][i]*b[i][idx];
c[idy][idx] = sum;
```

Optimizations

Optimization Steps



Memory Coalescing

- Global memory reads are high latency, high throughput
- If threads are accessing sequential memory, we can make one big memory request
- This is not possible if the reads are not aligned, or not sequential

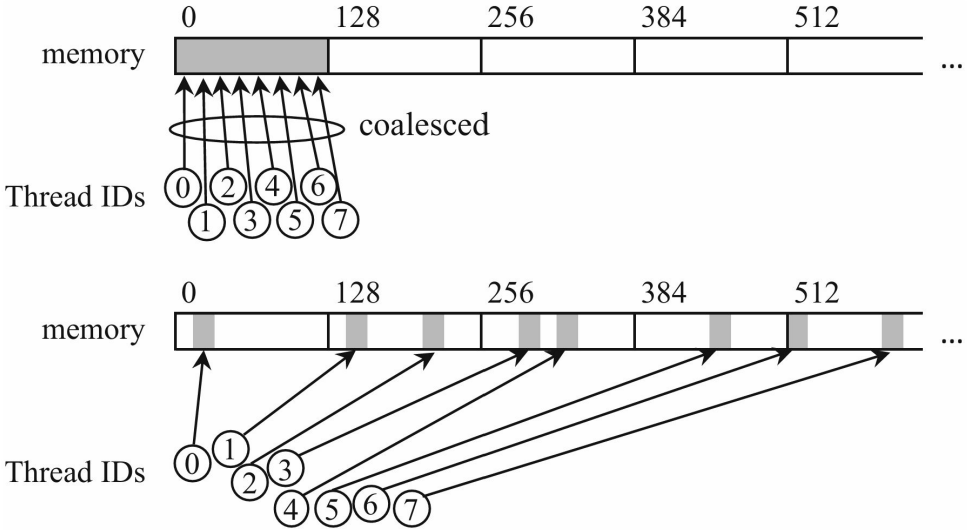


Image Credit: Real-Time FFT Computation Using GPGPU for OFDM-Based Systems

Detecting Uncoalesced Memory Accesses

1. Address computations

2. Compute strides per thread

3. Δaddress/ΔthreadId = data type size => coalesced access

Coalescing Memory Accesses through Shared Mem

- Shared memory is low latency
- Add a coalesced load from global to shared memory
- Replace global memory reads with shared memory reads

```
float sum = 0;
for (int i=0; i<w; i++)
        sum+=a[idy][i]*b[i][idx];
c[idy][idx] = sum;
```



```
(S0)for (i=0; i<w; i=(i+16)) {
(S1)   __shared__ float shared0[16];
(S2)   shared0[(0+tidx)]=a[idy][((i+tidx)+0)];
(S3)   __syncthreads();
(S4)   for (int k=0; k<16; k=(k+1)) {
(S5)     sum+=shared0[(0+k)]*b[(i+k)][idx]);
(S6)   }
(S7)   __syncthreads();
(S8)}
(S9)c[idy][idx] = sum;
```

Coalescing Memory Accesses - Other Techniques

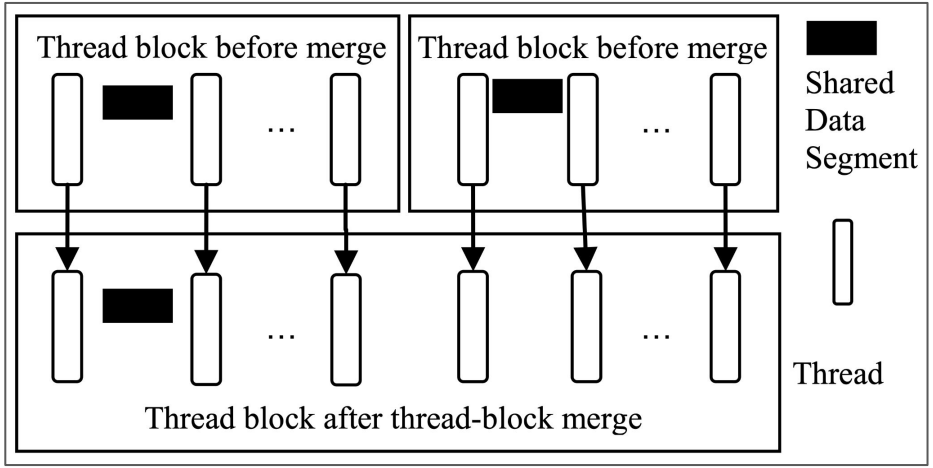
- Unroll loops
- Swap nested loops
- Change dimensional indexing

Data Dependencies and Data Sharing

- Compares coalesced segments
- Thread blocks or blocks with fixed stride
- Checks Memory Coalescing
  - Single Rule
  - Ranked

Thread/Thread Block Merge

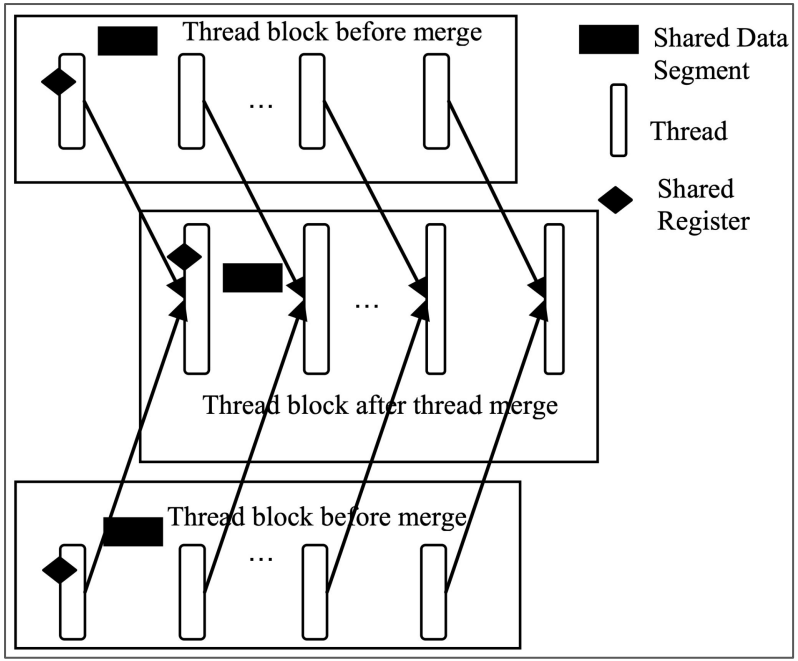
- Recompute thread ID: `tidx = idx%(N*blockDim.x)`
- Control Flow
- `blockDim.x = 2*blockDim.x`




```
int i = 0;
float sum = 0;
for (i=0; i<w; i=(i+16)) {
   __shared__ float shared0[16];
  if (tidx<16)  { /*inserted due to block
merge to remove redundant loads */
       shared0[(0+tidx)]=a[idy][((i+tidx)+0)];
  }
  __syncthreads();
  int k;
  for (k=0; k<16; k=(k+1)) {
      sum+=shared0[(0+k)]*b[(i+k)][idx]);
  }
  __syncthreads();
}
c[idy][idx] = sum;
```

Thread/Thread Block Merge

- Recompute thread ID: `idy*N, idy*N+1,…, idy*N+(N-1)`
- Computations




```
int i = 0;
float sum_0 = 0;
……
float sum_31 = 0;
for (i=0; i<w; i=(i+16)) {
    __shared__ float shared0_0[16];
    ……
    __shared__ float shared0_31[16];
    if (tidx<16)  {
      /* 32 is the number of the threads to
be merged */
        shared0_0[(0+tidx)]=
              a[idy*32+0][((i+tidx)+0)];
        ……
        shared0_31[(0+tidx)]=
              a[idy*32+31][((i+tidx)+0)];
      }
      syncthreads();
      int k;
      for (k=0; k<16; k=(k+1)) {
        float r0 = b[(i+k)][idx])
        sum_0+=shared0[(0+k)]*r0;
        ……
        sum_31+=shared0_31[0+k]*r0;
      }
      __syncthreads();
}
c[idy*32+0][idx] = sum_0;
……
c[idy*32+31][idx] = sum_31;
```

Data Prefetching

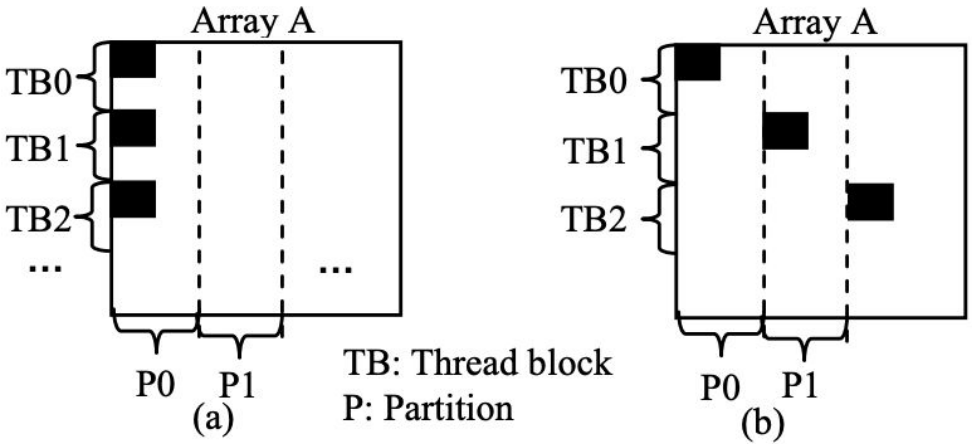
- Overlap memory access latency
- Analyzes loop
- Temp variable
- Data Reuse -> Skip

```
for (i=0; i<w; i=(i+16)){
    __shared__ float shared0[16];
   shared0[(0+tidx)]=a[idy][((i+tidx)+0)];
    __syncthreads();
    int k;
    for (k=0; k<16; k=(k+1))    {
        sum+=(shared0[(0+k)]*b[(i+k)][idx]);
    }
    __syncthreads();
}
```

```
/* temp variable */
float tmp = a[idy][((0+tidx)+0)];
for (i=0; i<w; i=(i+16)) {
    __shared__ float shared0[16];
   shared0[(0+tidx)]=tmp;
    __syncthreads();
    if (i+16<w) //bound check
        tmp = a[idy][(((i+16)+tidx)+0)];
    int k;
    for (k=0; k<16; k=(k+1))   {
        sum+=(shared0[(0+k)]*b[(i+k)][idx]);
    }
     __syncthreads();
}
```

Removing Partition Camping

- Global memory is split into partitions
  - Simultaneous access across thread blocks is slow
- Detect: when stride size is multiple of (partition size * #partitions)
- Mitigate: apply a fixed offset and change loop bounds



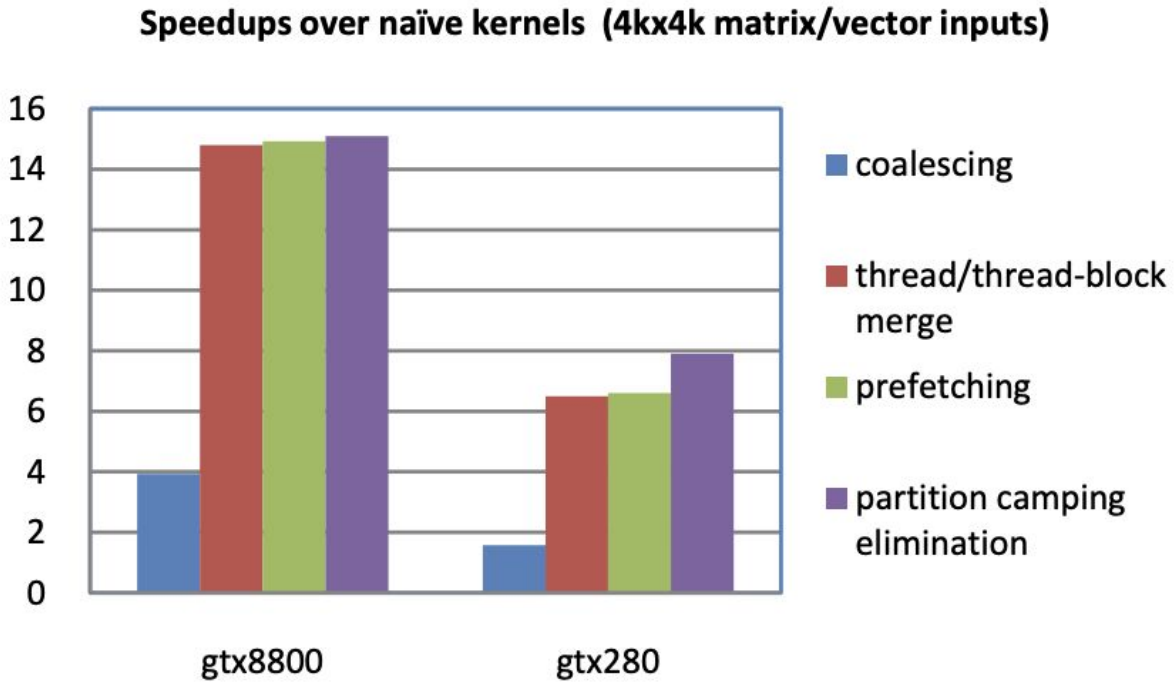
Results

- Test a variety of benchmarks to see their speedup
- Using GTX 8800 and GTX 280 (~2x memory, compute units, etc)

Algorithm	The size of input matrices/vectors	Num. of LOC in the naïve kernel
transpose matrix vector multiplication (tmv)	1kx1k to 4kx4k (1k to 4k vec.)	11
matrix mul. (mm)	1kx1k to 4kx4k	10
matrix-vector mul. (mv)	1kx1k to 4kx4k	11
vector-vector mul. (vv)	1k to 4k	3
reduction (rd)	1-16 million	9
matrix equation solver (strsm)	1kx1k to4kx4k	18
convolution (conv)	4kx4k image, 32x32 kernel	12
matrix transpose (tp)	1kx1k to 8kx8k	11
Reconstruct image (de-mosaicing)	1kx1k to 4kx4k	27
find the regional maxima (imregionmax)	1kx1k to 4kx4k	26

Table 1. A list of the algorithms optimized with our compiler.

Cumulative Effect of Techniques



Comparison to NVIDIA's Best

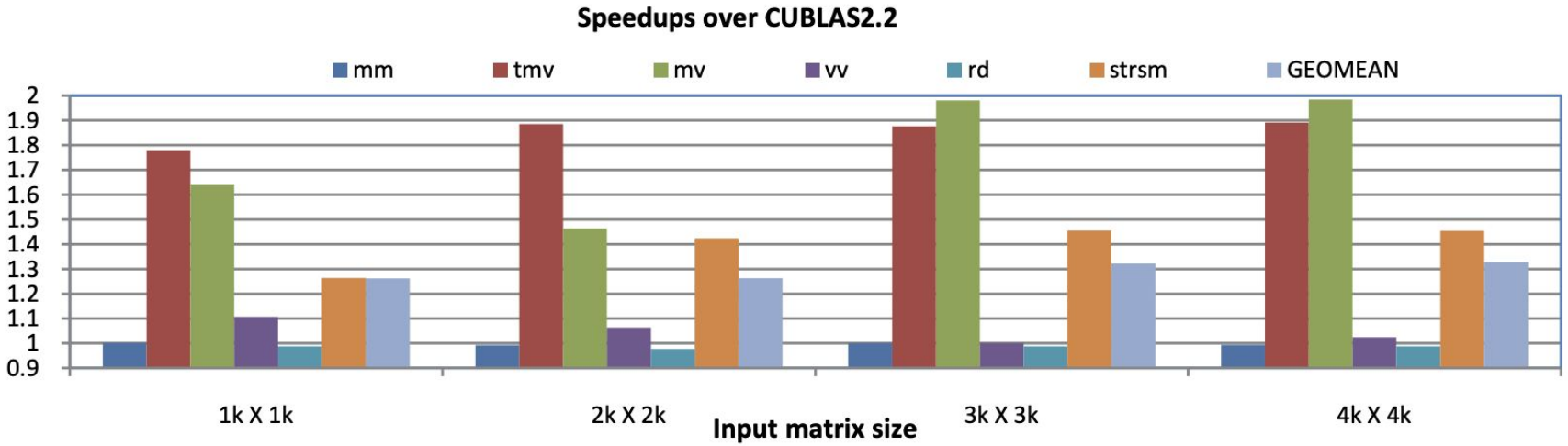


Figure 13. Performance improvement of our optimized kernels over CUBLAS 2.2 implementations on GTX280.

Group Commentary

- ✅ Good that it moves away from proprietary solutions (e.g. CUDA)
- ✅ Takes advantage of hardware structure/strengths
- ❌ They only performed comparisons against NVIDIA, not AMD/INTEL
- ❌ Can’t perform algorithm-level optimizations, still up to the developer
- ❌ Old paper