L4: Memory Hierarchy Optimization II, Locality and Data Placement

Overview of Lecture

• More on tiling for shared memory and constant memory
• Reading:
  – Chapter 5, Kirk and Hwu book
  – Or, http://courses.ece.illinois.edu/ece498/al/textbook/Chapter4-CudaMemoryModel.pdf

Review: Targets of Memory Hierarchy Optimizations

• Reduce memory latency
  – The latency of a memory access is the time (usually in cycles) between a memory request and its completion
• Maximize memory bandwidth
  – Bandwidth is the amount of useful data that can be retrieved over a time interval
• Manage overhead
  – Cost of performing optimization (e.g., copying) should be less than anticipated gain

Administrative

• Assignment due Friday, Jan. 18, 5 PM
  – Use handin program on CADE machines
    • handin CS6235 lab1 <probfile>
• Mailing list
  – CS6235@list.eng.utah.edu
  • Please use for all questions suitable for the whole class
  • Feel free to answer your classmates questions!
Hardware Implementation: Memory Architecture

- The local, global, constant, and texture spaces are regions of device memory (DRAM)
- Each multiprocessor has:
  - A set of 32-bit registers per processor
  - On-chip shared memory
    - Where the shared memory space resides
  - A read-only constant cache
    - To speed up access to the constant memory space
  - A read-only texture cache
    - To speed up access to the texture memory space
  - Data cache (Fermi only)

Review: Reuse and Locality

- Consider how data is accessed
  - **Data reuse**:
    - Some data used multiple times
    - Intrinsic in computation
  - **Data locality**:
    - Data is reused and present in "fast memory"
    - Same data or same data transfer
- If a computation has reuse, what can we do to get locality?
  - Appropriate data placement and layout
  - Code reordering transformations

Memory Hierarchy Example: Matrix vector multiply

```c
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        a[i] += c[j][i] * b[j];
    }
}
```

Remember to:
- Consider correctness of parallelization strategy (next week)
- Exploit locality in shared memory and registers

Resulting CUDA code (Automatically Generated by our Research Compiler)

```c
__global__ mv_GPU(float* a, float* b, float** c) {
    int bx = blockIdx.x; int tx = threadIdx.x;
    __shared__ float bcpy[32];
    double acpy = a[tx + 32 * bx];
    for (k = 0; k < 32; k++) {
        bcpy[tx] = b[32 * k + tx];
        __syncthreads();
        for (j = 32 * k; j <= 32 * k + 32; j++) {
            acpy = acpy + c[j][32 * bx + tx] * bcpy[j];
        }
        __syncthreads();
    }
    a[tx + 32 * bx] = acpy;
}
```
Ch. 4: Matrix Multiplication
A Simple Host Version in C

/* Matrix multiplication on the CPU: host in double precision */
void MatrixMulOnHost(float* M, float* N, float* P, int Width) {
    for (int i = 0; i < Width; ++i) {
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
    }
}

Discussion (Simplified Code)

for (int i = 0; i < Width; ++i) {
    for (int j = 0; j < Width; ++j) {
        double sum = 0;
        for (int k = 0; k < Width; ++k) {
            double a = M[i][k];
            double b = N[k][j];
            sum += a * b;
        }
        P[i][j] = sum;
    }
}

Let's Look at This Code

for (int ii = 0; ii < Width; ii+=TI) {
    for (int i=ii; i<ii+TI; i++) {
        for (int jj=0; jj<Width; jj+=TJ) {
            for (int j = jj; j < jj+TJ; j++) {
                double sum = 0;
                for (int kk = 0; kk < Width; kk+=TK) {
                    for (int k = kk; k < kk+TK; k++) {
                        sum += M[i][k] * N[k][j];
                    }
                }
                P[i][j] = sum;
            }
        }
    }
}

Strip-Mined Code

for (int ii = 0; ii < Width; ii+=TI) {
    for (int i=ii; i<ii+TI; i++) {
        for (int jj=0; jj<Width; jj+=TJ) {
            for (int j = jj; j < jj+TJ; j++) {
                double sum = 0;
                for (int kk = 0; kk < Width; kk+=TK) {
                    for (int k = kk; k < kk+TK; k++) {
                        sum += M[i][k] * N[k][j];
                    }
                }
                P[i][j] = sum;
            }
        }
    }
}
But this code doesn’t match CUDA Constraints

for (int ii = 0; ii < Width; ii+=TI)  
  for (int j = ii; j < ii+TI; j++)  
    for (int jj=0; jj<Width; jj+=TJ)  
      for (int j = jj; j < jj+TJ; j++) {  
        double sum = 0;  
        for (int kk = 0; kk < Width; kk+=TK) {  
          for (int k = kk; k < kk+TK; k++)  
            sum += M[i][k] * N[k][j];  
        }  
        P[i][j] = sum;  
      }

Can we fix this?

Unit Stride Tiling – Reflect Stride in Subscript Expressions

for (int ii = 0; ii < Width/TI; ii++)  
  for (int i=0; i<TI; i++)  
    for (int jj=0; jj<Width/TJ; jj++)  
      for (int j = 0; j < TJ; j++) {  
        double sum = 0;  
        for (int kk = 0; kk < Width; kk+=TK) {  
          for (int k = kk; k < kk+TK; k++)  
            sum += M[ii*TI+i][k] * N[k][jj*TJ+j];  
        }  
        P[ii*TI+i][jj*TJ+j] = sum;  
      }

What Does this Look Like in CUDA

#define TI 32  
#define TJ 32  
dim3 dimGrid(Width/TI, Width/TJ);  
dim3 dimBlock(TI,TJ);  
matMult<<<dimGrid,dimBlock>>>(M,N,P);  
__global__ matMult(float *M, float *N, float *P) {  
ii = blockIdx.y;  
jj = blockIdx.x;  
i = threadIdx.y;  
j = threadIdx.x;  
double sum = 0;  
for (int k = kk; k < kk+TK; k++) {  
  sum += M[ii*TI+i][k] * N[k][jj*TJ+j];  
}  
__syncthreads();  
P[ii*TI+i][jj*TJ+j] = sum;  
}

What Does this Look Like in CUDA

#define TI 32  
#define TJ 32  
dim3 dimGrid(Width/TI, Width/TJ);  
dim3 dimBlock(TI,TJ);  
matMult<<<dimGrid,dimBlock>>>(M,N,P);  
__global__ matMult(float *M, float *N, float *P) {  
ii = blockIdx.y;  
jj = blockIdx.x;  
i = threadIdx.y;  
j = threadIdx.x;  
double sum = 0;  
for (int k = kk; k < kk+TK; k++) {  
  Mds[j][k] = M[ii*TI+i][k];  
  Nds[j][k] = N[k][jj*TJ+j];  
}  
__syncthreads();  
P[ii*TI+i][jj*TJ+j] = sum;  
}
Derivation of code in text

- \( T1 = TJ = TK = "TILE_WIDTH" \)
- All matrices square, Width \( \times \) Width
- Copies of matrix \( M \) and \( N \) in shared memory
  - "Linearized" 2-d array accesses:
    - \( a[i][j] \) is equivalent to \( a[i \times \text{Row} + j] \)
- Each SM computes a "tile" of output matrix \( P \) from a block of consecutive rows of \( M \) and a block of consecutive columns of \( N \)
  - dim3 Grid (\( \text{Width}/\text{TILE_WIDTH}, \text{Width}/\text{TILE_WIDTH} \))
  - dim3 Block (TILE_WIDTH, TILE_WIDTH)
  - Location \( P[i][j] \) corresponds to \( P[(i \times \text{TILE_WIDTH} + i) \times \text{TILE_WIDTH} + j] \) or \( P[\text{Row}][\text{Col}] \)

Final Code (from text, p. 87)

```c
__global__ void MatrixMulKernel(float *Md, float *Nd, float *Pd, int Width) {
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    // Identify the row and column of the Pd element to work on
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    float Pvalue = 0;
    for (int m = 0; m < Width / TILE_WIDTH; ++m) {
        // Collaborative (parallel) loading of Md and Nd tiles into shared memory
        Mds[ty][tx] = Md[Row * Width + (m * TILE_WIDTH + tx)];
        Nds[ty][tx] = Nd[(m * TILE_WIDTH + ty) * Width + Col];
        __syncthreads(); // make sure all threads have completed copy before calculation
        for (int k = 0; k < TILE_WIDTH; ++k) {
            // Update Pvalue for TKxTK tiles in Mds and Nds
            Pvalue += Mds[ty][k] * Nds[k][tx];
        }
        __syncthreads(); // make sure calculation complete before copying next tile
    }
    Pd[Row * Width + Col] = Pvalue;
}
```

Summary of Lecture

- Matrix-matrix multiply example