

CUDA GPU Development

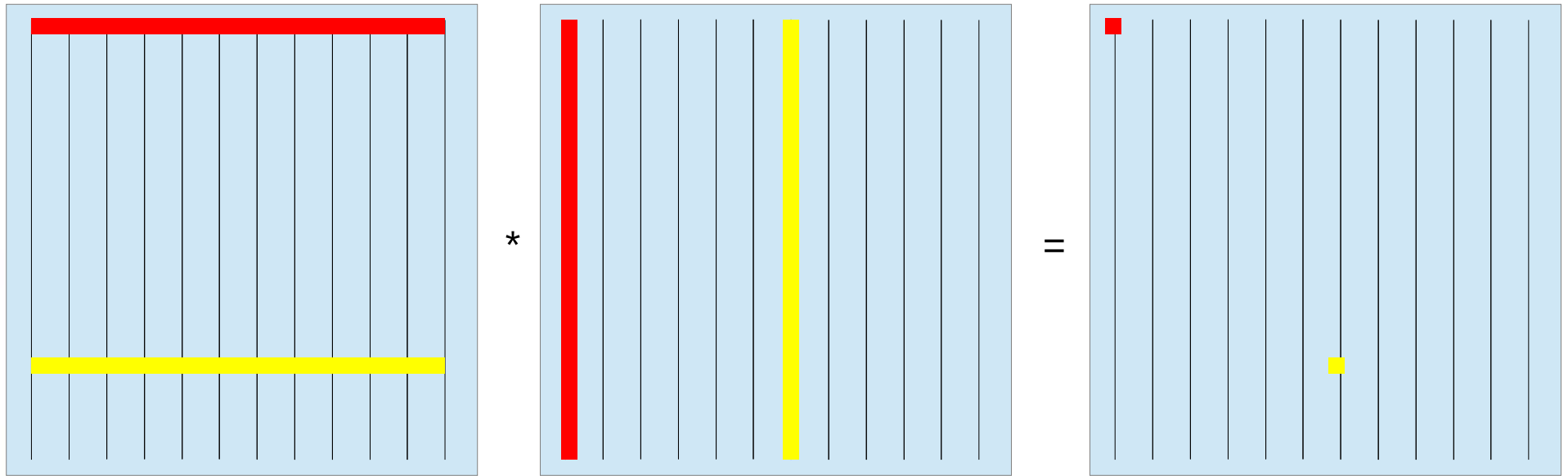
A simple dense Matrix-Multiply example

by Clifford Wolf, April 2013

Example: Multiply-Add dense Matrices

- We are going to look at code for:
 - A simple Matrix abstraction class
 - A CPU implementation of $Y += A*B$
 - A simple GPU implementation
 - An optimized GPU implementation
- We are going to profile all three implementations

Matrix Multiplication



- Memory Layout: Column-major order
- N^3 Multiplications and Additions

Simple CUDA-aware Matrix Class

```
struct Matrix_t
{
    int m, n;
    float *host_data, *device_data; // column-major order

    Matrix_t(int m, int n) : m(m), n(n),
        host_data(NULL), device_data(NULL) { }

    __device__ __host__ float &operator()(int i, int j) {
#ifdef __CUDA_ARCH__
        return device_data[j*m + i];
#else
        allocHostData();
        return host_data[j*m + i];
#endif
    }

    ...
};
```

Matrix Class: Allocate and Free Memory

```
void Matrix_t::allocHostData() {
    if (host_data == NULL)
        CUDA_CHECK_RETURN(cudaMallocHost((void**)&host_data,
                                           sizeof(float)*m*n));
}

void Matrix_t::allocDeviceData() {
    if (device_data == NULL)
        CUDA_CHECK_RETURN(cudaMalloc((void**)&device_data,
                                      sizeof(float)*m*n));
}

void Matrix_t::free() {
    if (host_data != NULL) {
        CUDA_CHECK_RETURN(cudaFreeHost(host_data));
        host_data = NULL;
    }
    if (device_data != NULL) {
        CUDA_CHECK_RETURN(cudaFree(device_data));
        device_data = NULL;
    }
}
```

Matrix Class: Zero-Initialize

```
void Matrix_t::zeroHostData() {
    allocHostData();
    memset((void*)host_data, 0, sizeof(float)*m*n);
}

void Matrix_t::zeroDeviceData() {
    allocDeviceData();
    CUDA_CHECK_RETURN(cudaMemset((void*)device_data, 0,
                                sizeof(float)*m*n));
}
```

Matrix Class: Copy Host \leftrightarrow Device

```
void Matrix_t::copyToDevice() {  
    allocDeviceData();  
    CUDA_CHECK_RETURN(cudaMemcpy(device_data, host_data,  
                                sizeof(float)*m*n, cudaMemcpyHostToDevice));  
}  
  
void Matrix_t::copyFromDevice() {  
    allocHostData();  
    CUDA_CHECK_RETURN(cudaMemcpy(host_data, device_data,  
                                sizeof(float)*m*n, cudaMemcpyDeviceToHost));  
}
```

CPU Implementation

```
void matrixmuladd_omp(Matrix_t &matA, Matrix_t &matB,  
                      Matrix_t &matY)  
{  
    #pragma omp parallel for  
    for (int i = 0; i < matY.m; i++)  
    for (int j = 0; j < matY.n; j++) {  
        float sum = 0;  
        for (int k = 0; k < matA.n; k++)  
            sum += matA(i, k) * matB(k, j);  
        matY(i, j) += sum;  
    }  
}
```


Simple GPU Driver

```
void matrixmuladd_simple(Matrix_t &matA, Matrix_t &matB,  
    Matrix_t &matY)  
{  
    int sqrtThreadsPerBlock = sqrt(256);  
    dim3 gridDim, blockDim;  
  
    blockDim.x = std::min(matY.m, sqrtThreadsPerBlock);  
    blockDim.y = std::min(matY.n, sqrtThreadsPerBlock);  
    blockDim.z = 1;  
  
    gridDim.x = (matY.m + blockDim.x - 1) / blockDim.x;  
    gridDim.y = (matY.n + blockDim.y - 1) / blockDim.y;  
    gridDim.z = 1;  
  
    matrixmuladd_simple_kernel<<<gridDim, blockDim>>>(matA,  
        matB, matY);  
}
```

Simple GPU Kernel

```
// calculate matY using one thread per result element
__global__ void matrixmuladd_simple_kernel(Matrix_t matA,
      Matrix_t matB, Matrix_t matY)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    int j = threadIdx.y + blockIdx.y*blockDim.y;

    if (i < matY.m && j < matY.n) {
        float sum = 0;
        for (int k = 0; k < matA.n; k++)
            sum += matA(i, k) * matB(k, j);
        matY(i, j) += sum;
    }
}
```

Better GPU Driver

```
void matrixmuladd_better(Matrix_t &matA, Matrix_t &matB,  
    Matrix_t &matY)  
{  
    dim3 gridDim, blockDim;  
  
    blockDim.x = 32;  
    blockDim.y = 32;  
    blockDim.z = 1;  
  
    gridDim.x = (matY.m + 31) / 32;  
    gridDim.y = (matY.n + 31) / 32;  
    gridDim.z = 1;  
  
    matrixmuladd_better_kernel<<<gridDim, blockDim>>>(matA,  
        matB, matY);  
}
```

Better GPU Kernel (1/2)

```
// calculate a 32x32 square of matY using 32x32 threads
__global__ void __launch_bounds__(32*32)
    matrixmuladd_better_kernel(Matrix_t matA,
                                Matrix_t matB, Matrix_t matY)
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    int i_off = blockIdx.x*32;
    int j_off = blockIdx.y*32;
    float sum = 0;

    // add one byte padding in last dimension to avoid shared
    // memory access pattern with a stride of 128 bytes (32 words)
    __shared__ float windowA[32][33];
    __shared__ float windowB[32][33];

    float nextA = matA(i_off + i, j);
    float nextB = matB(i, j_off + j);

    ...
}
```

Better GPU Kernel (2/2)

...

```
for (int k = 0; k < matA.n; k += 32)
{
    windowA[i][j] = nextA;
    windowB[i][j] = nextB;
    __syncthreads();

    nextA = i_off + i < matA.m && k + j + 32 < matA.n ?
            matA(i_off + i, k + j + 32) : 0;
    nextB = k + i + 32 < matB.m && j_off + j < matB.n ?
            matB(k + i + 32, j_off + j) : 0;

    #pragma unroll
    for (int l = 0; l < 32; l++)
        sum += windowA[i][l] * windowB[l][j];
    __syncthreads();
}

if (i_off + i < matY.m && j_off + j < matY.n)
    matY(i_off + i, j_off + j) += sum;
}
```

Using CUBLAS

```
void matrixmuladd_cublas(Matrix_t &matA, Matrix_t &matB,  
    Matrix_t &matY)  
{  
    cublasHandle_t handle;  
    CUBLAS_CHECK_RETURN(cublasCreate(&handle));  
  
    float alpha = 1, beta = 1;  
    CUBLAS_CHECK_RETURN(cublasSgemm(handle,  
        CUBLAS_OP_N, CUBLAS_OP_N,  
        matY.m, matY.n, matA.n,  
        &alpha, matA.device_data, matA.m,  
        matB.device_data, matB.m,  
        &beta, matY.device_data, matY.m));  
  
    CUBLAS_CHECK_RETURN(cublasDestroy(handle));  
}
```

Building with nvcc

```
nvcc -O3 -Xcompiler -fopenmp,-march=native \
    -gencode arch=compute_20,code=compute_20 \
    -gencode arch=compute_20,code=sm_21 \
    -o matrixmul matrixmul.cu
```

- Generates PTX (JIT) code for Fermi:

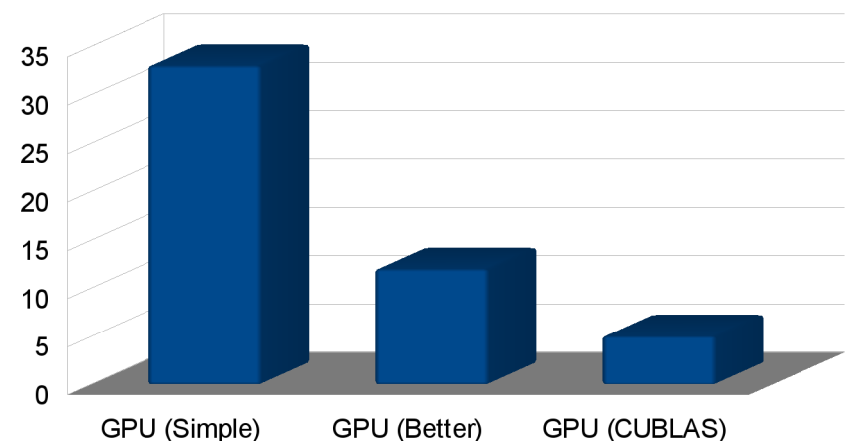
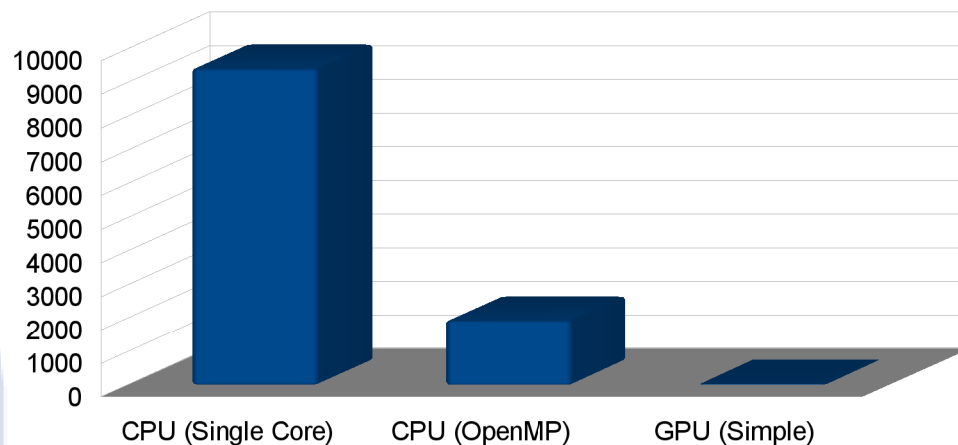
```
-gencode arch=compute_20,code=compute_20
```

- Generates pre-compiled device code for Fermi:

```
-gencode arch=compute_20,code=sm_21
```

Profiling Results

- Problem size: squaring a 1024x1024 matrix
- CPU: AMD FX-8150, 8 cores, 4187 MHz
- GPU: GeForce GTX 560 Ti, 384 cores, 1721 MHz
- GPU vs. Single Core: 285x (simple) or 784x (better)
- GPU vs. OpenMP: 58x (simple) or 160x (better)



Questions?