Lecture 10

CUDA (II)

Review

- host ← → device
- CUDA – many core processor
- threads → thread blocks → grid
  - # threads >> # of cores to be efficient
  - Threads within blocks can cooperate
  - Threads between thread blocks cannot cooperate
  - Kernels are executed by threads
- CUDA-C API
  - Memory management:cudaMalloc, cudaFree, cudaMemset, cudaMemcpy

Review

- General pattern of using CUDA
  - Copy data to be computed onto device memory
  - Execute kernels in a specific thread configuration
  - Copy computed data from the device memory back to the host memory
- Threads determine their portions of work by using
  - threadIdx
  - blockIdx
  - blockDim

Today's Outline

- Matrix addition
  - Pinned memory
  - 2D memory layouts
- Vector scaling
  - Effect of pinned memory
  - Various optimization techniques
- Simple Moving Average
  - Arithmetic Intensity
  - Use of texture memory

03?.cu

Matrix addition

03b.cu

```
#define DT float
int main(int argc, char **argv) {
    const int n = 10;
    DT *A = genMatrix(n);
    DT *B = genMatrix(n);
    DT *C;
    cudaMallocHost(&C, n*n*sizeof(DT));

    GPUADD_INT(n, C, A, B);

    cudaFree(A);
    cudaFree(B);
    cudaFree(C);
    return 0;
}
```

```
DT *genMatrix(const int n) {
  DT *ptr;
  cudaMallocHost(&ptr, n*n*sizeof(DT));
  for(int i=0;i<n*n;i++) {
      ptr[i]=((double)rand())/RAND_MAX;
  }
  return ptr;
}
```

03b.cu

```
__global__
void matrixAdd(
    const int n, DT *ptrC, DT *ptrA, DT *ptrB) {

    const int i = blockIdx.y * blockDim.y + threadIdx.y;
    const int j = blockIdx.x * blockDim.x + threadIdx.x;

    if(i<n && j<n) {
        const int where = i*n+j;
        ptrC[where] = ptrA[where] + ptrB[where];
    }
}
```

7

03b.cu

```
void GPUADD_INT(const int n, DT *C, DT *A, DT *B) {
    DT *ptrA, *ptrB, *ptrC;
    const int size = sizeof(DT)*n*n;
    cudaMalloc(&ptrA, size);
    cudaMalloc(&ptrB, size);
    cudaMalloc(&ptrC, size);
    cudaMemcpy(ptrA, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(ptrB, B, size, cudaMemcpyHostToDevice);
    dim3 blocks(UNIT_X, UNIT_Y);
    dim3 grids((n+UNIT_X-1)/UNIT_X,(n+UNIT_Y-1)/UNIT_Y));
    matrixAdd <<< grids, blocks >>> (n, ptrC, ptrA, ptrB);
    cudaMemcpy(C, ptrC, size, cudaMemcpyDeviceToHost);
    cudaFree(ptrA);
    cudaFree(ptrB);
    cudaFree(ptrC);
}
```

8

03c.cu

Matrix addition with 2D memory allocation

9

CUDA-C

APIs for memory management

- cudaError_t cudaMallocPitch (void **devPtr, size_t *pitch, size_t width, size_t height)
  - *Allocates at least width in Bytes & height bytes of linear memory on the device and returns in devPtr a pointer to the allocated memory. The function may* **pad** *the allocation to ensure that corresponding pointers in any given row will continue to meet the* ***alignment requirements for coalescing*** *as the address is updated from row to row. The pitch returned in pitch by cudaMallocPitch() is the* ***width in bytes*** *of the allocation*
  - *T* pElement = (T*)((char*)BaseAddress + Row * pitch) + Column;*
  - Pitch is similar to leading dimensions in BLAS discussed.
- cudaError_t cudaMemcpy2D (void *devPtr, size_t dpitch, const void *src, size_t spitch, size_t width, size_t height, enum cudaMemcpyKind kind)

10

CUDA Reference manual: Section 4.8: memory management

03c.cu

Matrix addition with 2D layout

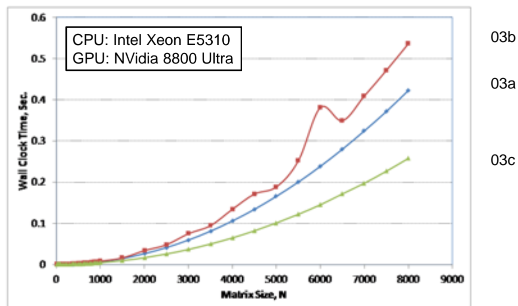
```
__global__ void matrixAdd( const int n, const int pitch,
        DT *ptrC, DT *ptrA, DT *ptrB)  {

    const int i = blockIdx.y * blockDim.y + threadIdx.y;
    const int j = blockIdx.x * blockDim.x + threadIdx.x;

    if(i<n && j<n) {
            DT *C = (DT*) ((char*)ptrC + i*pitch) + j;
            DT *A = (DT*) ((char*)ptrA + i*pitch) + j;
            DT *B = (DT*) ((char*)ptrB + i*pitch) + j;
            C[0] = A[0] + B[0];
    }
}
```

11

```
void GPUADD_INT2(const int n, DT *C, DT *A, DT *B) {
DT *ptrA, *ptrB, *ptrC;
size_t pitch;
size_t size = n * sizeof(DT);

cudaMallocPitch((void**) &ptrA, &pitch, size, n) ;
cudaMallocPitch((void**) &ptrB, &pitch, size, n) ;
cudaMallocPitch((void**) &ptrC, &pitch, size, n) ;
cudaMemcpy2D(ptrA,pitch,A,size,size,n,cudaMemcpyHostToDevice);
cudaMemcpy2D(ptrB,pitch,B,size,size,n,cudaMemcpyHostToDevice);

dim3 blocks(UNIT_X, UNIT_Y);
dim3 grids((n+UNIT_X-1)/UNIT_X,(n+UNIT_Y-1)/UNIT_Y);
matrixAdd <<< grids, blocks >>> (n, pitch, ptrC, ptrA, ptrB);

cudaMemcpy2D(C,size,ptrC,pitch,size,n,cudaMemcpyDeviceToHost);
cudaFree(ptrA);
cudaFree(ptrB);
cudaFree(ptrC);
}
```

12

Performance



Lesson Learned

- When dealing with 2D data (matrices, 2D arrays), use 2D memory allocation to ensure proper alignment of data.
- cudaMallocPitch, cudaMemcpy2D
- The pitch parameter in CUDA API is similar to leading dimensions in BLAS, except that pitch uses "bytes" while leading dimensions use # of elements as its unit.

04?.cu

Vector scaling

(1, 2, 3) * 2 → (2,4,6)

Example: vector scaling

Vanilla C code vs. CUDA Kernel function

```
void dscal    (const float scale, const int n, float *v)
{
  int i;
  for(i=0;i<n;i++) {
    v[i] *= scale;
  }
}
```

```
#define THREAD_PER_BLOCK 512

__global__
void dscal_dev(const float scale, const int n, float *v)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i < n) v[i] *= scale;
}
```

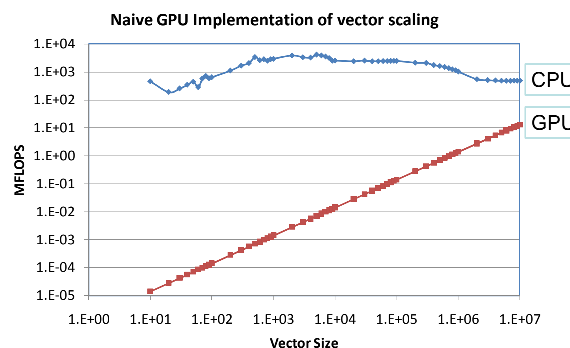
Example: vector scaling

CUDA code - interface

```
void dscal_interface(const float scale, const int n, float *v) {
  float *v_d;
  const int bytes = n * sizeof(float);

1 cudaMalloc((void**) &v_d, bytes);
  cudaMemcpy((void*)v_d,(void*)v,bytes,cudaMemcpyHostToDevice);

  dim3 blocks(THREAD_PER_BLOCK);
2 dim3 grids ((n+THREAD_PER_BLOCK-1) / THREAD_PER_BLOCK);
  dscal_device <<<grids, blocks>>> (scale, n, v_d);

3 cudaMemcpy((void*)v,(void*) v_d,bytes,cudaMemcpyDeviceToHost);
  cudaFree(v_d);
}
```

1. Allocate memory on device and copy data from host memory to device memory.
2. Divide work into a) grids of blocks and b) blocks of threads, and then execute the kern
3. Copy computed results from device memory to host memory.

Preliminary Results

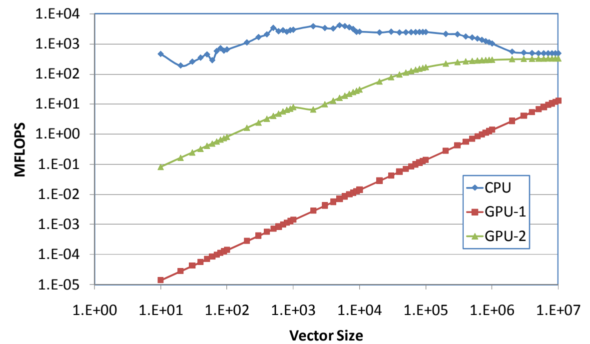


Example: vector scaling

Optimization #1 – use pinned memory

```
float *vec;
vec = (float*) malloc(sizeof(float) * size);
assert(vec);

for(i=0;i<size;i++) {
  vec[i] = 1.0f;
}
. . .
free(vec);
```

```
float *vec;
cudaMallocHost((void**) &vec, sizeof(float)*size);
assert(vec);

for(i=0;i<size;i++) {
  vec[i] = 1.0f;
}
. . .
cudaFree(vec);
```

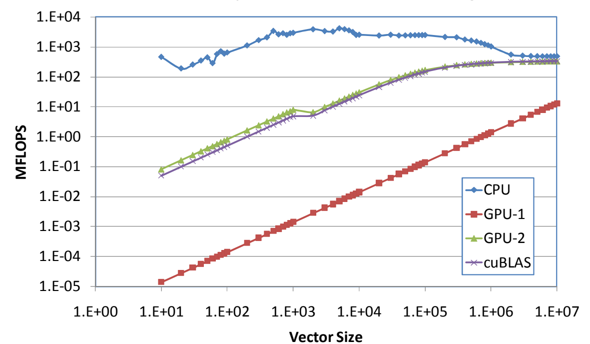


Example: vector scaling

Optimization #2? – use cuBLAS

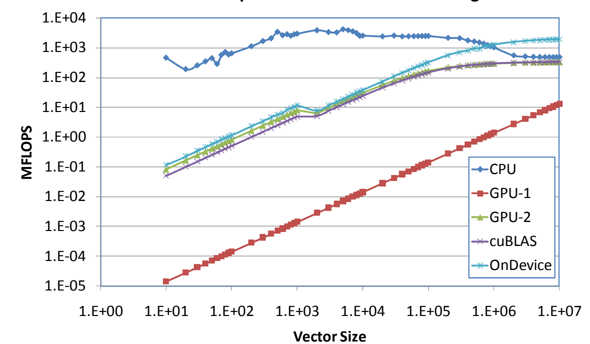
```
void dscal_cublas(const float scale,const int size,float *vec)
{
  cublasStatus stat;
  float *vec_d;
  cublasInit();

  stat = cublasAlloc(size, sizeof(float), (void**)&vec_d);
  cublasSetVector(size, sizeof(float), vec, 1, vec_d, 1);

  cublasSscal(size, scale, vec_d, 1);

  cublasGetVector(size, sizeof(float), vec_d, 1, vec, 1);
  cublasFree(vec_d);
}
```

BLAS stands for Basic Linear Algebra Subprograms: a library specification for performing vector, matrix-vector, matrix-matrix operations.



Where is the performance of GPU?

```
__global__
void dscal_device(const float scale, const int n, float *v) {
  int where = blockIdx.x * blockIdx.x + threadIdx.x;
  if(where < n) {
    v[where] = 1.0f;
    v[where] *= scale;
  }
}

void dscal_interface(const float scale, const int n, float *v) {
  float *v_d;

  cudaMalloc((void**) &v_d, n*sizeof(float));

  dim3 blocks(THREAD_PER_BLOCK);
  dim3 grids ((n+THREAD_PER_BLOCK-1) / THREAD_PER_BLOCK);
  dscal_device<<<grids, blocks>>> (scale, n, v_d);

  cudaMemcpyDeviceToHost);
  cudaFree(v_d);
}
```



What have we learned so far?

- GPGPU using CUDA is easy if you know C/C++.
  - Loops are replaced by "configuration" of threads
- Copying data between host and device kills performance
- Use of pinned (page-locked) memory boosts performance
  - Data in page-locked memory are transferred using DMA
- Do everything on device whenever possible
- Need to compute lots of data to show performance.
- GPU programs do NOT necessarily run faster than CPU.

05?.cu

Simple Moving Average

26

Simple Moving Average

- Data: 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2
- Window size: 2
  - 1, 1.5, 1.5, 1.5,1.5, 1.5, …
- Windows size: 3
  - 1, 2, 4/3, 5/3, 4/3, 5/3, 4/3, 5/3, …
- Window size: 4
  - 1, 2, 1.5, 1.5, 1.5, 1.5, 1.5, …
- Most numbers in the series is referenced as many times as the window size!

Example: moving average

Vanilla C code vs. CUDA Kernel

```
void
movingAverage(const int n,
  const float *v, const int ws,
  float *out)
{
  int i, j;

  float sum;

  for(i=0;i<ws-1;i++) {
    out[i]= v[i];
  }
  for(;i<n;i++) {
    sum = 0;
    for(j=0;j<ws;j++) {
      sum += v[i-j];
    }
    out[i] = sum/ws;
  }
}
```

```
__global__
void mavg_device(const float n,
  const float *v, const int ws,
  float *out)
{
  int i=blockIdx.x*blockIdx.x
        + threadIdx.x, j;
  float sum;

  if(i < ws-1) {
    out[i] = v[i];
  }
  else if(i < n) {
    sum = 0;
    for(j=0;j<ws;j++) {
      sum += v[i-j];
    }
    out[i] = sum/ws;
  }
}
```

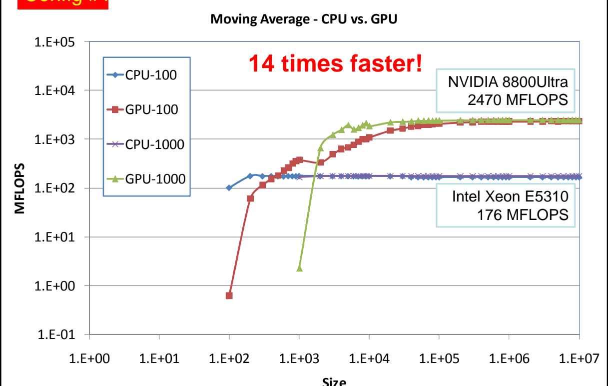
Example: moving average

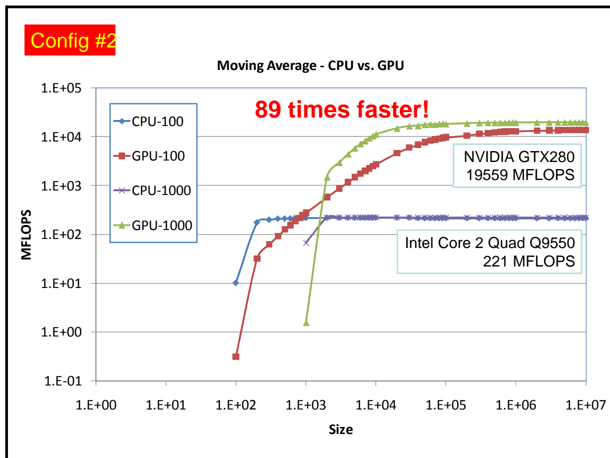
CUDA code - interface

```
void movingAverage_int(const int n, const float *v, const int ws,
        float *out)
{
  float *v_d, *out_d;
  int bytes = n*sizeof(float);

  cudaMalloc((void**) &v_d, bytes);
  cudaMalloc((void**) &out_d, bytes);
  cudaMemcpy(v_d, v, n*sizeof(float), cudaMemcpyHostToDevice);

  dim3 blocks(THREAD_PER_BLOCK);
  dim3 grids (n/THREAD_PER_BLOCK+1);
  mavg_device<<<grids, blocks>>> (n, v_d, ws, out_d);

  cudaMemcpy((void*)out,(void*)out_d,bytes,cudaMemcpyDeviceToHost);
  cudaFree(v_d);
  cudaFree(out_d);
}
```

Both *v and *out are allocated using cudaMallocHost

Config #1





Example: moving average

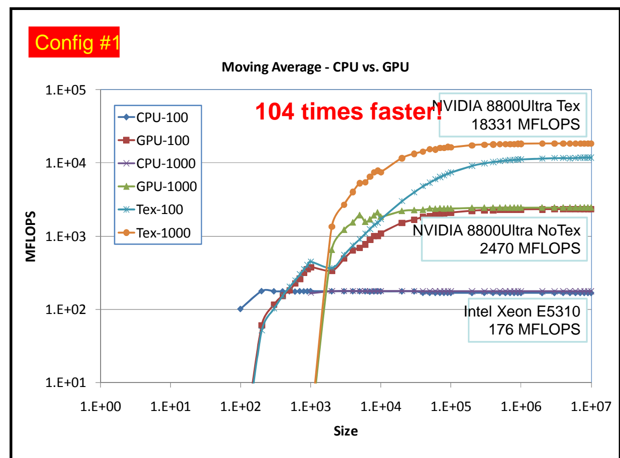
Optimization #2 – use texture memory

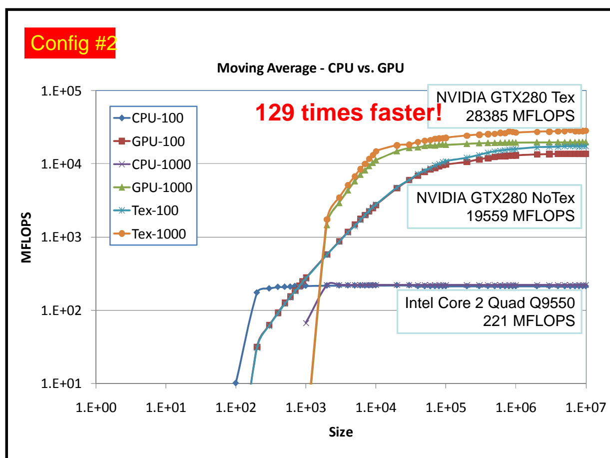
```
void movingAverage_int(const int n, const float *v, const int ws,
       float *out)
{
  float *v_d, *out_d;
  int bytes = n * sizeof(float);

  cudaMalloc((void**) &v_d  , bytes);
  cudaMalloc((void**) &out_d, bytes);
  cudaMemcpy(v_d, v, n*sizeof(float), cudaMemcpyHostToDevice);

  cudaBindTexture(0, v_Tex, v_d, n*sizeof(float));

  dim3 blocks(THREAD_PER_BLOCK);
  dim3 grids (n/THREAD_PER_BLOCK+1);
  mavg_device<<<grids, blocks>>> (n, v_d, ws, out_d);

  cudaMemcpy((void*)out,(void*)out_d,bytes,cudaMemcpyDeviceToHost);
  cudaFree(v_d);
  cudaFree(out_d);
}
```

Example: moving average

Optimization #2 – use texture memory

```
texture<float, 1, cudaReadModeElementType> v_Tex;

__global__ void
mavg_device(const float n, const float *v, const int ws, float *out)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j;
  float sum;

  if(i < ws-1) {
    out[i] = tex1Dfetch(v_Tex, i);
  }
  else if(i < n) {
    sum = 0;
    for(j=0;j<ws;j++) {
      sum += tex1Dfetch(v_Tex, i-j);
    }
    out[i] = sum/ws;
  }
}
```





Lesson Learned from Simple Moving Average

- Moving average is different from vector scaling in terms of “arithmetic intensity” (number of computations per memory reference)
- Moving average on CPU is bounded by memory bandwidth.
  - 176MFLOPS vs. 6.4GFLOPS theoretical peak.
- As GPU has much higher memory bandwidth, significant performance boost can be obtained!
- Use texture memory (read-only, cached) can further enhance performance. →100x faster than CPU version!
  - Note on the latest iterations of CUDA devices (e.g. Tesla C2050, GTX460, …) cache memory is introduced on the device. Texture memory may no longer provide any benefit!