Memory Hierarchies in CPU/GPU Architectures

Siegfried Höfinger

VSC Research Center, TU Wien

December 9, 2020

→ https://tinyurl.com/cuda4dummies/i/l2/notes-l2.pdf

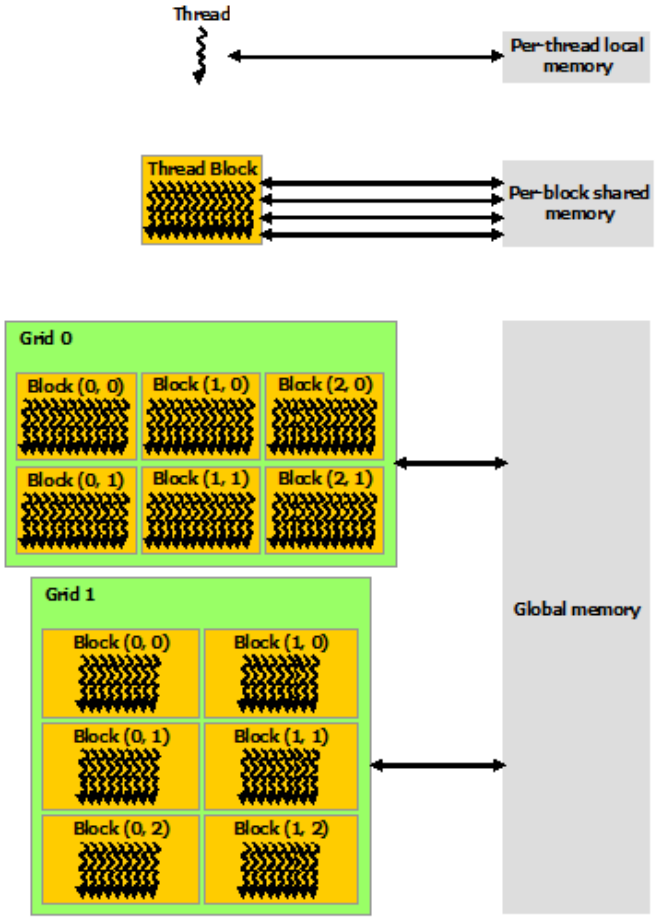
Outline

Memory Hierarchy

Take Home Messages

Memory Hierarchy

CUDA C-Programming Guide, Programming Model cont.

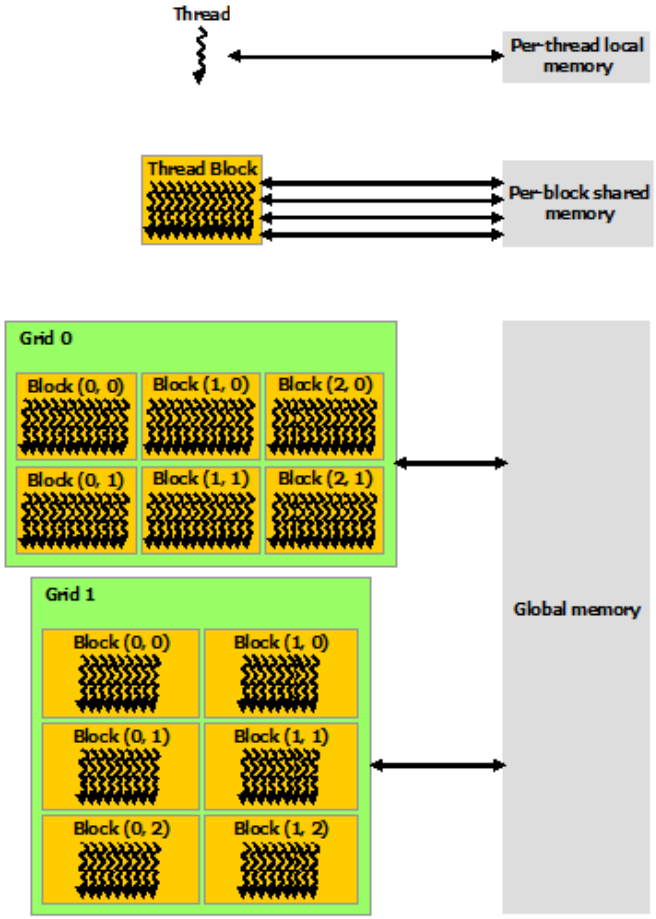


- Threads have access to several memory spaces

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Programming Model cont.

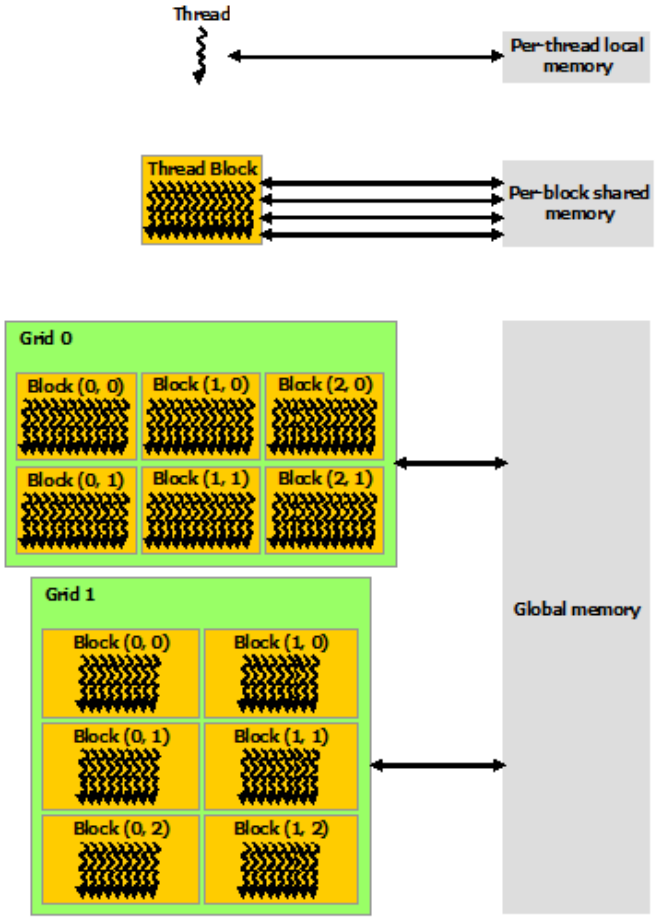


- Threads have access to several memory spaces
- Private local memory for each individual thread (fast)

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Programming Model cont.

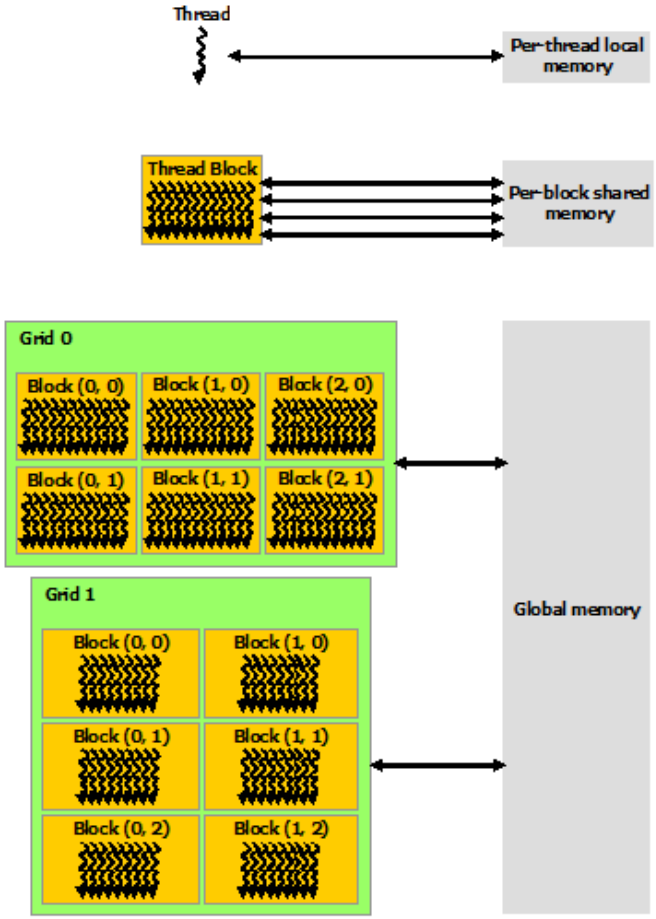


- Threads have access to several memory spaces
- Private local memory for each individual thread (fast)
- Shared memory for all threads within a thread block (pretty fast)

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Programming Model cont.

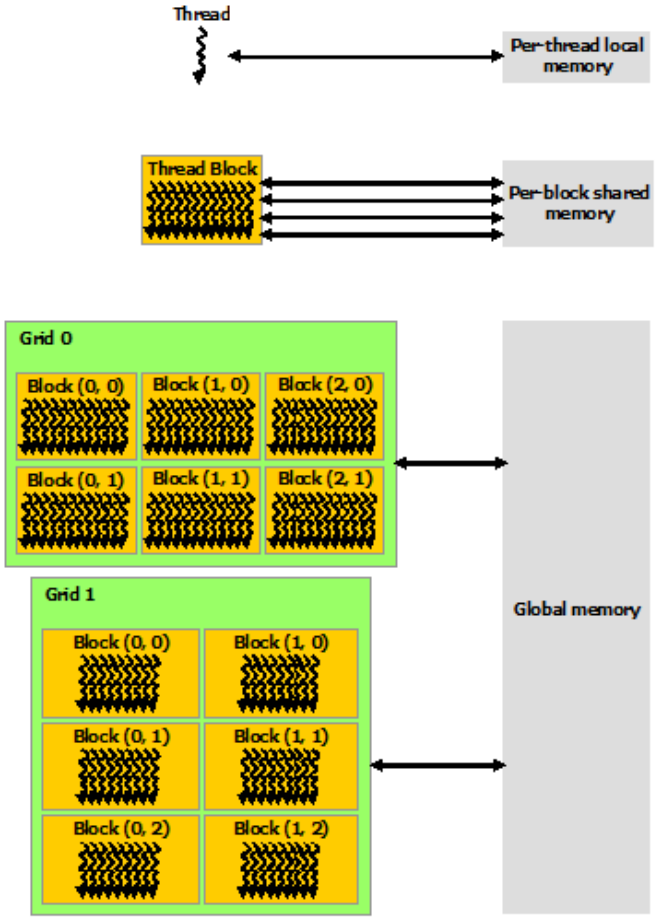


- Threads have access to several memory spaces
- Private local memory for each individual thread (fast)
- Shared memory for all threads within a thread block (pretty fast)
- Global memory accessible to all threads (slow)

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Programming Model cont.

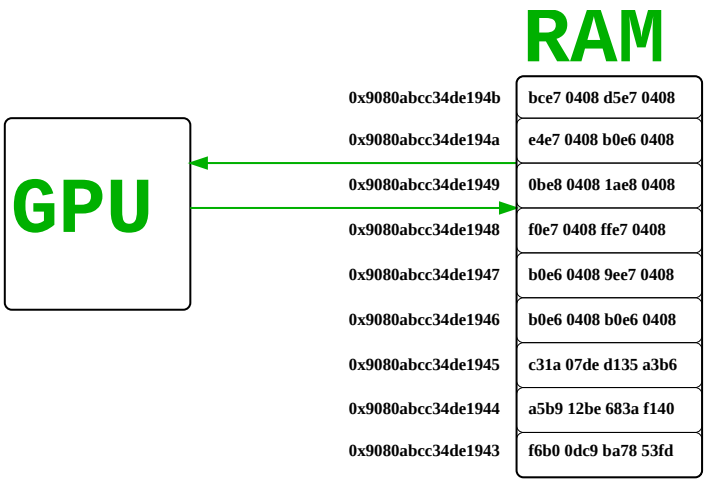


- Threads have access to several memory spaces
- Private local memory for each individual thread (fast)
- Shared memory for all threads within a thread block (pretty fast)
- Global memory accessible to all threads (slow)
- Special ROMs, constant and texture memory

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming



Unified Memory:

GPUs of compute capability $\geq$ 6.x and $\geq$ CUDA 8.0

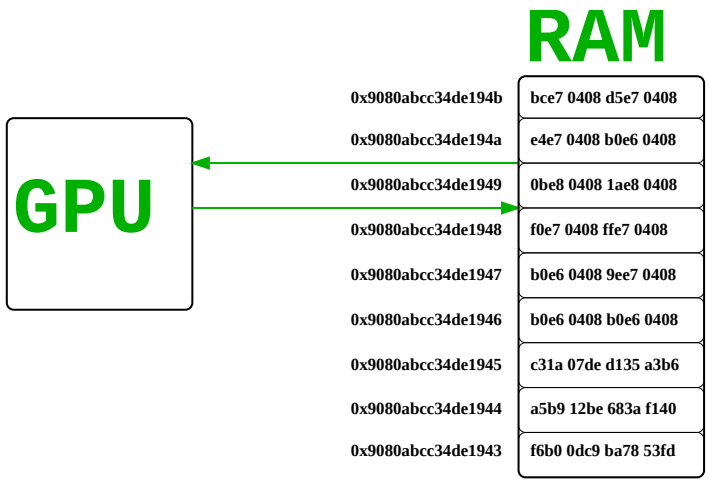
Separate Device/Host Memory:

GPUs of compute capability $<$ 6.x and $<$ CUDA 8.0

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming



Unified Memory:

GPUs of compute capability $\geq$ 6.x and $\geq$ CUDA 8.0

Separate Device/Host Memory:

GPUs of compute capability $<$ 6.x and $<$ CUDA 8.0

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

- Unified memory (managed) for recent GPUs (Pascal class or newer) greatly simplifies programming4/porting2 the GPU

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

- Unified memory (managed) for recent GPUs (Pascal class or newer) greatly simplifies programming4/porting2 the GPU
- Single common address space accessible from any processor in a system

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

- Unified memory (managed) for recent GPUs (Pascal class or newer) greatly simplifies programming4/porting2 the GPU
- Single common address space accessible from any processor in a system
- Strict distinction into device- and host-memory becoming less important to the programmer

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

- Unified memory (managed) for recent GPUs (Pascal class or newer) greatly simplifies programming4/porting2 the GPU
- Single common address space accessible from any processor in a system
- Strict distinction into device- and host-memory becoming less important to the programmer
- Single-pointer-to-data model, no longer any need to copy forth/back data into/from GPU memory

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

- Unified memory (managed) for recent GPUs (Pascal class or newer) greatly simplifies programming4/porting2 the GPU
- Single common address space accessible from any processor in a system
- Strict distinction into device- and host-memory becoming less important to the programmer
- Single-pointer-to-data model, no longer any need to copy forth/back data into/from GPU memory
- On-demand page migration (hardware supported) — Page Migration Engine

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

- Unified memory (managed) for recent GPUs (Pascal class or newer) greatly simplifies programming4/porting2 the GPU
- Single common address space accessible from any processor in a system
- Strict distinction into device- and host-memory becoming less important to the programmer
- Single-pointer-to-data model, no longer any need to copy forth/back data into/from GPU memory
- On-demand page migration (hardware supported) — Page Migration Engine
- Simply invoked via cudaMallocManaged()

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

Multiple Thread Blocks Matrix Addition

```
__global__void MatAdd( float **A, float **B, float **C )
{
    int i, j;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    j = (blockIdx.y * blockDim.y) + threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    int i;
    dim3 threadsPerBlock, numBlocks;
    float **A, **B, **C;
    ...
    // unified memory allocation, B and C analogous
    cudaMallocManaged(&A, N * sizeof(float *));
    for (i = 0; i < N; i++) {
       cudaMallocManaged(&A[i], N * sizeof(float));
    }
    // kernel invocation
    MatAdd <<< numBlocks, threadsPerBlock >>> (A, B, C);
    cudaDeviceSynchronize();
    ...
    cudaFree(A);
}
```

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

Multiple Thread Blocks Matrix Addition

Standard malloc-like allocation on the host

```
__global__void MatAdd( float **A, float **B, float **C )
{
    int i, j;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    j = (blockIdx.y * blockDim.y) + threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    int i;
    dim3 threadsPerBlock, numBlocks;
    float **A, **B, **C;
    ...
    // unified memory allocation, B and C analogous
    cudaMallocManaged(&A, N * sizeof(float *));
    for (i = 0; i < N; i++) {
        cudaMallocManaged(&A[i], N * sizeof(float));
    }
    // kernel invocation
    MatAdd <<< numBlocks, threadsPerBlock >>> (A, B, C);
    cudaDeviceSynchronize();
    ...
    cudaFree(A);
}
```

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

Multiple Thread Blocks Matrix Addition

```
__global__ void MatAdd( float **A, float **B, float **C )
{
    int i, j;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    j = (blockIdx.y * blockDim.y) + threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    int i;
    dim3 threadsPerBlock, numBlocks;
    float **A, **B, **C;
    ...
    // unified memory allocation, B and C analogous
    cudaMallocManaged(&A, N * sizeof(float *));
    for (i = 0; i < N; i++) {
        cudaMallocManaged(&A[i], N * sizeof(float));
    }
    // kernel invocation
    MatAdd <<< numBlocks, threadsPerBlock >>> (A, B, C);
    cudaDeviceSynchronize();
    ...
    cudaFree(A);
}
```

Standard malloc-like allocation on the host

host pointers directly usable in kernel code

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

Multiple Thread Blocks Matrix Addition

```
__global__ void MatAdd( float **A, float **B, float **C )
{
    int i, j;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    j = (blockIdx.y * blockDim.y) + threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    int i;
    dim3 threadsPerBlock, numBlocks;
    float **A, **B, **C;
    ...
    // unified memory allocation, B and C analogous
    cudaMallocManaged(&A, N * sizeof(float *));
    for (i = 0; i < N; i++) {
        cudaMallocManaged(&A[i], N * sizeof(float));
    }
    // kernel invocation
    MatAdd <<< numBlocks, threadsPerBlock >>> (A, B, C);
    cudaDeviceSynchronize();
    ...
    cudaFree(A);
}
```

Standard malloc-like allocation on the host

host pointers directly usable in kernel code

ensure proper kernel completion

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

Multiple Thread Blocks Matrix Addition

```
__global__ void MatAdd( float **A, float **B, float **C )
{
    int i, j;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    j = (blockIdx.y * blockDim.y) + threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    int i;
    dim3 threadsPerBlock, numBlocks;
    float **A, **B, **C;
    ...
    // unified memory allocation, B and C analogous
    cudaMallocManaged(&A, N * sizeof(float *));
    for (i = 0; i < N; i++) {
        cudaMallocManaged(&A[i], N * sizeof(float));
    }
    // kernel invocation
    MatAdd <<< numBlocks, threadsPerBlock >>> (A, B, C);
    cudaDeviceSynchronize();
    ...
    cudaFree(A);
}
```

Standard malloc-like allocation on the host

free memory when done

host pointers directly usable in kernel code

ensure proper kernel completion

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

2nd Form — Managed Global Memory

```
#define ARRAYDIM 268435456

// global managed declaration on GPU
__device__ __managed__ float x[ARRAYDIM], y[ARRAYDIM], z[ARRAYDIM];

__global__ void KrnlDmmy()
{
    int i;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    x[i] = y[i] + z[i];
    return;
}

int main()
{
    ...
   // kernel invocation
   KrnlDmmy <<< numBlocks, threadsPerBlock >>> ();
   cudaDeviceSynchronize();
    ...
}
```

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

2nd Form — Managed Global Memory

Global variables directly usable on device & host

```
#define ARRAYDIM 268435456

// global managed declaration on GPU
__device__ __managed__ float x[ARRAYDIM], y[ARRAYDIM], z[ARRAYDIM];

__global__ void KrnlDmmy()
{
    int i;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    x[i] = y[i] + z[i];
    return;
}

int main()
{
    ...
    // kernel invocation
    KrnlDmmy <<< numBlocks, threadsPerBlock >>> ();
    cudaDeviceSynchronize();
    ...
}
```

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

2nd Form — Managed Global Memory

Global variables directly usable on device & host

```
#define ARRAYDIM 268435456

// global managed declaration on GPU
__device__ __managed__ float x[ARRAYDIM], y[ARRAYDIM], z[ARRAYDIM];

__global__ void KrnlDmmy()
{
    int i;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    x[i] = y[i] + z[i];
    return;
}

int main()
{
    ...
    // kernel invocation
    KrnlDmmy <<< numBlocks, threadsPerBlock >>> ();
    cudaDeviceSynchronize();
    ...
}
```

void kernel call

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

2nd Form — Managed Global Memory

Global variables directly usable on device & host

```
#define ARRAYDIM 268435456

// global managed declaration on GPU
__device__ __managed__ float x[ARRAYDIM], y[ARRAYDIM], z[ARRAYDIM];

__global__ void KrnlDmmy()
{
    int i;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    x[i] = y[i] + z[i];
    return;
}

int main()
{
    ...
    // kernel invocation
    KrnlDmmy <<< numBlocks, threadsPerBlock >>> ();
    cudaDeviceSynchronize();
    ...
}
```

void kernel call

ensure proper kernel completion

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Unified Memory Example Version 1

```
#define ARRAYDIM 268435456

__global__ void KrnlDmmy(float *x, float *y, float *z)
{
    int i;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    x[i] = (float) i;
    y[i] = (float) (i + 1);
    z[i] = x[i] + y[i];
    return;
}

int main()
{
    ...
    // unified memory allocation, b and c analogous
    cudaMallocManaged(&a, ARRAYDIM * sizeof(float));
    // kernel invocation
    KrnlDmmy <<< numBlocks, threadsPerBlock >>> (a, b, c);
    cudaDeviceSynchronize();
     ...
}
```

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Unified Memory Example Version 1

```
#define ARRAYDIM 268435456

__global__ void KrnlDmmy(float *x, float *y, float *z)
{
    int i;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    x[i] = (float) i;
    y[i] = (float) (i + 1);
    z[i] = x[i] + y[i];
    return;
}

int main()
{
    ...
    // unified memory allocation, b and c analogous
    cudaMallocManaged(&a, ARRAYDIM * sizeof(float));
    // kernel invocation
    KrnlDmmy <<< numBlocks, threadsPerBlock >>> (a, b, c);
    cudaDeviceSynchronize();
     ...
}
```

Kernel initialization and calculation

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Unified Memory Example Version 1

```
#define ARRAYDIM 268435456

__global__ void KrnlDmmy(float *x, float *y, float *z)
{
    int i;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    x[i] = (float) i;
    y[i] = (float) (i + 1);
    z[i] = x[i] + y[i];
    return;
}

int main()
{
    ...
    // unified memory allocation, b and c analogous
    cudaMallocManaged(&a, ARRAYDIM * sizeof(float));
    // kernel invocation
    KrnlDmmy <<< numBlocks, threadsPerBlock >>> (a, b, c);
    cudaDeviceSynchronize();
     ...
}
```

Kernel initialization and calculation

1 GB arrays

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

```
[sh@n71-006]$ nvcc ./unified_memory_example_1.cu
[sh@n71-006]$ ./a.out
[sh@n71-006]$ nvprof ./a.out

==126896== NVPROF is profiling process 126896, command: ./a.out
==126896== Profiling application: ./a.out
==126896== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  558.88ms         1  558.88ms  558.88ms  558.88ms  KrnlDmmy(float*, float*, flo
      API calls:   66.84%  558.90ms         1  558.90ms  558.90ms  558.90ms  cudaDeviceSynchronize
                   20.99%  175.51ms         3  58.504ms  16.390us  175.45ms  cudaMallocManaged
                   12.04%  100.67ms         3  33.557ms  33.476ms  33.610ms  cudaFree
                    0.07%  616.14us        94  6.5540us     600ns  248.06us  cuDeviceGetAttribute
                    0.04%  364.67us         1  364.67us  364.67us  364.67us  cuDeviceTotalMem
                    0.01%  71.540us         1  71.540us  71.540us  71.540us  cudaLaunch
                    0.01%  69.450us         1  69.450us  69.450us  69.450us  cuDeviceGetName
                    0.00%  5.7300us         3  1.9100us     250ns  5.0900us  cudaSetupArgument
                    0.00%  4.5400us         3  1.5130us     630ns  2.8700us  cuDeviceGetCount
                    0.00%  2.3300us         2  1.1650us     650ns  1.6800us  cuDeviceGet
                    0.00%  1.2900us         1  1.2900us  1.2900us  1.2900us  cudaConfigureCall

==126896== Unified Memory profiling result:
Device "GeForce GTX 1080 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    5275         -         -         -           -  548.8451ms  Gpu page fault groups
```

→ https://tinyurl.com/cuda4dummies/i/l2/unified_memory_example_1.cu

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

```
[sh@n71-006]$  nvcc ./unified_memory_example_1.cu
[sh@n71-006]$  ./a.out
[sh@n71-006]$  nvprof ./a.out

==126896== NVPROF is profiling process 126896, command: ./a.out
==126896== Profiling application: ./a.out
==126896== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  558.88ms         1  558.88ms  558.88ms  558.88ms  KrnlDmmy(float*, float*, flo
      API calls:   66.84%  558.90ms         1  558.90ms  558.90ms  558.90ms  cudaDeviceSynchronize
                   20.99%  175.51ms         3  58.504ms  16.390us  175.45ms  cudaMallocManaged
                   12.04%  100.67ms         3  33.557ms  33.476ms  33.610ms  cudaFree
                    0.07%  616.14us        94  6.5540us     600ns  248.06us  cuDeviceGetAttribute
                    0.04%  364.67us         1  364.67us  364.67us  364.67us  cuDeviceTotalMem
                    0.01%  71.540us         1  71.540us  71.540us  71.540us  cudaLaunch
                    0.01%  69.450us         1  69.450us  69.450us  69.450us  cuDeviceGetName
                    0.00%  5.7300us         3  1.9100us     250ns  5.0900us  cudaSetupArgument
                    0.00%  4.5400us         3  1.5130us     630ns  2.8700us  cuDeviceGetCount
                    0.00%  2.3300us         2  1.1650us     650ns  1.6800us  cuDeviceGet
                    0.00%  1.2900us         1  1.2900us  1.2900us  1.2900us  cudaConfigureCall

==126896== Unified Memory profiling result:
Device "GeForce GTX 1080 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    5275         -         -         -           -  548.8451ms  Gpu page fault groups
```

→ https://tinyurl.com/cuda4dummies/i/l2/unified_memory_example_1.cu

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

- While straightforward profiling with nvprof is nice (especially in terms of unified memory analysis) the results showed a significant number of page faults

→ https://devblogs.nvidia.com/unified-memory-cuda-beginners

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

- While straightforward profiling with nvprof is nice (especially in terms of unified memory analysis) the results showed a significant number of page faults
- In a real application, the GPU is likely to use the data more often and for more complex computations, so the overhead from page faulting may become negligible;
  to quantify, we...

→ https://devblogs.nvidia.com/unified-memory-cuda-beginners

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

- While straightforward profiling with nvprof is nice (especially in terms of unified memory analysis) the results showed a significant number of page faults
- In a real application, the GPU is likely to use the data more often and for more complex computations, so the overhead from page faulting may become negligible;
  to quantify, we...

1. Could separate the actual calculation from the initialization with the help of a second kernel

→ https://devblogs.nvidia.com/unified-memory-cuda-beginners

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

- While straightforward profiling with nvprof is nice (especially in terms of unified memory analysis) the results showed a significant number of page faults
- In a real application, the GPU is likely to use the data more often and for more complex computations, so the overhead from page faulting may become negligible;
  to quantify, we...

1. Could separate the actual calculation from the initialization with the help of a second kernel
2. Could repeat the kernel doing the calculation many times

→ https://devblogs.nvidia.com/unified-memory-cuda-beginners

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

- While straightforward profiling with nvprof is nice (especially in terms of unified memory analysis) the results showed a significant number of page faults
- In a real application, the GPU is likely to use the data more often and for more complex computations, so the overhead from page faulting may become negligible;
  to quantify, we...

1. Could separate the actual calculation from the initialization with the help of a second kernel
2. Could repeat the kernel doing the calculation many times
3. Could use unified memory prefetching to explicitly move the data to the GPU

→ https://devblogs.nvidia.com/unified-memory-cuda-beginners

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

```
[sh@n71-006]$  nvcc ./unified_memory_example_2.cu
[sh@n71-006]$  nvprof ./a.out

==3316== NVPROF is profiling process 3316, command: ./a.out
==3316== Profiling application: ./a.out
==3316== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   63.49%  341.33ms         1  341.33ms  341.33ms  341.33ms  KrnlDmmyInit(float*, float*,
                   36.51%  196.25ms         1  196.25ms  196.25ms  196.25ms  KrnlDmmyCalc(float*, float*,
      API calls:   62.34%  537.60ms         2  268.80ms  196.26ms  341.34ms  cudaDeviceSynchronize
                   25.62%  220.97ms         3  73.657ms  16.760us  220.91ms  cudaMallocManaged
                   11.75%  101.31ms         3  33.770ms  33.658ms  33.845ms  cudaFree
                    0.11%  950.09us         2  475.05us  75.991us  874.10us  cudaLaunch
                    0.07%  607.36us        94  6.4610us     540ns  244.02us  cuDeviceGetAttribute
                    0.07%  578.76us         1  578.76us  578.76us  578.76us  cuDeviceGetName
                    0.04%  386.76us         1  386.76us  386.76us  386.76us  cuDeviceTotalMem
                    0.00%  8.7200us         6  1.4530us     230ns  5.4600us  cudaSetupArgument
                    0.00%  4.0800us         3  1.3600us     640ns  2.6100us  cuDeviceGetCount
                    0.00%  2.3400us         2  1.1700us  1.1400us  1.2000us  cudaConfigureCall
                    0.00%  2.0800us         2  1.0400us     620ns  1.4600us  cuDeviceGet

==3316== Unified Memory profiling result:
Device "GeForce GTX 1080 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    5503         -         -         -           -  533.2729ms  Gpu page fault groups
```

→ https://tinyurl.com/cuda4dummies/i/l2/unified_memory_example_2.cu

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

```
[sh@n71-006]$  nvcc ./unified_memory_example_2.cu
[sh@n71-006]$  nvprof ./a.out

==3316== NVPROF is profiling process 3316, command: ./a.out
==3316== Profiling application: ./a.out
==3316== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   63.49%  341.33ms         1  341.33ms  341.33ms  341.33ms  KrnlDmmyInit(float*, float*,
                   36.51%  196.25ms         1  196.25ms  196.25ms  196.25ms  KrnlDmmyCalc(float*, float*,
      API calls:   62.34%  537.60ms         2  268.80ms  196.26ms  341.34ms  cudaDeviceSynchronize
                   25.62%  220.97ms         3  73.657ms  16.760us  220.91ms  cudaMallocManaged
                   11.75%  101.31ms         3  33.770ms  33.658ms  33.845ms  cudaFree
                    0.11%  950.09us         2  475.05us  75.991us  874.10us  cudaLaunch
                    0.07%  607.36us        94  6.4610us     540ns  244.02us  cuDeviceGetAttribute
                    0.07%  578.76us         1  578.76us  578.76us  578.76us  cuDeviceGetName
                    0.04%  386.76us         1  386.76us  386.76us  386.76us  cuDeviceTotalMem
                    0.00%  8.7200us         6  1.4530us     230ns  5.4600us  cudaSetupArgument
                    0.00%  4.0800us         3  1.3600us     640ns  2.6100us  cuDeviceGetCount
                    0.00%  2.3400us         2  1.1700us  1.1400us  1.2000us  cudaConfigureCall
                    0.00%  2.0800us         2  1.0400us     620ns  1.4600us  cuDeviceGet

==3316== Unified Memory profiling result:
Device "GeForce GTX 1080 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    5503         -         -         -           -  533.2729ms  Gpu page fault groups
```

Compute kernel much faster now !

→ https://tinyurl.com/cuda4dummies/i/l2/unified_memory_example_2.cu

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

```
[sh@n71-006]$   nvcc ./unified_memory_example_2.cu
[sh@n71-006]$   nvprof ./a.out

==3316== NVPROF is profiling process 3316, command: ./a.out
==3316== Profiling application: ./a.out
==3316== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   63.49%  341.33ms         1  341.33ms  341.33ms  341.33ms  KrnlDmmyInit(float*, float*,
                                           1  196.25ms  196.25ms  196.25ms  KrnlDmmyCalc(float*, float*,
                                           2  268.80ms  196.26ms  341.34ms  cudaDeviceSynchronize
                                           3  73.657ms  16.760us  220.91ms  cudaMallocManaged
                                           3  33.770ms  33.658ms  33.845ms  cudaFree
                                           2  475.05us  75.991us  874.10us  cudaLaunch
                                          94  6.4610us     540ns  244.02us  cuDeviceGetAttribute
                                           1  578.76us  578.76us  578.76us  cuDeviceGetName
                                           1  386.76us  386.76us  386.76us  cuDeviceTotalMem
                                           6  1.4530us     230ns  5.4600us  cudaSetupArgument
                    0.00%  4.0800us         3  1.3600us     640ns  2.6100us  cuDeviceGetCount
                    0.00%  2.3400us         2  1.1700us  1.1400us  1.2000us  cudaConfigureCall
                    0.00%  2.0800us         2  1.0400us     620ns  1.4600us  cuDeviceGet

==3316== Unified Memory profiling result:
Device "GeForce GTX 1080 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    5503         -         -         -           -  533.2729ms  Gpu page fault groups
```

Compute kernel much faster now !

Still very low effective memory bandwith,

$$\frac{\frac{3\times 268435456\times 4}{10^9}}{196.25\times 10^{-3}} = 16.4\,GB/s$$

from theoretical $320\,GB/s$

→ https://tinyurl.com/cuda4dummies/i/l2/unified_memory_example_2.cu

VIENNA SCIENTIFIC CLUSTER

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

```
[sh@n71-006]$ nvcc ./unified_memory_example_3.cu
[sh@n71-006]$ nvprof ./a.out

==3497== NVPROF is profiling process 3497, command: ./a.out
==3497== Profiling application: ./a.out
==3497== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   82.04%  1.48704s       100  14.870ms  13.222ms  177.58ms  KrnlDmmyCalc(float*, float*,
                   17.96%  325.61ms         1  325.61ms  325.61ms  325.61ms  KrnlDmmyInit(float*, float*,
      API calls:   82.58%  1.81309s       101  17.951ms  13.225ms  325.63ms  cudaDeviceSynchronize
                                  s         3  90.630ms  16.100us  271.84ms  cudaMallocManaged
                                  s         3  33.626ms  33.537ms  33.672ms  cudaFree
                                  s       101  85.217us  42.150us  846.06us  cudaLaunch
                                  s        94  6.2330us     550ns  246.25us  cuDeviceGetAttribute
                                  s         1  346.65us  346.65us  346.65us  cuDeviceTotalMem
                                  s       303     298ns     200ns  4.9500us  cudaSetupArgument
                                  s         1  68.361us  68.361us  68.361us  cuDeviceGetName
                                  s       101     465ns     280ns  3.6800us  cudaConfigureCall
                                  s         3  1.4700us     640ns  2.9300us  cuDeviceGetCount
                                  s         2  1.0400us     590ns  1.4900us  cuDeviceGet

==3497== Unified Memory profiling result:
Device "GeForce GTX 1080 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    5509         -         -         -           -  501.2754ms  Gpu page fault groups
```

100× greatly improves effective memory bandwith, 217GB/s and compute performance

→ https://tinyurl.com/cuda4dummies/i/l2/unified_memory_example_3.cu

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

```
[sh@n71-006]$   nvcc ./unified_memory_example_4.cu
[sh@n71-006]$   nvprof ./a.out

==9756== NVPROF is profiling process 9756, command: ./a.out
==9756== Profiling application: ./a.out
==9756== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   79.51%  1.32179s       100  13.218ms  13.213ms  13.224ms  KrnlDmmyCalc(float*, float*,
                   20.49%  340.61ms         1  340.61ms  340.61ms  340.61ms  KrnlDmmyInit(float*, float*,
      API calls:   82.06%  1.66280s       101  16.463ms  13.215ms  340.62ms  cudaDeviceSynchronize
                   11.24%  227.77ms         3  75.924ms  27.941us  227.68ms  cudaMallocManaged
                    4.91%  99.538ms         3  33.179ms  32.044ms  33.897ms  cudaFree
                                 s         3  9.0126ms  305.04us  25.975ms  cudaMemPrefetchAsync
                                 s       101  79.679us  49.111us  553.30us  cudaLaunch
                                 s        94  6.1240us     590ns  239.69us  cuDeviceGetAttribute
                                 s         1  361.41us  361.41us  361.41us  cuDeviceTotalMem
                                 s       303     314ns     190ns  7.7900us  cudaSetupArgument
                                 s         1  69.211us  69.211us  69.211us  cuDeviceGetName
                    0.00%  58.470us       101     578ns     350ns  4.6800us  cudaConfigureCall
                    0.00%  4.4700us         3  1.4900us     700ns  2.6800us  cuDeviceGetCount

==9756== Unified Memory profiling result:
Device "GeForce GTX 1080 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    3184         -         -         -           -  351.4517ms  Gpu page fault groups
```

Prefetching: fastest, $244 GB/s$, fewest page faults

→ https://tinyurl.com/cuda4dummies/i/l2/unified_memory_example_4.cu

VIENNA SCIENTIFIC CLUSTER

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

```
[sh@n71-006]$  nvcc ./unified_memory_example_5.cu
[sh@n71-006]$  nvprof ./a.out

==11798== NVPROF is profiling process 11798, command: ./a.out
==11798== Profiling application: ./a.out
==11798== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   99.38%  1.32146s       100  13.215ms  13.208ms  13.220ms  KrnlDmmyCalc(void)
                    0.62%  8.2463ms         1  8.2463ms  8.2463ms  8.2463ms  KrnlDmmyInit(void)
      API calls:   67.65%  1.33004s       101  13.169ms  8.2460ms  13.241ms  cudaDeviceSynchronize
                   32.30%  635.07ms       101  6.2878ms  68.721us  627.53ms  cudaLaunch
                    0.03%  582.78us        94  6.1990us     610ns  233.16us  cuDeviceGetAttribute
                    0.02%  366.10us         1  366.10us  366.10us  366.10us  cuDeviceTotalMem
                    0.00%  66.710us         1  66.710us  66.710us  66.710us  cuDeviceGetName
                    0.00%  59.720us       101     591ns     310ns  13.860us  cudaConfigureCall
                    0.00%  4.2500us         3  1.4160us     650ns  2.5900us  cuDeviceGetCount
                    0.00%  2.2100us         2  1.1050us     630ns  1.5800us  cuDeviceGet
```

Globally managed ex aequo, $244 GB/s$, NO page faults

→ https://tinyurl.com/cuda4dummies/i/l2/unified_memory_example_5.cu

GPU Memory Oversubscription

```
#define DBLEARRAY16GB 2147483648

__global__ void KrnlDmmy(double *a)
{
    unsigned int i;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    a[i] = (double) i + 5.0e00;
    return;
}

int main()
{
    ...
    // unified memory allocation
    cudaMallocManaged(&c, DBLEARRAY16GB * sizeof(double));
    // kernel invocation
    KrnlDmmy <<< numBlocks, threadsPerBlock >>> (c);
    cudaDeviceSynchronize();
    sgnl = check_array(c);
    cudaFree(c);
    return(0);
}
```

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

GPU Memory Oversubscription

```
#define DBLEARRAY16GB 2147483648

__global__ void KrnlDmmy(double *a)
{
    unsigned int i;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    a[i] = (double) i + 5.0e00;
    return;
}

int main()
{
    ...
    // unified memory allocation
    cudaMallocManaged(&c, DBLEARRAY16GB * sizeof(double));
    // kernel invocation
    KrnlDmmy <<< numBlocks, threadsPerBlock >>> (c);
    cudaDeviceSynchronize();
    sgnl = check_array(c);
    cudaFree(c);
    return(0);
}
```

16 GB array

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

GPU Memory Oversubscription

```
#define DBLEARRAY16GB 2147483648

__global__ void KrnlDmmy(double *a)
{
    unsigned int i;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    a[i] = (double) i + 5.0e00;
    return;
}

int main()
{
    ...
    // unified memory allocation
    cudaMallocManaged(&c, DBLEARRAY16GB * sizeof(double));
    // kernel invocation
    KrnlDmmy <<< numBlocks, threadsPerBlock >>> (c);
    cudaDeviceSynchronize();
    sgnl = check_array(c);
    cudaFree(c);
    return(0);
}
```

Kernel calculation (8 GB on-board)

16 GB array

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

GPU Memory Oversubscription

```
#define DBLEARRAY16GB 2147483648

__global__ void KrnlDmmy(double *a)
{
    unsigned int i;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    a[i] = (double) i + 5.0e00;
    return;
}

int main()
{
    ...
    // unified memory allocation
    cudaMallocManaged(&c, DBLEARRAY16GB * sizeof(double));
    // kernel invocation
    KrnlDmmy <<< numBlocks, threadsPerBlock >>> (c);
    cudaDeviceSynchronize();
    sgnl = check_array(c);
    cudaFree(c);
    return(0);
}
```

Kernel calculation (8 GB on-board)

Simple correctness check

16 GB array

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

```
[sh@n71-006]$ nvcc ./unified_memory_oversubscription.cu
[sh@n71-006]$ nvprof ./a.out

==19259== NVPROF is profiling process 19259, command: ./a.out
expecting 2305843018877370368 while receiving 2305843018877370368.000000
==19259== Profiling application: ./a.out
==19259== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  4.01542s         1  4.01542s  4.01542s  4.01542s  KrnlDmmy(double*)
      API calls:   76.93%  4.01561s         1  4.01561s  4.01561s  4.01561s  cudaDeviceSynchronize
                   17.36%  906.20ms         1  906.20ms  906.20ms  906.20ms  cudaFree
                    5.69%  296.86ms         1  296.86ms  296.86ms  296.86ms  cudaMallocManaged
                    0.01%  604.35us        94  6.4290us     550ns  250.47us  cuDeviceGetAttribute
                    0.01%  364.77us         1  364.77us  364.77us  364.77us  cuDeviceTotalMem
                    0.00%  142.05us         1  142.05us  142.05us  142.05us  cudaLaunch
                    0.00%  70.930us         1  70.930us  70.930us  70.930us  cuDeviceGetName
                    0.00%  7.6000us         1  7.6000us  7.6000us  7.6000us  cudaSetupArgument
                    0.00%  4.9100us         3  1.6360us     810ns  3.2200us  cuDeviceGetCount
                    0.00%  2.8600us         1  2.8600us  2.8600us  2.8600us  cudaConfigureCall

==19259== Unified Memory profiling result:
Device "GeForce GTX 1080 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
   52018  322.53KB  4.0000KB  2.0000MB  16.00000GB   2.348844s  Device To Host
   40520         -         -         -           -  5.497636s  Gpu page fault groups
Total CPU Page faults: 28113
```

Much more time on CPU

Correctness test ok

Many page faults

→ https://tinyurl.com/cuda4dummies/i/l2/unified_memory_oversubscription.cu

Memory Hierarchy

CUDA C-Programming Guide, Programming Model cont.

How did this work in the days before CUDA managed unified memory...

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Unified Memory Programming cont.

- On pre-Pascal-class GPUs cudaMallocManaged() behaves like cudaMalloc()
- On pre-Pascal-class GPUs managed allocations are automatically visible to all GPUs in a system via peer-to-peer capabilities
- Managed memory is allocated in GPU memory as long as all GPUs have peer-to-peer support enabled, otherwise the driver will migrate all managed allocations to the CPU/host memory ("zero-copy" memory)
- P2P GPUs will experience PCIe bandwidth restrictions

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Device Memory

On pre-Pascal-class GPUs a distinction is made between host- and device-memory. Kernels operate out of device memory, so the CUDA runtime provides functions to allocate, deallocate, and copy data into device memory as well as transfer data between host memory and device memory. A typical sequence of operations is:

1. Declare and allocate host and device memory
2. Initialize host data
3. Transfer data from the host to the device
4. Execute one or more kernels
5. Transfer results from the device to the host

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide
→ https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c

Memory Hierarchy

CUDA C-Programming Guide, Device Memory cont.

```
#define ARRAYDIM 268435456
__global__ void KrnlDmmy(float *x, float *y, float *z)
{
    int i;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    z[i] = x[i] + y[i];
    return;
}

int main()
{
    ...
    // host memory allocation, hb, hc analogous
    ha = (float *) malloc(ARRAYDIM * sizeof(float));
    // device memory allocation, db, dc analogous
    cudaMalloc(&da, ARRAYDIM * sizeof(float));
    // host to device memory transfer
    cudaMemcpy(da, ha, ARRAYDIM * sizeof(float), cudaMemcpyHostToDevice);
    // kernel invocation
    KrnlDmmy <<< numBlocks, threadsPerBlock >>> (da, db, dc);
    // device to host memory transfer
    cudaMemcpy(hc, dc, ARRAYDIM * sizeof(float), cudaMemcpyDeviceToHost);
    // free device/host memory
    cudaFree(da);
    free(ha);
    ...
}
```

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Device Memory cont.

```
#define ARRAYDIM 268435456
__global__ void KrnlDmmy(float *x, float *y, float *z)
{
    int i;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    z[i] = x[i] + y[i];
    return;
}

int main()
{
    ...
    // host memory allocation, hb, hc analogous
    ha = (float *) malloc(ARRAYDIM * sizeof(float));
    // device memory allocation, db, dc analogous
    cudaMalloc(&da, ARRAYDIM * sizeof(float));
    // host to device memory transfer
    cudaMemcpy(da, ha, ARRAYDIM * sizeof(float), cudaMemcpyHostToDevice);
    // kernel invocation
    KrnlDmmy <<< numBlocks, threadsPerBlock >>> (da, db, dc);
    // device to host memory transfer
    cudaMemcpy(hc, dc, ARRAYDIM * sizeof(float), cudaMemcpyDeviceToHost);
    // free device/host memory
    cudaFree(da);
    free(ha);
    ...
}
```

Kernel memory allocation & set up

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Device Memory cont.

```
#define ARRAYDIM 268435456
__global__ void KrnlDmmy(float *x, float *y, float *z)
{
    int i;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    z[i] = x[i] + y[i];
    return;
}

int main()
{
    ...
    // host memory allocation, hb, hc analogous
    ha = (float *) malloc(ARRAYDIM * sizeof(float));
    // device memory allocation, db, dc analogous
    cudaMalloc(&da, ARRAYDIM * sizeof(float));
    // host to device memory transfer
    cudaMemcpy(da, ha, ARRAYDIM * sizeof(float), cudaMemcpyHostToDevice);
    // kernel invocation
    KrnlDmmy <<< numBlocks, threadsPerBlock >>> (da, db, dc);
    // device to host memory transfer
    cudaMemcpy(hc, dc, ARRAYDIM * sizeof(float), cudaMemcpyDeviceToHost);
    // free device/host memory
    cudaFree(da);
    free(ha);
    ...
}
```

Kernel memory allocation & set up

Stnd kernel call

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Device Memory cont.

```
#define ARRAYDIM 268435456
__global__ void KrnlDmmy(float *x, float *y, float *z)
{
    int i;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    z[i] = x[i] + y[i];
    return;
}

int main()
{
    ...
    // host memory allocation, hb, hc analogous
    ha = (float *) malloc(ARRAYDIM * sizeof(float));
    // device memory allocation, db, dc analogous
    cudaMalloc(&da, ARRAYDIM * sizeof(float));
    // host to device memory transfer
    cudaMemcpy(da, ha, ARRAYDIM * sizeof(float), cudaMemcpyHostToDevice);
    // kernel invocation
    KrnlDmmy <<< numBlocks, threadsPerBlock >>> (da, db, dc);
    // device to host memory transfer
    cudaMemcpy(hc, dc, ARRAYDIM * sizeof(float), cudaMemcpyDeviceToHost);
    // free device/host memory
    cudaFree(da);
    free(ha);
    ...
}
```

Kernel memory allocation & set up

Stnd kernel call

Device memory back-transfer

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Device Memory cont.

- Special forms cudaMallocPitch() and cudaMalloc3D() for 2D and 3D arrays
- Optimized for best performance when accessing via pointers
- Also good for device copies cudaMemcpy2D() and cudaMemcpy3D()
- Returned pitch (or stride) must be used to access array elements
- Optimally padded to meet alignment requirements
- Additional types of global memory, e.g. __device__ float *devPointer

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Matrix-Matrix Multiplication (simplest relationships):

$$\underbrace{\begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,N} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,N} \\ \vdots & \vdots & \vdots & \vdots \\ A_{k,1} & A_{k,2} & \cdots & A_{k,N} \\ \vdots & \vdots & \vdots & \vdots \\ A_{N,1} & A_{N,2} & \cdots & A_{N,N} \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} B_{1,1} & B_{1,2} & \cdots & B_{1,N} \\ B_{2,1} & B_{2,2} & \cdots & B_{2,N} \\ \vdots & \vdots & \vdots & \vdots \\ B_{k,1} & B_{k,2} & \cdots & B_{k,N} \\ \vdots & \vdots & \vdots & \vdots \\ B_{N,1} & B_{N,2} & \cdots & B_{N,N} \end{pmatrix}}_{\mathbf{B}} = \underbrace{\begin{pmatrix} C_{1,1} & C_{1,2} & \cdots & C_{1,N} \\ C_{2,1} & C_{2,2} & \cdots & C_{2,N} \\ \vdots & \vdots & \vdots & \vdots \\ C_{k,1} & C_{k,2} & \cdots & C_{k,N} \\ \vdots & \vdots & \vdots & \vdots \\ C_{N,1} & C_{N,2} & \cdots & C_{N,N} \end{pmatrix}}_{\mathbf{C}}$$

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Shared Memory cont.

Matrix Matrix Multiplication Version 1

```
#define N 4800
__global__ void KrnlMMM(float **A, float **B, float **C)
{
    int i, j, k;
    float tmpC;
    i = ((blockIdx.x * blockDim.x) + threadIdx.x);
    j = ((blockIdx.y * blockDim.y) + threadIdx.y);
    tmpC = (float) 0;
    for (k=0; k<N; k++) {
        tmpC += A[i][k] * B[k][j];
    }
    C[i][j] = tmpC;
    return;
}

int main()
{
    ...
    threadsPerBlock.x = 16;
    threadsPerBlock.y = 16;
    numBlocks.x = N / threadsPerBlock.x;
    numBlocks.y = N / threadsPerBlock.y;
    KrnlMMM <<< numBlocks, threadsPerBlock >>> (A, B, C);
    ...
}
```

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Shared Memory cont.

Matrix Matrix Multiplication Version 1

```
#define N 4800
__global__ void KrnlMMM(float **A, float **B, float **C)
{
    int i, j, k;
    float tmpC;
    i = ((blockIdx.x * blockDim.x) + threadIdx.x);
    j = ((blockIdx.y * blockDim.y) + threadIdx.y);
    tmpC = (float) 0;
    for (k=0; k<N; k++) {
        tmpC += A[i][k] * B[k][j];
    }
    C[i][j] = tmpC;
    return;
}

int main()
{
    ...
    threadsPerBlock.x = 16;
    threadsPerBlock.y = 16;
    numBlocks.x = N / threadsPerBlock.x;
    numBlocks.y = N / threadsPerBlock.y;
    KrnlMMM <<< numBlocks, threadsPerBlock >>> (A, B, C);
    ...
}
```

Each thread computes its specific C[i][j]

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Shared Memory cont.

Matrix Matrix Multiplication Version 1

```
#define N 4800
__global__ void KrnlMMM(float **A, float **B, float **C)
{
    int i, j, k;
    float tmpC;
    i = ((blockIdx.x * blockDim.x) + threadIdx.x);
    j = ((blockIdx.y * blockDim.y) + threadIdx.y);
    tmpC = (float) 0;
    for (k=0; k<N; k++) {
        tmpC += A[i][k] * B[k][j];
    }
    C[i][j] = tmpC;
    return;
}

int main()
{
    ...
    threadsPerBlock.x = 16;
    threadsPerBlock.y = 16;
    numBlocks.x = N / threadsPerBlock.x;
    numBlocks.y = N / threadsPerBlock.y;
    KrnlMMM <<< numBlocks, threadsPerBlock >>> (A, B, C);
    ...
}
```

Far-from-optimal memory access !

Each thread computes its specific C[i][j]

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

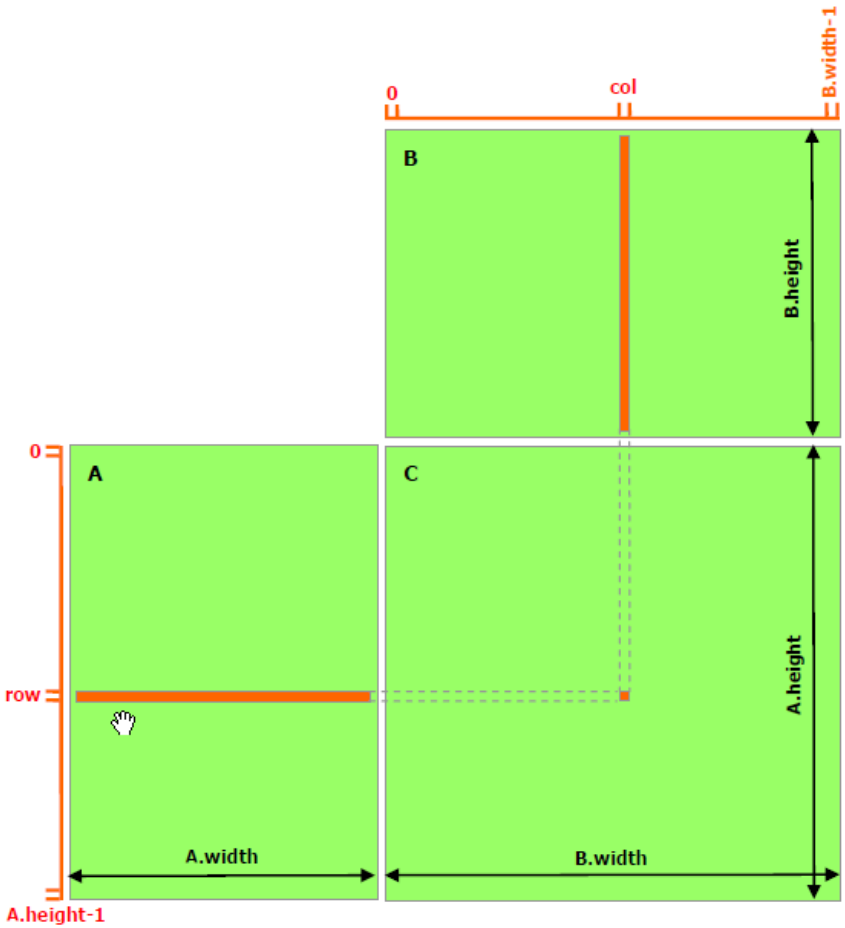
CUDA C-Programming Guide, Shared Memory cont.

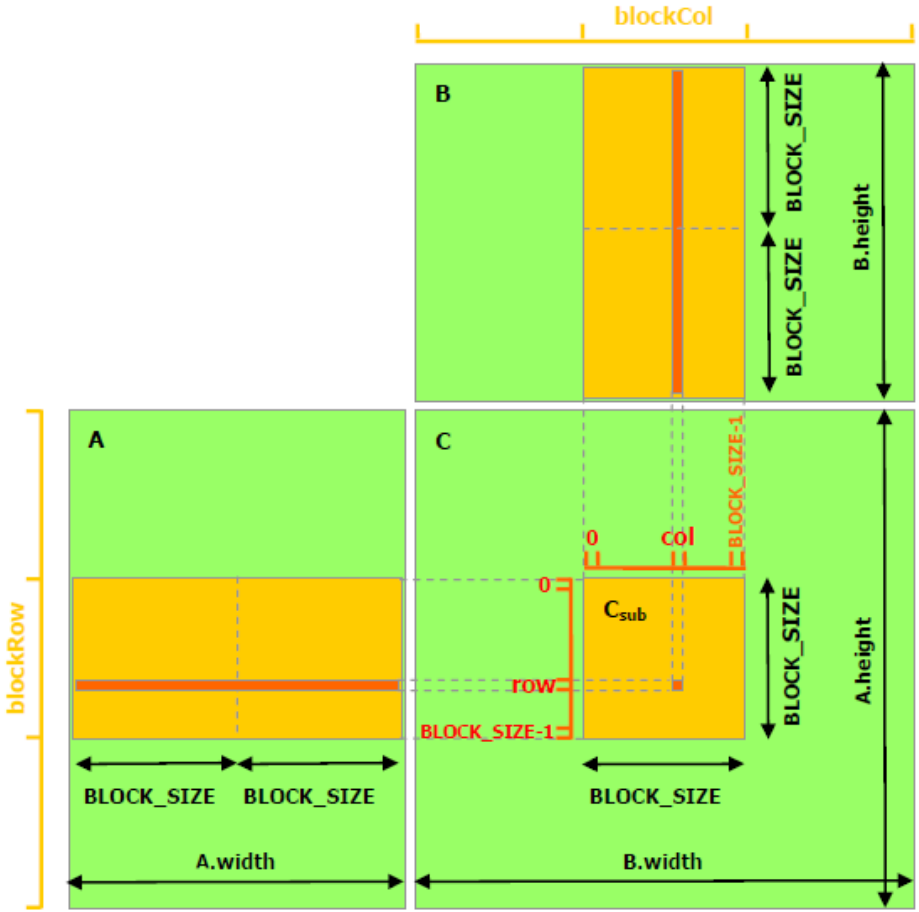
```
[sh@n71-006]$  nvcc ./mmm_example_2.cu
[sh@n71-006]$  nvprof ./a.out

==61563== NVPROF is profiling process 61563, command: ./a.out
==61563== Profiling application: ./a.out
==61563== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  3.53112s         1  3.53112s  3.53112s  3.53112s  KrnlMMM(float**, float**, fl
      API calls:   89.97%  3.53114s         1  3.53114s  3.53114s  3.53114s  cudaDeviceSynchronize
                   10.00%  392.31ms     14403  27.238us  7.9600us  240.64ms  cudaMallocManaged
                    0.01%  575.54us        94  6.1220us     560ns  235.94us  cuDeviceGetAttribute
                    0.01%  359.78us         1  359.78us  359.78us  359.78us  cuDeviceTotalMem
                    0.00%  183.95us         3  61.317us  22.690us  109.91us  cudaFree
                    0.00%  99.741us         1  99.741us  99.741us  99.741us  cudaLaunch
                    0.00%  67.791us         1  67.791us  67.791us  67.791us  cuDeviceGetName
                    0.00%  8.2200us         3  2.7400us     270ns  7.1800us  cudaSetupArgument
                    0.00%  6.8700us         1  6.8700us  6.8700us  6.8700us  cudaConfigureCall
                    0.00%  4.2700us         3  1.4230us     750ns  2.4100us  cuDeviceGetCount

==61563== Unified Memory profiling result:
Device "GeForce GTX 1080 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    4495  61.508KB  4.0000KB  0.9961MB  270.0000MB  25.77424ms  Host To Device
    2435  113.54KB  4.0000KB  0.9922MB  270.0000MB  23.44659ms  Device To Host
     407         -         -         -           -  74.12499ms  Gpu page fault groups
Total CPU Page faults: 1622
```

→ https://tinyurl.com/cuda4dummies/i/l2/mmm_example_2.cu

Memory Hierarchy

CUDA C-Programming Guide, Shared Memory cont.





→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Shared Memory cont.

Matrix Matrix Multiplication Version 2

```
#define N 4800
#define BS 16
__global__ void KrnlMMM(float **A, float **B, float **C)
{
    int i, j, k, Blckk;
    __shared__ float BlckA[BS][BS], BlckB[BS][BS], BlckC[BS][BS];
    float tmpC;
    i = threadIdx.y;
    j = threadIdx.x;
    BlckC[i][j] = (float) 0;
    for (Blckk = 0; Blckk < (N/BS); Blckk++) {
        BlckA[i][j] = A[(blockIdx.y*BS)+i][(Blckk*BS)+j];
        BlckB[i][j] = B[(Blckk*BS)+i][(blockIdx.x*BS)+j];
        __syncthreads();
        tmpC = (float) 0;
        for (k = 0; k<BS; k++) {
            tmpC += BlckA[i][k] * BlckB[k][j];
        }
        BlckC[i][j] += tmpC;
        __syncthreads();
    }
    C[(blockIdx.y*BS+i][(blockIdx.x*BS)+j] = BlckC[i][j];
    __syncthreads();
    return;
}
```

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Shared Memory cont.

Matrix Matrix Multiplication Version 2

Declaration of shared arrays

```
#define N 4800
#define BS 16
__global__ void KrnlMMM(float **A, float **B, float **C)
{
    int i, j, k, Blckk;
    __shared__ float BlckA[BS][BS], BlckB[BS][BS], BlckC[BS][BS];
    float tmpC;
    i = threadIdx.y;
    j = threadIdx.x;
    BlckC[i][j] = (float) 0;
    for (Blckk = 0; Blckk < (N/BS); Blckk++) {
        BlckA[i][j] = A[(blockIdx.y*BS)+i][(Blckk*BS)+j];
        BlckB[i][j] = B[(Blckk*BS)+i][(blockIdx.x*BS)+j];
        __syncthreads();
        tmpC = (float) 0;
        for (k = 0; k<BS; k++) {
            tmpC += BlckA[i][k] * BlckB[k][j];
        }
        BlckC[i][j] += tmpC;
        __syncthreads();
    }
    C[(blockIdx.y*BS+i][(blockIdx.x*BS)+j] = BlckC[i][j];
    __syncthreads();
    return;
}
```

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Shared Memory cont.

Matrix Matrix Multiplication Version 2

Declaration of shared arrays

Initialization by entire threadblock

```
#define N 4800
#define BS 16
__global__ void KrnlMMM(float **A, float **B, float **C)
{
    int i, j, k, Blckk;
    __shared__ float BlckA[BS][BS], BlckB[BS][BS], BlckC[BS][BS];
    float tmpC;
    i = threadIdx.y;
    j = threadIdx.x;
    BlckC[i][j] = (float) 0;
    for (Blckk = 0; Blckk < (N/BS); Blckk++) {
        BlckA[i][j] = A[(blockIdx.y*BS)+i][(Blckk*BS)+j];
        BlckB[i][j] = B[(Blckk*BS)+i][(blockIdx.x*BS)+j];
        __syncthreads();
        tmpC = (float) 0;
        for (k = 0; k<BS; k++) {
            tmpC += BlckA[i][k] * BlckB[k][j];
        }
        BlckC[i][j] += tmpC;
        __syncthreads();
    }
    C[(blockIdx.y*BS+i][(blockIdx.x*BS)+j] = BlckC[i][j];
    __syncthreads();
    return;
}
```

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Shared Memory cont.

Matrix Matrix Multiplication Version 2

Declaration of shared arrays

Initialization by entire threadblock

Loop combining A/B blocks in a dot-product like fashion

```
#define N 4800
#define BS 16
__global__ void KrnlMMM(float **A, float **B, float **C)
{
    int i, j, k, Blckk;
    __shared__ float BlckA[BS][BS], BlckB[BS][BS], BlckC[BS][BS];
    float tmpC;
    i = threadIdx.y;
    j = threadIdx.x;
    BlckC[i][j] = (float) 0;
    for (Blckk = 0; Blckk < (N/BS); Blckk++) {
        BlckA[i][j] = A[(blockIdx.y*BS)+i][(Blckk*BS)+j];
        BlckB[i][j] = B[(Blckk*BS)+i][(blockIdx.x*BS)+j];
        __syncthreads();
        tmpC = (float) 0;
        for (k = 0; k<BS; k++) {
            tmpC += BlckA[i][k] * BlckB[k][j];
        }
        BlckC[i][j] += tmpC;
        __syncthreads();
    }
    C[(blockIdx.y*BS+i][(blockIdx.x*BS)+j] = BlckC[i][j];
    __syncthreads();
    return;
}
```

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Shared Memory cont.

Matrix Matrix Multiplication Version 2

Declaration of shared arrays

Initialization by entire threadblock

Copy into shared memory arrays

Loop combining A/B blocks in a dot-product like fashion

```
#define N 4800
#define BS 16
__global__ void KrnlMMM(float **A, float **B, float **C)
{
    int i, j, k, Blckk;
    __shared__ float BlckA[BS][BS], BlckB[BS][BS], BlckC[BS][BS];
    float tmpC;
    i = threadIdx.y;
    j = threadIdx.x;
    BlckC[i][j] = (float) 0;
    for (Blckk = 0; Blckk < (N/BS); Blckk++) {
        BlckA[i][j] = A[(blockIdx.y*BS)+i][(Blckk*BS)+j];
        BlckB[i][j] = B[(Blckk*BS)+i][(blockIdx.x*BS)+j];
        __syncthreads();
        tmpC = (float) 0;
        for (k = 0; k<BS; k++) {
            tmpC += BlckA[i][k] * BlckB[k][j];
        }
        BlckC[i][j] += tmpC;
        __syncthreads();
    }
    C[(blockIdx.y*BS+i][(blockIdx.x*BS)+j] = BlckC[i][j];
    __syncthreads();
    return;
}
```

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Shared Memory cont.

Matrix Matrix Multiplication Version 2

Declaration of shared arrays

Initialization by entire threadblock

Copy into shared memory arrays

```
#define N 4800
#define BS 16
__global__ void KrnlMMM(float **A, float **B, float **C)
{
    int i, j, k, Blckk;
    __shared__ float BlckA[BS][BS], BlckB[BS][BS], BlckC[BS][BS];
    float tmpC;
    i = threadIdx.y;
    j = threadIdx.x;
    BlckC[i][j] = (float) 0;
    for (Blckk = 0; Blckk < (N/BS); Blckk++) {
        BlckA[i][j] = A[(blockIdx.y*BS)+i][(Blckk*BS)+j];
        BlckB[i][j] = B[(Blckk*BS)+i][(blockIdx.x*BS)+j];
        __syncthreads();
        tmpC = (float) 0;
        for (k = 0; k<BS; k++) {
            tmpC += BlckA[i][k] * BlckB[k][j];
        }
        BlckC[i][j] += tmpC;
        __syncthreads();
    }
    C[(blockIdx.y*BS+i][(blockIdx.x*BS)+j] = BlckC[i][j];
    __syncthreads();
    return;
}
```

Loop combining A/B blocks in a dot-product like fashion

Thread-wise update of BlckC

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Shared Memory cont.

Matrix Matrix Multiplication Version 2

```
#define N 4800
#define BS 16
__global__ void KrnlMMM(float **A, float **B, float **C)
{
    int i, j, k, Blckk;
    __shared__ float BlckA[BS][BS], BlckB[BS][BS], BlckC[BS][BS];
    float tmpC;
    i = threadIdx.y;
    j = threadIdx.x;
    BlckC[i][j] = (float) 0;
    for (Blckk = 0; Blckk < (N/BS); Blckk++) {
        BlckA[i][j] = A[(blockIdx.y*BS)+i][(Blckk*BS)+j];
        BlckB[i][j] = B[(Blckk*BS)+i][(blockIdx.x*BS)+j];
        __syncthreads();
        tmpC = (float) 0;
        for (k = 0; k<BS; k++) {
            tmpC += BlckA[i][k] * BlckB[k][j];
        }
        BlckC[i][j] += tmpC;
        __syncthreads();
    }
    C[(blockIdx.y*BS+i][(blockIdx.x*BS)+j] = BlckC[i][j];
    __syncthreads();
    return;
}
```

Declaration of shared arrays

Initialization by entire threadblock

Copy into shared memory arrays

Loop combining A/B blocks in a dot-product like fashion

Thread-wise update of BlckC

Back-copy from shared to global memory

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Memory Hierarchy

CUDA C-Programming Guide, Shared Memory cont.

```
[sh@n71-006]$ nvcc ./mmm_example_3.cu
[sh@n71-006]$ nvprof ./a.out

==61610== NVPROF is profiling process 61610, command: ./a.out
==61610== Profiling application: ./a.out
==61610== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:  100.00%  374.99ms         1  374.99ms  374.99ms  374.99ms  KrnlMMM(float**, float**, fl
      API calls:   51.52%  400.09ms     14403  27.778us  7.9300us  246.60ms  cudaMallocManaged
                   48.31%  375.18ms         1  375.18ms  375.18ms  375.18ms  cudaDeviceSynchronize
                    0.07%  576.53us        94  6.1330us     560ns  233.07us  cuDeviceGetAttribute
                    0.04%  348.92us         1  348.92us  348.92us  348.92us  cuDeviceTotalMem
                    0.03%  226.24us         3  75.414us  23.070us  149.58us  cudaFree
                    0.01%  101.57us         1  101.57us  101.57us  101.57us  cudaLaunch
                    0.01%  65.181us         1  65.181us  65.181us  65.181us  cuDeviceGetName
                    0.00%  9.8100us         1  9.8100us  9.8100us  9.8100us  cudaConfigureCall
                    0.00%  9.7700us         3  3.2560us     250ns  9.1200us  cudaSetupArgument
                    0.00%  4.1500us         3  1.3830us     610ns  2.5600us  cuDeviceGetCount

==61610== Unified Memory profiling result:
Device "GeForce GTX 1080 (0)"
   Count  Avg Size  Min Size  Max Size  Total Size  Total Time  Name
    5988  46.172KB  4.0000KB  0.9922MB  270.0000MB  26.24221ms  Host To Device
    2390  115.68KB  4.0000KB  0.9805MB  270.0000MB  23.27978ms  Device To Host
     350         -         -         -           -  81.76749ms  Gpu page fault groups
Total CPU Page faults: 1622
```

Factor ≈10× faster

→ https://tinyurl.com/cuda4dummies/i/l2/mmm_example_3.cu

Memory Hierarchy

CUDA C-Programming Guide, Local Memory, Registers

- Local memory for certain automatic variables
- Access to local memory is similar to global memory, i.e. high latency and low bandwidth
- Organized such that consecutive 32-bit words are accessed by consecutive thread IDs
- Analyzable with Nsight
- Fastest memory is registers, L1, for small, statically indexed arrays, e.g. A[16]

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide
→ https://stackoverflow.com/questions/10297067/in-a-cuda-kernel-how-do-i-store-an-array-in-local-threa

Take Home Messages

- Unified memory — a big improvement

Take Home Messages

- Unified memory — a big improvement
- Single address space also facilitates easier handling of 2D/3D arrays on the GPU

Take Home Messages

- Unified memory — a big improvement
- Single address space also facilitates easier handling of 2D/3D arrays on the GPU
- Straightforward profiling with nvprof

Take Home Messages

- Unified memory — a big improvement
- Single address space also facilitates easier handling of 2D/3D arrays on the GPU
- Straightforward profiling with nvprof
- Page migration engine facilitates seamless data transfer for array sizes exceeding largely the amount of RAM available on board of the GPU

Take Home Messages

- Unified memory — a big improvement
- Single address space also facilitates easier handling of 2D/3D arrays on the GPU
- Straightforward profiling with nvprof
- Page migration engine facilitates seamless data transfer for array sizes exceeding largely the amount of RAM available on board of the GPU
- Using shared memory greatly improves kernel performance