CUDA Misc

Mergesort, Pinned Memory, Device Query, Multi GPU

Parallel Mergesort

• O(N) runtime with memory copy overhead
  – Not really worth it compared to O(NlgN) sequential version but an interesting exercise

• Regular mergesort
CUDA Mergesort

• Split portion
  – Assign each thread to a number in the unsorted array
  – Example: 2 blocks, 4 threads per block

\[
\begin{array}{ccccccccc}
\text{BOT0} & \text{BOT1} & \text{BOT2} & \text{BOT3} & \text{B1T0} & \text{B1T1} & \text{B1T2} & \text{B1T3} \\
38 & 27 & 43 & 3 & 9 & 82 & 15 & 37
\end{array}
\]

index = threadIdx.x + (blockIdx.x * blockDim.x)

e.g. index = 3 + (1 * 4) = 7 for Block1 Thread 3

• Merge split into two phases
  – First phase: Sort each block by merging into shared memory

\[
\begin{array}{cccccc}
\text{BOT0} & \text{BOT2} & \text{B1T0} & \text{B1T2} \\
27 & 38 & 3 & 43 & 9 & 82 & 15 & 37
\end{array}
\]

\[
\begin{array}{cccc}
\text{BOT0} & \text{B1T0} \\
3 & 38 & 27 & 43 & 9 & 15 & 37 & 82
\end{array}
\]

Why can’t we keep doing this for the whole array?

Code to sort blocks

```
// This version only works for N = THREADS*BLOCKS
__global__ void sortBlocks(int *a)
{
    int i=2;
    __shared__ int temp[THREADS];
    while (i <= THREADS)
    {
        if ((threadIdx.x % i)==0)
        {
            int index1 = threadIdx.x + (blockIdx.x * blockDim.x);
            int endIndex1 = index1 + i/2;
            int index2 = endIndex1;
            int endIndex2 = index2 + i/2;
            int targetIndex = threadIdx.x;
            int done = 0;
            while (!done)
            {
                if ((index1 == endIndex1) && (index2 < endIndex2))
                    temp[targetIndex++] = a[index2++];
                else if ((index2 == endIndex2) && (index1 < endIndex1))
                    temp[targetIndex++] = a[index1++];
                else if (a[index1] < a[index2])
                    temp[targetIndex++] = a[index1++];
                else
                    temp[targetIndex++] = a[index2++];
                if ((index1==endIndex1) && (index2==endIndex2))
                    done = 1;
            }
        }
    }
    __syncthreads();
    a[threadIdx.x + (blockIdx.x*blockDim.x)] = temp[threadIdx.x];
    __syncthreads();
    i *= 2;
}
```
int main()
{
    int a[N];
    int *dev_a, *dev_temp;

cudaMalloc((void **) &dev_a, N*sizeof(int));
cudaMalloc((void **) &dev_temp, N*sizeof(int));

    // Fill array
    srand(time(NULL));
    for (int i = 0; i < N; i++)
    {
        int num = rand() % 100;
        a[i] = num;
        printf("%d ",a[i]);
    }
    printf(\n);

    // Copy data from host to device
    cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);

    sortBlocks<<<BLOCKS,THREADS>>>(dev_a);
    cudaMemcpy(a, dev_a, N*sizeof(int), cudaMemcpyDeviceToHost);

    ...
Single Step of Parallel Merge

```c
__global__ void mergeBlocks(int *a, int *temp, int sortedsize)
{
    int id = blockIdx.x;
    int index1 = id * 2 * sortedsize;
    int endIndex1 = index1 + sortedsize;
    int index2 = endIndex1;
    int endIndex2 = index2 + sortedsize;
    int targetIndex = id * 2 * sortedsize;
    int done = 0;
    while (!done)
    {
        if ((index1 == endIndex1) && (index2 < endIndex2))
            temp[targetIndex++] = a[index2++];
        else if ((index2 == endIndex2) && (index1 < endIndex1))
            temp[targetIndex++] = a[index1++];
        else if (a[index1] < a[index2])
            temp[targetIndex++] = a[index1++];
        else
            temp[targetIndex++] = a[index2++];
        if ((index1==endIndex1) && (index2==endIndex2))
            done = 1;
    }
}
```

temp = device memory
same size as a

sortedsize = length of a sorted “block” (doubles in size from original block)

Main code

```c
int blocks = BLOCKS/2;
int sortedsize = THREADS;
while (blocks > 0)
{
    mergeBlocks<<<blocks,1>>>(dev_a, dev_temp, sortedsize);
    cudaMemcpy(dev_a, dev_temp, N*sizeof(int), cudaMemcpyDeviceToDevice);
    blocks /= 2;
    sortedsize *= 2;
}
```
MergeSort

• With bigger array:
  #define N 1048576
  #define THREADS 512
  #define BLOCKS 2048
• Our implementation is limited to a power of 2 for the number of blocks and for the number of threads per block
• The slowest part seems to be copying the data back to the host, is there anything we can do about that?

Page-Locked or Pinned Memory

• The CUDA runtime offers cudaHostAlloc() which is similar to malloc
• malloc memory is standard, pageable host memory
• cudaHostAlloc() memory is page-locked host memory or pinned memory
  – The OS guarantees it will never page the memory to disk and will reside in physical memory
  – Faster copying to the GPU because paged memory is first copied to pinned memory then DMA copies it to the GPU
• Does take away from total available system memory, may affect system performance
cudaHostAlloc

- Instead of malloc use:

  ```c
  int *a;
  cudaHostAlloc((void **) &a, size, cudaHostAllocDefault);
  ...
  cudaFreeHost(a);
  ```

- Won’t make much difference on our small mergesort but benchmark test with hundreds of copies:
  - Time using cudaMalloc: 9298.7 ms
    - MB/s during copy up: 2753.1
  - Time using cudaMemcpy: 17415.4 ms
    - MB/s during copy down: 1470.0
  - Time using cudaHostAlloc: 6794.8 ms
    - MB/s during copy up: 3767.6
  - Time using cudaMemcpy: 17167.1 ms
    - MB/s during copy down: 1491.2

Zero-Copy Host Memory

- Skipping, but pinned memory allows the possibility for the GPU to directly access host memory
  - Requires some different flags for cudaHostAlloc
  - Performance win if the GPU is integrated with the host (memory shared with the host anyway)
  - Performance loss for data read multiple times since zero-copy memory is not cached on the GPU
Device Query

• How do you know if you have integrated graphics?
  – Can use deviceQuery to see what devices you have
  – cudaGetDeviceCount( &count )
    • Stores number of CUDA-enabled devices in count
  – cudaGetDeviceProperties( &prop, i )
    • Stores device info into the prop struct for device i

Code

```c
#include "stdio.h"

int main()
{
  cudaDeviceProp prop;
  int count;
  cudaGetDeviceCount(&count);
  for (int i=0; i<count; i++)
  {
    cudaGetDeviceProperties(&prop, i);
    printf( "--- General Information for device %d ---\n", i );
    printf( "Name:  %s\n", prop.name );
    printf( "Compute capability:  %d.%d\n", prop.major, prop.minor );
    printf( "Clock rate:  %d\n", prop.clockRate );
    printf( "Device copy overlap: ");
    printf( "Integrated graphics: ");
    if (prop.integrated)
      printf( "True\n" );
    else
      printf( "False\n" );
    if (prop.deviceOverlap)
      printf( "Enabled\n" );
    else
      printf( "Disabled\n" );
```

Using Multiple GPU’s

- Can use cudaSetDevice(deviceNum) but has to run on separate threads
- Fortunately this is not too bad
  - Thread implementation varies by OS
  - Simple example using pthreads
    - Better than fork/exec since threads share the same memory instead of a copy of the memory space

**Thread Sample**

```c
/* Need to compile with -pthread */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct argdata
{
  int i;
  int return_val;
} arg_data;

void *TaskCode(void *argument)
{
  int tid;
  arg_data *p;
  p = (arg_data *) argument;
  tid = (*p).i;
  printf("Hello World! It's me, thread %d!\n", tid);
  p->return_val = tid;
  return NULL;
}

int main()
{
  pthread_t thread1, thread2;
  arg_data arg1, arg2;

  /* create two threads */
  arg1.i = 1;
  arg2.i = 2;
  pthread_create(&thread1, NULL, TaskCode, (void *) &arg1);
  pthread_create(&thread2, NULL, TaskCode, (void *) &arg2);

  /* wait for all threads to complete */
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

  printf("Done, values in return: %d %d\n", arg1.return_val, arg2.return_val);
  return 0;
}
```
Threads with GPU Code

// Using two GPU's to increment by 1 an array of 4 integers,
// one GPU to increment the first two, the second GPU to increment the next two
// Don't need to use -pthread with nvcc

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

typedef struct argdata {
    int deviceID;
    int *data;
} arg_data;

__global__ void kernel(int *data)
{
    data[threadIdx.x]++;
}

void *TaskCode(void *argument)
{
    arg_data *p;
    int *dev_data;

    p = (arg_data *) argument;
    cudaSetDevice(p->deviceID);
    cudaMalloc((void **) &dev_data, 2*sizeof(int));
    cudaMemcpy(dev_data, p->data, 2*sizeof(int), cudaMemcpyHostToDevice);
    kernel<<<1,2>>>(dev_data);
    cudaMemcpy(p->data, dev_data, 2*sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(dev_data);
    return NULL;
}

Main

int main ()
{
    pthread_t thread1, thread2;
    arg_data arg1, arg2;
    int a[4];

    a[0] = 0; a[1] = 1; a[2] = 2; a[3] = 3;
    arg1.deviceID = 0;
    arg2.deviceID = 1;
    arg1.data = &a[0]; // Address of first 2 ints
    arg2.data = &a[2]; // Address of second 2 ints

    /* create two threads */
    pthread_create(&thread1, NULL, TaskCode, (void *) &arg1);
    pthread_create(&thread2, NULL, TaskCode, (void *) &arg2);

    /* wait for all threads to complete */
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    for (int i=0; i < 4; i++)
        printf("%d ", a[i]);
    printf("\n");
    return 0;
}