GPGPU Parallel Merge Sort Algorithm

Jim Kukunas and James Devine

May 4, 2009

Abstract

The increasingly high data throughput and computational power of today's Graphics Processing Units (GPUs), has led to the the development of General Purpose computation on the GPU (GPGPU). nVidia has created a unified GPGPU architecture, known as CUDA, which can be utilized through language extensions to the C programming language. This work seeks to explore CUDA through the implementation of a parallel merge sort algorithm.

1 Introduction

GPGPU allows for the parallel execution of code on the GPU. The GPU is essentially used as a co-processor to handle the execution of highly parallel code. There are very large performance improvements that can be realized by outsourcing highly parallel computationally intense tasks to the GPU.

Typically, Central Processing Units (CPUs) achieve computational parallelism through the use of an out-of-order execution core, which typically contains only a few Arithmetic Logic Units (ALUs). ALUs are co-processors dedicated to integer and floating point computations. For example, three out of the six ports within the execution core of the Intel Core 2 Duo are ALUs. This means that, assuming a full pipeline, three arithmetic computations can execute at once. Graphical Processing Units (GPUs), unlike CPUs, are specialized for intense floating point calculations, and thus contain many stream processors. Stream processors are ALUs which specialize in a limited form of parallelism, similar to SIMD vertex instructions, in which there exists a stream of data and a kernel function which is applied to each element in the stream. These streams are ideal for intensive, highly specialized, parallel floating point operations. The nVidia GTX 260, for example, contains 250 on-board stream

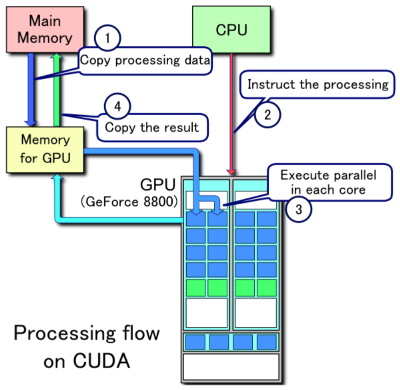

Figure 1: CUDA Architecture

processors. Compared with a CPU, the GPU has a much greater computation throughput, and thus it is ideal to outsource heavy embarrassingly parallel computations onto the GPU. This outsourcing onto the graphics card is known as General Purpose computation on Graphics Processing Units (GPGPU).

Historically, GPGPU architectures were highly specific to certain graphics cards and required the developer to write all GPGPU code in assembly language, assembled specifically for the specific card in use, however with the increase in the number of stream processors, the increase in the amount of on-board memory, and increased pipeline speeds, the demand for GPGPU tools has increased and both nVidia and ATI have released unified GPGPU architectures for their latest cards. nVidia's architecture, CUDA, is a set of language extensions to the C programming language. The CUDA architecture can be seen in figure 1.

2 Extended CUDA C

To allow for simple integration with existing applications, nVidia implemented their GPGPU architecture through extensions to the C programming language. The nVidia CUDA compiler supports full ISO C++ for code running on the host, but for functions executed on the device, only supports CUDA C.

Function Qualifiers

The CUDA extensions for C contain three types of function type qualifiers, `__device`,`__global__` and `__host__`. The `__device__` qualifier declares a function executed completely on the GPU, and hence this function is the most restricted. These functions can not be dereferenced, can not be recursive, can not contain static variable declarations and can not take a variable number of arguments. These functions can also only be called from the device. Functions declared with the `__global__` qualifier defines a kernel function, the function which will be executed on each element in the stream of data. `__global__` functions are executed on the GPU, but are only callable from the host. They also must return void. The third type of qualifier, `__host__`, declares a function which is executed on the host. These functions are only callable from the host.Once a `__host__` function has been declared, it can be called using the following syntax:

$$FUNC <<< Dg, Db, Ns >>> (Parameters);$$

Here, Dg is a three-element tuple specifying the dimensions and size of the grid to be launched. Db is also a three-element tuple specifying the dimension and size of each block. Finally, Ns is of the type `size_t`, and represents the number of bytes in shared memory that is dynamically allocated per block. The number of threads spawned per block can be represented by[1]:

$$threadsperblock = Db.x * Db.y * Db.z$$

Variable Qualifiers

The CUDA extensions for C contain three types of variable qualifiers, `__device__`,`__constant__`, and `__shared__`. Variables declared with the `__device__` qualifier reside within the memory of the device. These variables are in the global memory space and have the lifetime of the application. The `__constant__` qualifier declares a variable similar to `__device__`, but the value of which is constant. The third qualifier `__shared__`, declares a variable which is accessible from all threads[1].

Learning About CUDA and GPGPU

The starting point for the project was to learn about GPGPU and CUDA. The Internet severed as our best resource. We started with the wikipedia entry for general information and then followed the links to more authoritative websites. These links can be found in the References section.

Install the CUDA SDK and Driver

The CUDA SDK and drivers are available for free from nVidia's CUDA ZONE website[2]. After downloading the software we first installed the CUDA drivers. The drivers replace the video drivers that were installed to allow the CUDA extensions to run. Once the CUDA drivers were installed the SDK installer was run. This installed the files required to write code that uses the CUDA extensions. Along with the SDK a package of sample CUDA programs were installed.

Integrate the CUDA SDK into Visual Studio 2008

After the CUDA driver is installed, it is necessary to set the environmental variables in Visual Studio to allow proper integration. Inside of Visual Studio, you must set the bin, include, lib and source paths to that of those installed by the driver and to that of those installed by the SDK.Once that is completed, you must create a custom build rule to allow for the CUDA application to be built by the NVCC compiler.

Porting an Iterative Merge Sort to the CUDA Architecture

Since CUDA does not support recursion, we had to implement an iterative version of merge sort. After a few various attempts, we settled on an implementation we found at [4]. Using existing CUDA examples and the CUDA Programming Guide[1], we created a host function which was passed two input arrays, the first being a list of the numbers to be sorted and the second was the destination array. Both of these arrays were first created and initialized in host memory. Then, using `cudaMalloc` and `cudaMemCpy`, they were shuttled onto the

GPU's dedicated memory.

Inside the host function, the merge function is called. The merge function is a `__device__` function which performs the actual merge sort algorithm on the GPU.

3 Challenges

The two main challenges involved in this project were setting up Visual Studio 2008 and writing the algorithm. After several hours of work we were able to get Visual Studio to build applications using the CUDA API. It was quite frustrating at the time, but after the correct files were added to the Windows PATH variable programs began to compile without any problems. Once we were able to use the API we were faced with the challenge of learning CUDA and then coding the merge sort algorithm. One of the more difficult tasks in writing the algorithm was deciding on how to convert the recursive merge sort into an iterative one. After finding some sample C code for an iterative merge sort online we were able to replicate it using the CUDA API.

4 Future Work

If more time were alloted for this project it would have been interesting to look at the speed up gained by performing merge sort on the GPU. There is much further work that can be done with GPGPU. The increasing speed and decreasing price of GPUs makes them a promising platform for future research. Undoubtedly, better tools and easier extensions to implement GPGPU are not far down the road.

5 Conclusion

The project was both fun and educational. We got a change to learn how to write parallel code that runs on the GPU.

6 Appendix

Iterative Merge Sort

[4]

```
// This is an iterative version of merge sort.
// It merges pairs of two consecutive lists one after another.
// After scanning the whole array to do the above job,
// it goes to the next stage. Variable k controls the size
// of lists to be merged. k doubles each time the main loop
// is executed.
#include <stdio.h>
#include <math.h>
int i,n,t,k;
int a[100000],b[100000];
int merge(l,r,u)
  int l,r,u;
{ int i,j,k;
  i=l; j=r; k=l;
  while (i<r && j<u) {
    if (a[i]<=a[j]) {b[k]=a[i]; i++;}
    else {b[k]=a[j]; j++;}
    k++;
  }
  while (i<r) {
    b[k]=a[i]; i++; k++;
  }
  while (j<u) {
    b[k]=a[j]; j++; k++;
  }
  for (k=l; k<u; k++) {
    a[k]=b[k];
  }
}
```

```
sort()
{ int k,u;
  k=1;
  while (k<n) {
    i=1;
    while (i+k<=n) {
      u=i+k*2;
      if (u>n) u=n+1;
      merge(i,i+k,u);
      i=i+k*2;
    }
    k=k*2;
  }
}
main()
{ printf("input size \n");
  scanf("%d",&n);
/*  for (i=1;i<=n;i++) scanf("%d",&a[i]); */
  for (i=1;i<=n;i++) a[i]=random()%1000;
  t=clock();
  sort();
  for (i=1;i<=10;i++) printf("%d ",a[i]);
  printf("\n");
  printf("time= %d millisec\n",(clock()-t)/1000);
}
```

GPGPU Merge Sort Kernel

```
/*
 * merge_kernel.cu
 *
 *  merge sort Implementation run in parallel threads
 *   on the GPU through the nVidia CUDA Framework
 *
 * Jim Kukunas and James Devine
 *
 */

#ifndef _MERGE_KERNEL_CU_
#define _MERGE_KERNEL_CU_

#define NUM     64



 __device__ inline
   void Merge(int* values, int* results, int l, int r, int u)
 {
 int i,j,k;
   i=l; j=r; k=l;
   while (i<r && j<u) {
     if (values[i]<=values[j]) {results[k]=values[i]; i++;}
     else {results[k]=values[j]; j++;}
     k++;
   }

   while (i<r) {
     results[k]=values[i]; i++; k++;
   }

   while (j<u) {
     results[k]=values[j]; j++; k++;
   }
   for (k=l; k<u; k++) {
     values[k]=results[k];
   }
 }
```

```
__global__ static void MergeSort(int * values, int* results)
{
    extern __shared__ int shared[];

    const unsigned int tid = threadIdx.x;
    int k,u,i;

    // Copy input to shared mem.
    shared[tid] = values[tid];

    __syncthreads();

    k = 1;
    while(k < NUM)
    {
        i = 1;
        while(i+k <=NUM)
        {
            u = i+k*2;
            if(u > NUM)
            {
                u = NUM+1;
            }
            Merge(shared, results, i, i+k, u);
            i = i+k*2;
        }
        k = k*2;
        __syncthreads();
    }

    values[tid] = shared[tid];
}
```

```
#endif
```

GPGPU Merge Sort Driver Application

```
/*
 * merge.cu
 *
 * Tests the merge sort Implementation run in parallel threads
 *    on the GPU through the nVidia CUDA Framework
 *
 * Jim Kukunas and James Devine
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <cutil_inline.h>
#include "merge_kernel.cu"

int main(int argc, char** argv)
{
        if( cutCheckCmdLineFlag(argc, (const char**)argv, "device") )
                cutilDeviceInit(argc, argv);
        else
                cudaSetDevice( cutGetMaxGflopsDeviceId() );

    int values[NUM];

    /* initialize a random dataset */
    for(int i = 0; i < NUM; i++)
    {
        values[i] = rand();
    }
    int* dvalues,
       * results;
    cutilSafeCall(cudaMalloc((void**)&dvalues, sizeof(int) * NUM));
    cutilSafeCall(cudaMemcpy(dvalues, values, sizeof(int) * NUM, cudaMemcpyHostToDevice
    cutilSafeCall(cudaMalloc((void**)&results, sizeof(int) * NUM));
    cutilSafeCall(cudaMemcpy(results,values , sizeof(int)* NUM, cudaMemcpyHostToDevice)

    MergeSort<<<1, NUM, sizeof(int) * NUM*2>>>(dvalues, results);

    // check for any errors
    cutilCheckMsg("Kernel execution failed");

    cutilSafeCall(cudaFree(dvalues));
    cutilSafeCall(cudaMemcpy(values, results, sizeof(int)*NUM, cudaMemcpyDeviceToHost))
    cutilSafeCall(cudaFree(results));

    bool passed = true;
    for(int i = 1; i < NUM; i++)
    {
        if (values[i-1] > values[i])
        {
            passed = false;
        }
    }
    printf( "Test %s\n", passed ? "PASSED" : "FAILED");
    cudaThreadExit();
    cutilExit(argc, argv);
}
```

References

[1] NVIDIA Corporation. Nvidia cuda programming guide. *NVIDIA*, 2008. `http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf`

[2] NVIDIA Corporation. Cuda zone. *NVIDIA*, 2009. `http://www.nvidia.com/object/cuda_home.html`

[3] Mark Harris. General-purpose computation on graphics hardware. *GPGPU.org*, 2009. `http://gpgpu.org`

[4] Tadao Takaok. Iterative merge sort. `http://www.cosc.canterbury.ac.nz/tad.takaoka/cosc229/imerge.c`