Different Optimization Strategies and Performance Evaluation of Reduction on Multicore CUDA Architecture

Chhaya Patel
CE/IT Department, School of Engineering
RK University
Rajkot, India

***Abstract*—the objective of this paper is to use different optimization strategies on multicore GPU architecture. Here for performance evaluation we have used parallel reduction algorithm. GPU on-chip shared memory is very fast than local and global memory. Shared memory latency is roughly 100x lower than non-cached global memory (make sure that there are no bank conflicts between the threads). We have used on chip shared memory for actual operations. One of the most common tasks in CUDA programming is to parallelize a loop using a kernel. To efficiently parallelize this, we need to launch enough threads to fully utilize the GPU. Instead of completely eliminating the loop when parallelizing the computation, we used grid-stride loop. By using grid stride loop we can process more data elements with limited number of threads. Grid stride loop provide scalability and thread reusability. A different strategies used to optimize parallel reduction algorithm. Here in this paper we tried to compare time and bandwidth of each version. In next each version we observed significant improvement.**

Keywords—GPU, CUDA, Multicore, Shared Memory, Multi threading, parallel reduction

I. INTRODUCTION

In the past few years new multicore heterogeneous architectures have been introduced in high-performance parallel computing. Examples of such architectures include Graphics Processing Units (GPUs). GPUs are small devices with hundreds of computing cores which are designed for high performance computing. GPU computing is the use of a GPU (graphics processing unit) as a co-processor to accelerate CPUs for general purpose scientific and engineering computing. The GPU accelerates applications running on the CPU by offloading some of the compute intensive and time consuming portions of the code. The rest of the application still runs on the CPU. From a user's perspective, the application runs faster because it is using the massively parallel processing power of the GPU to boost performance. This is known as heterogeneous or hybrid computing architecture. GPUs usually contribute to the overall computation in two different ways: carrying out some specific tasks through user-designed kernels or executing some data-parallel primitives provided by a growing number of libraries (e.g. CUDPP, CLPP, GPULib, Thrust…). GPU provide different types of memory for application. We can boost performance accordingly application need.

When we implement any algorithm with GPU then it will be definitively faster than CPU. Here we are not compare CUP and GPU code, instead of that we tried to compare different version of GPU code only.

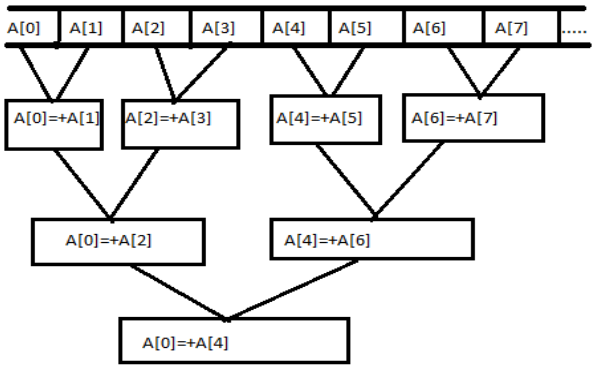
In this paper we tried to optimize the basic parallel reduction algorithm using multicore heterogeneous architecture known as compute unified device architecture (CUDA).

Summarize a set of input values into one value is called reduction. For example sum of 1 to N numbers, finding MAX or MIN from list, average of numbers given in array. In this paper we take sum of 1 to N number as reduction algorithm. It is most important that which operator is used in parallel reduction. The operator used in algorithm that must be a binary and associate. + Operator is binary and associate. Algorithm will take array of N data [A [0], A [1]…A [N-1]] as input and will product one value as output. Here we compared different parallel version of code. We used five different strategies for optimization of parallel code.

Serial version of reduction is very straight forward. In serial reduction, each iteration is dependent on the previous iteration. We have a serial variable named sum, we initialize it to zero. We then loop throughout set of elements and on each iteration. Add the current element to the previous sum. So, in the first iteration, we do this first add here, and we take the result on the second iteration. Do a second add here on the third iteration, we do a third add, and so on. So, that this adds operation is dependent on the previous one. So complexity of basic reduction is O (N).

Here we used tree based structured for parallel reduction given in figure 1. We done like that by first computing A[0] plus A[1], and perhaps at the same time, computing A[2] plus A[3] and then add the results together. We done to do parallel reduce, is regrouping these operations in a different order, and this exposes more concurrency. We now have the ability to run multiple operations in parallel at the same time. In tree based structure, if we have N elements for reduction than N/2 operations will do concurrently. Results of N/2 operations again reduce in second step so in second step N/4 operation will performed concurrently. So potentially, we can run this with parallel hardware, and it will complete faster. Parallel implementation time complexity is O (logN).

Figure: 1: Tree based version of parallel reduction.



Main aim of this paper is to compare different parallel versions of reduction. We are comparing two parameters one is time and second is bandwidth. Performance measured on GforceGTX 480.
GeForceGTX 480 having 15 Multiprocessor and each multiprocessor having 32 cores so total 480 Cores.

II. RELATED WORK

The kernels presented by Harris [2] are the most popular CUDA implementations for the reduction primitive. They are actually included as project examples in every CUDA SDK release. His document introduces seven kernels from a didactic perspective, in such a way that each kernel improves the performance of the previous one. Nevertheless, many of them require the operator to be commutative.
The kernels presented by Pedro [1] are represented segmented version of CUDA implementations for the reduction. In this *segmented* version, the input array is divided into segments of consecutive data, and the output is the individual reduction of each segment. Thus, the output size is the number of segments included in the given input. Author observed that the segmented primitive could be easily implemented in terms of the unsegment primitive, by extracting each segment and reducing it, isolated from the global input, with unsegment primitive.

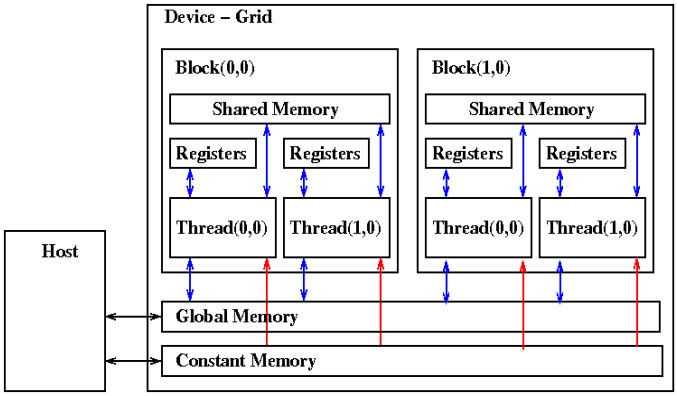
The different optimization strategies used by Shane Ryoo [3]. For optimization author considered manage resources include the number of register, and amount of memory used per thread block.
They also obtained increased performance by reordering access to on chip memory and applied classical optimizations to reduce the number of executed operations. They applied these strategies across various applications and achieved 10X to 450X speedup in different kernel codes.

In this paper [4] author presented different CUDA architectures, including Fermi, and optimized a set of algorithms for each using widely-known optimization techniques. They used fast prototyping tool, for an effortless process. The result of analysis can guide researchers on the right path towards efficient code optimization. Preliminary results show that some optimizations recommended for older CUDA architectures may not be useful for the newer ones.

III ARCHITECTURE OVERVIEW

The CUDA (Compute Unified Device Architecture) programming model used in NVIDIA GPU architecture. GPU (Graphics Processing Unit) works as co-processor with CPU. Basically program execution start with CPU and computational intensive parts of the code execute with GPU and still rest of the serial code execute with CPU. In this way we can get higher performance of algorithms. CUDA programming model support different types of memories. By default when we transfer data from CPU to GPU it will be store in global memory. Global memory is off chip memory. GPU performance is influenced by the architectural organization of the hardware platform. NVIDIA suggests that achieving the highest GPU occupancy and optimizing the use of the memory hierarchy are the two main factors behind GPU performance [7]. In fact, both of them are related since maximizing the occupancy can help to cover latency during global memory loads. We present several experiments aimed at analysing their relative importance. Our results indicate that code that target efficient memory usage are the major determinant of actual performance. Overall, they ensure the best performance even if some resources remain unutilized. Therefore, maximizing occupancy should be examined at a later stage in the compilation process, once data related issues have been properly addressed. Figure 2 shows memory hierarchy of CUDA architecture. To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module. Here we tried to get maximum benefit of shared memory.

Figure: 2: Memory hierarchy of CUDA architecture.



CUDA Execution model consist of Grid, thread blocks, and Threads. Entire grid is handled by a single GPU chip. The GPU chip is organized as a collection of multiprocessors (MPs), with each multiprocessor responsible for handling one or more blocks in a grid. A block is never divided across multiple MPs. Each MP is further divided into a number of stream processors (SPs), with each SP handling one or more threads in a block.
NVIDIA CUDA SDK has been designed for running parallel computations on the device hardware: it consist of a compiler, host and device runtime libraries and a driver API. CUDA software stack is composed of several layers: a hardware driver (CUDA Driver), an API and its runtime (CUDA Runtime), two higher-level mathematical libraries (CUDA Libraries) of common usage.

IV OPTIMIZATION STRATEGIES

Very first we will transfer data from CPU to GPU. When we transfer data by default it will be load in global memory. We are using 32MB data for reduction.

```
cudaMemcpy(d, hIn, size, cudaMemcpyHostToDevice);
```

Version 1: In first parallel reduction shared memory is used so required again data transfer from global memory to shared memory. Then operation will be performed with shared memory. Kernel will be called with dynamic shared memory.

```
version1<<<dimGrid,dimBlock,smemSize>>>(d);
```

Shared memory is used for operation so actual code having two parts. One is load shared memory and second is do actual reduction in shared memory.

```
__global__ void version1(int *d)
{
extern __shared__ int sm[];
//First part:
// load shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sm[tid] = d[i];
__syncthreads();
//second part:
// do reduction in shared mem
for(unsigned int stride=1;stride<blockDim.x;stride*=2)
{
if ((tid % (2*stride)) == 0) sm[tid]+=sm[tid+stride];
__syncthreads();  }
// write result for this block to global mem
if (tid == 0) d[blockIdx.x]=sm[0];  }
```

First version is used Interleaved Addressing.
Stride grid loop is used to process data elements with threads. In first stride (step 1) N/2 elements will be added. Interleaved threads will added. Same process recursively repeat until all data add with threadIdx 0. Partial result will save with threadIdx 0 for all blocks. Now last step is transfer partial result to global memory and then do final sum with global memory.

Version 2: In version 1 branch divergence occurs due to interleaved branch decisions. Due to branch divergence some of the threads will do task and some of them idle, so threads may perform work serially which leads to performance loss. Version 2 is implemented with stride index and non-divergent branch.

```
// do reduction in shared memory
for(unsigned int stride=1; stride<blockDim.x; stride*=2)
{
int index=2*stride*tid;
if (index<blockDim.x)
sm[index]+=sm[index+stride];
__syncthreads();
}
```

Above code use stride index, remove branch divergence and remove % operator so code is optimized.

Version 3: Actual operation doing with shared memory so in version 2, there is bank conflict which leads to performance loss. Version 3 implemented with linear addressing that remove bank conflicts.

```
// do reduction in shared mem
for (unsigned int
stride=blockDim.x/2;stride>0;stride>>=1)
{
if (tid<stride)
sm[tid]+=sm[tid+stride];
__syncthreads();
}
```

Version 4: Here in version4 reduction portion as it is optimized in version 3, instead of that load shared memory data portion is optimized. Here first operation does with global memory and result only transfer to the shared memory. So only half of the threads with used shared memory and rest of reduction operation as it are used. With this optimization half of threads will reduce in shared memory so we can get maximum use of threads.

```
// With two loads and first add of the reduction
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2)+threadIdx.x;
sm[tid] = d[i]+d[i+blockDim.x];
__syncthreads();
```

Version 5: in this version again just optimized do reduction in shared memory portion of code. With two loads and first add of the reduction code we just used that is last optimized**.** Here loop unrolling technique is used. By this technique no. of iterations is reduced and ultimately code is optimized.

```
// do reduction in shared mem
for (unsigned int s
{ if (tid<stride) sm[tid]+=sm[tid+stride];
__syncthreads(); }
if (tid < 32)
{ sm[tid]+=sm[tid+32];
sm[tid]+=sm[tid+16];
sm[tid]+=sm[tid+8];
sm[tid]+=sm[tid+4];
sm[tid]+=sm[tid+2];
sm[tid]+=sm[tid+1];
}
```

V COMPARISON OF DIFFERENT OPTIMIZATION STRATEGIES

Figure: 3: Time measurement of different versions.

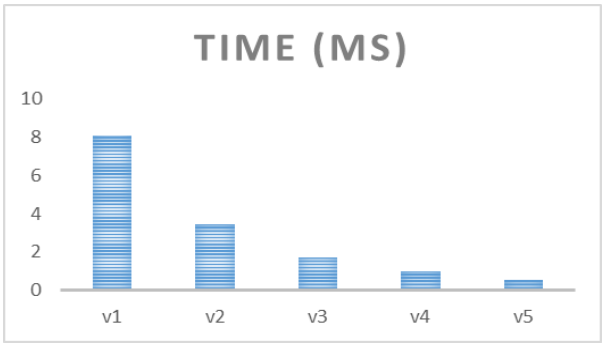
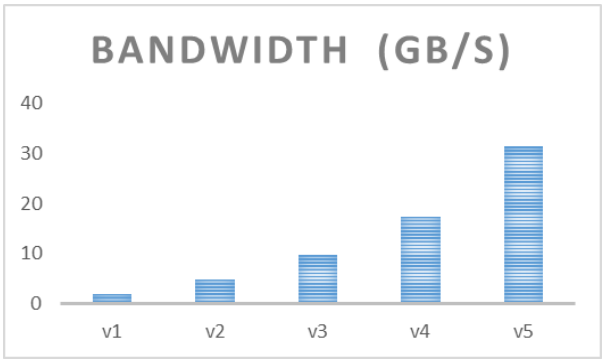


Figure : 4 : Bandwidth measurement of different versions.



VI CONCLUSION

Parallel approaches have a better performance than serial dependence reduction. With regards of shared memory usage we can get higher performance. When using shared memory should take about bank conflict. Bank conflict may leads performance overhead. Loop unrolling is good optimization technique just by rearranging loop structure. This work presents general principles for optimizing application with CUDA architecture. We also present an application suite that has been ported to shared memory.

REFERENCES

1. Pedro J. Martín, Luis F. Ayuso, Roberto Torres, Antonio Gavilanes , "Algorithmic Strategies for Optimizing the Parallel Reduction Primitive in CUDA," 978-1-4673-2362-8/12, ©2012 IEEE.
2. M. Harris, Optimizing Parallel Reduction in CUDA, (2007), http://developer.download.nvidia.com/compute/cuda/1_1/Website/projec ts/reduction/doc/reduction.pdf.
3. Shane Ryooy Christopher I. Rodriguesy Sara S. Baghsorkhiy Sam S. Stoney, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA", Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign NVIDIA Corporation.
4. Ruymán Reyes , Francisco de Sande, "Optimization strategies in different CUDA architectures using llCoMP", Microprocessors & Microsystems, v.36 n.2, p.78-87, March, 2012 [doi>10.1016/j.micpro.2011.05.006]
5. Javier Setoain1, Christian Tenllado1, Manuel Arenaz, and Manuel Prieto1, "To wards Automatic Code Generation for GPU architectures", Computer Architec- ture Group, Department of Electronics and Systems, University of A Coruna, Spain.
6. NVIDIA CUDA. http://developer.nvidia.com/object/cuda.html.
7. J. Nickolls and I. Buck. NVIDIA CUDA software and GPU parallel computing architecture. Microprocessor Forum, May 2007
8. K. Kennedy and J. R. Allen. Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publishers Inc., 2002.
9. S. Sengupta, M. Harris, HM. GarlandH, "Efficient Parallel Scan Algorithms for GPUs," H NVIDIA TR NVR-2008-003H, Dec. 2008.
10. S. Sengupta, M. Harris, Y. Zhang, and J. Owens, "Scan Primitives for GPU Computing," Proc. Graphics Hardware (GH 07), ACM, 2007, pp. 97-106.