Parallel Implementation of the Modified Subset Sum Problem in OpenCL

Dushan Petkovski[1], Igor Mishkovski[2]

[1,2] Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University
Skopje, R. Macedonia
[1] dushan.petkovski@gmail.com
[2] igor.mishkovski@finki.ukim.mk

Abstract. The rapidly changing capabilities of modern graphics processing units (GPUs) give developers the opportunity to implement parallel programming techniques, or even combine the traditional (single process) programming with parallel programming. As the computing is shifting from central processing (on the CPU) to co-processing (on the CPU and GPU), platforms have emerged and gave the developers opportunity to experiment. In this work, a parallel solution of the modified subset sum algorithm is implemented using OpenCL, and the obtained speedup is compared to the CPU version and other parallel implementations of the problem.

Keywords: Parallel computing, Subset Sum, OpenCL, GPU

1 Introduction

A graphics processing unit (GPU) is a single chip processor used primarily for 3D applications. It creates lighting effects and transformations of objects when a 3D scene is redrawn. As these are intensive tasks to process they can overwhelm the CPU, so using the GPU to process them, the CPU could be used for other jobs [1].

The performances using parallel programming compared to the performances of the traditional way are the main reason why the parallel programming is in upswing. The ratio between many-core GPUs and multi-core CPUs for floating-point calculation throughput is about 10 to 1 and the speed that can be supported in these chips are 1000 gigaflops (GPU) versus 100 gigaflops (CPU) [2]. The large ratio between the GPUs and CPUs lies in the design of the two types of processors, as shown in Fig. 1 [3].

S. Loshkovska, S. Koceski (Editors): ICT Innovations 2015, Web Proceedings, ISSN 1857-7288

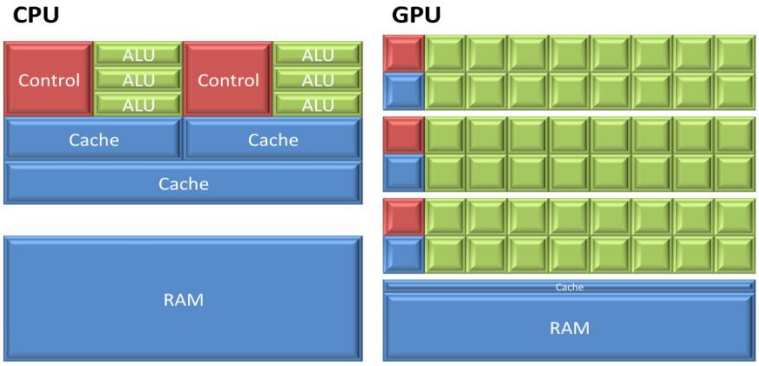

Fig. 1. CPU vs GPU design

Execution of algorithms with high complexity, algorithms with large linear iterations, or a combination of both can be fairly long if it's done sequentially. Applications that have independent elements and don't require synchronization can have great benefits from GPU execution. For instance, in computer science the subset sum problem is an important problem in complexity theory and cryptography. The problem includes a set of integers, and determines whether there is a non-empty subset whose sum is zero. For instance, given the set {−4, −3, −2, 5, 20}, the answer is the subset {−3, −2, 5} which sums to zero. The problem we will work on is a similar problem, with an additional input: given a set of integers and an integer *SUM*, does any non-empty subset sum to *SUM*. For more detailed explanation see [4].

In this work, we describe an efficient parallel algorithmic implementation of a modified version of the subset sum problem. The problem we are solving is counting how many vectors with *N* elements smaller than *K* add up to a number *S*, where the variables (*N*, *K*, and *S*) are input variables. This problem can also be thought as a special case of the knapsack problem [5], and is also NP-complete problem. It has practical usage in job scheduling, workload allocation, and also in cryptography [6].

Not only that the algorithm counts the vectors, but it also returns them, which is important, because mathematical approximations can be made to count the vectors that satisfy the condition [7], [8]. The main characteristic of the approach covered in this paper is that it requires significantly less space. All the memory is dynamically allocated, which will be explained in full detail in Section 3.

One field where this is practical solution is for a peer-assisted Video-on-Demand streaming. The peers host certain videos which they stream to other peers through the network. These clients send their requests to the index server, and it checks the availability of streaming resources on the peers that contain a copy of that content. If the index server finds a peer that has copy of the content and enough uplink capacity, it assigns the peer to serve the request. If there is none, the server itself is serving the request [4]. The solution can be used in a situation where the system has *N* peers that

have uplink capacity to simultaneously stream *K* streams, and specific vectors should be returned.

Section 2 gives more information on the OpenCL standard for parallel programming. In Section 3, the parallel implementation of the algorithm is explained in detail, whereas Section 4 concludes this work.

2 OpenCL

OpenCL (Open Computing Language) is the first open standard for parallel programming of modern processors found in personal computers. OpenCL greatly improves speed and responsiveness for a wide spectrum of applications in various categories. These categories vary from gaming and entertainment to scientific and medical software [9]. It's a new framework for writing programs that execute in parallel on different compute devices (CPU / GPU) from different manufacturers. The framework defines a language to write functions, called kernels, which run on different compute devices [10].

According to websites that introduce users with OpenCL, despite its advantages, OpenCL is not easy to learn, since it's not derived from any distributed computing framework. Its concepts are similar to NVIDIA's CUDA, but the syntax, the data structures and the functions are unique. In Table1 we show the terminologies used in CUDA (left), and those used in OpenCL (right).

CUDA Terminology	**OpenCL Terminology**
GENERAL TERMINOLOGY	
Thread	Work item
Thread block	Work group
Global memory	Global memory
Constant memory	Constant memory
Shared memory	Local memory
Local memory	Private memory
QUALIFIERS FOR KERNEL FUNCTIONS	
__global__ (callable from host)	__kernel (callable from device)
__device (not callable from host)	
__constant (variable declaration)	__constant (variable declaration)
__device__ (variable declaration)	__global (variable declaration)
__shared__ (variable declaration)	__local (variable declaration)

Table1. CUDA vs OpenCL Terminology

The first step in developing an OpenCL project is to write the host application. The host application runs on a computer and it forwards the kernels to the devices. The

S. Loshkovska, S. Koceski (Editors): ICT Innovations 2015, Web Proceedings, ISSN 1857-7288

main thing regarding the host application is setting up the kernels and devices, which is accomplished by using five data structures: cl_device_id, cl_kernel, cl_program, cl_command_queue, and cl_context.

- **Device**: OpenCL devices receive kernels from the host. The devices are represented by cl_device_id.
- **Kernel**: A host application distributes kernels to devices. The kernels are represented by cl_kernel.
- **Program**: The host selects kernels from a program. A program is represented by a cl_program.
- **Command queue**: Each device receives kernels through a command queue. In code, a command queue is represented by a cl_command_queue.
- **Context**: An OpenCL context allows devices to receive kernels and transfer data. In code, a context is represented by a cl_context.

The host application can be written in C or C++.

According to [11], an OpenCL host application is like a game of cards: "In a card game, a dealer sits at a table with one or more players and distributes cards from a deck. Each player receives these cards as part of a hand and then analyzes how best to play. The players can't interact with one another or see another player's cards, but they can make requests to the dealer for additional cards or a change in stakes. The dealer handles these requests and takes control once the game is over." Fig. 2 shows how the data structures work with the host application.

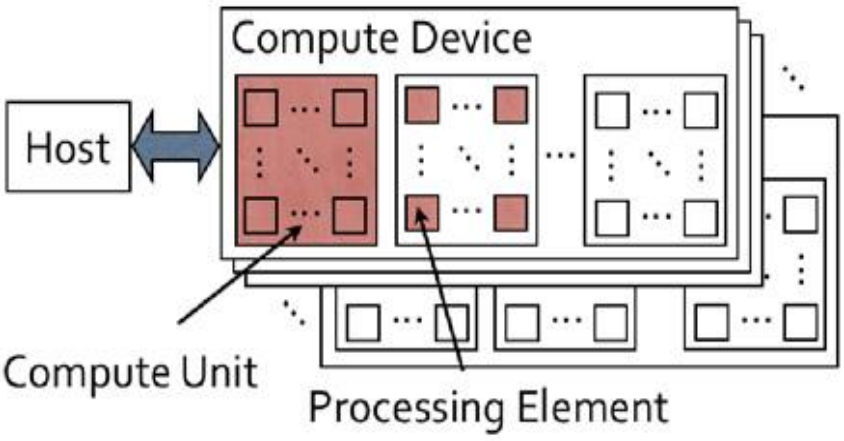

Fig. 2. OpenCL application with its data structures

While CUDA programs function only on NVIDIA and need to be rewritten for other platforms, OpenCL programs can run on hardware from different manufacturers. These include Intel, AMD, NVIDIA, and IBM. OpenCL kernels can run on dif-

ferent types of devices too, on GPUs and CPUs. This is one of the main reasons of choosing OpenCL. Not only that, but a single application can forward kernels to multiple devices. For instance, if a computer contains an AMD Fusion processor and an AMD graphics card, you can synchronize kernels running on both devices and share data between them. OpenCL kernels can even be used to accelerate OpenGL or Direct3D processing [11].

3 Parallel implementation of the modified subset sum problem in OpenCL

In this section, the parallel algorithm used to solve the modified subset sum problem will be explained in details. The solution is to find (count and return) all the vectors with *N* elements, each element smaller than *K*, and their (the elements') sum is equal to *SUM*. *N*, *K* and *SUM* are user inputs.

The mathematical formulation of the modified subset sum problem that this work explores is as follows: $A = a_1, a_2, \ldots, a_n$ is an input vector with $|A| = N$, the algorithm triesto find all vector elements a_i, where $i = 1 \ldots N$, under the following cconstraints: $a_i \in [0, K]$ and $\sum_i^N a_i = s$, for every i.

Because the number of permutations for larger *N* and *K* is fairly big, the computation time needed to find the vectors is going to infinite. That's why we will parallelize this process using OpenCL.

The main difference between this and other algorithms [5] is that finding the vectors with the way we'll explain requires significantly less space. Usually there is a predefined maximum regarding the length of the vectors, and the memory used for the parallel execution is statically allocated, i.e. the maximum multiplied by the number of work items per work group (and most often a greater part of the allocated memory remains unused). Also, the number of work items per work group and the number of work groups (number of threads per block and number of blocks in CUDA) is usually predefined, and the logic of finding the permutations and checking the conditions is adapted to these numbers.

In our approach, all the parameters that we need are dynamically allocated (as will be explained in the next paragraphs in detail), which allows the program to run without using more memory than it needs to. Also, the number of work groups (blocks) and work items (threads) are set to do optimal amount of work depending on the user input (the length of the vector and the maximal value of an element).

Thus, before anything happens, the user is asked to enter values for *N*, *K*, and *SUM*. *N* is the size of the vector, *K* is the maximal value for the elements, and the *SUM* is the value to which all the elements in the vector should add up to for us to return the vector. As mentioned before, the number of processes and elements per process are set based on this input. So, we set the number of processes to *K*, and the number of elements per process to *N*. The number *SUM* doesn't affect the behavior of

S. Loshkovska, S. Koceski (Editors): ICT Innovations 2015, Web Proceedings, ISSN 1857-7288

the program, since it's used just to check if a condition is true or not. Next, we define *LIST_SIZE*, which is equal to *ELEMENTS_PER_PROCESS* (*N*) multiplied by *PROCESSES* (*K*), which is the memory we will use. So, we define:

```
int *A = (int*)malloc(sizeof(int)*LIST_SIZE);
```

Next, we get the platform and device information, we create an OpenCL context, command queue and memory buffer for the vector. After creating the OpenCL kernel, we set its arguments (*A, ELEMENTS_PER_PROCESS, PROCESSES, SUM*), and we execute the kernel on the list, after dynamically setting the global and local item size:

```
...
        // Create an OpenCL context
        cl_context context = clCreateContext( NULL, 1,
&device_id, NULL, NULL, &ret);

        // Create a command queue
        cl_command_queue command_queue = clCreateCommand-
Queue(context, device_id, 0, &ret);
...
        // Create a program from the kernel source
        cl_program program = clCreateProgramWithSource(context,
1, (const char **)&source_str, (const size_t *)&source_size,
&ret);

        // Build the program
        ret = clBuildProgram(program, 1, &device_id, NULL, NULL,
NULL);

        // Create the OpenCL kernel
        cl_kernel kernel = clCreateKernel(program, "doEvery-
thing", &ret);
        size_t global_item_size = LIST_SIZE; // The entire list
        size_t local_item_size = PROCESSES; // Divide work items
into groups
        ret = clEnqueueNDRangeKernel(command_queue, kernel, 1,
NULL, &global_item_size, &local_item_size, 0, NULL, NULL);
```

The code that is executed here is completely parallel, and after its execution the results are displayed, and we are releasing the kernel and memory objects and also freeing the allocated memory. The kernel code is displayed and explained next.

```
__kernel void doEverything(__global int *a, __global int *rez,
int elements, int k, int sum, __global int *retV);
```


S. Loshkovska, S. Koceski (Editors): ICT Innovations 2015, Web Proceedings, ISSN 1857-7288

The "__kernel" qualifier tells us that this function can be called from the device (from the "host" in CUDA), and "__global" is used for variable declaration. The variables other than *N, K,* and *SUM* are used for counting the vectors that satisfy the condition in each process. The integer e*lements* is the number of elements that will be processed by the thread (*N*).

The reason why we want to parallelize the code for finding the vectors that satisfy the condition is because the number of permutations gets fairly big as *N* and *K* increase (which is the main reason for the long execution). The idea behind our code is to minimize the number of permutations we need to check in each thread, so we'll have faster execution of the code, and also without allocating redundant memory. The number of permutations we need to check is $K^{elements}$. So, as we multiply *K* by itself, every next product has greater and greater (negative) impact on the speed of execution of the code. In other words, if we can decrease the exponent even by 1, we would have much faster execution. This difference is noticeable even for the smallest *N* and *K*, and scales with their value, since we're looking at an exponential growth.

The way we accomplish the minimization of permutations is by using the vector *A*, for which we allocated memory equal to *N*K* (or *elements*k*). We get the *global id* for each process, which is a unique incremental integer that identifies each process. The reason to use *N*K* elements for our vector (*A*) is because we will give each process *N* elements to work with, while using *K* processes, and the first element of the vector for each process will be set to its *global id*. The way we get to the first element for each process from the vector *A* is by multiplying the respective *global id* with *elements* (the number of elements per process).

```
int i = get_global_id(0); // Get the index of the current ele-
ment to be processed
        a[i*elements] = i;
```

In this way, each process no longer needs to go through all the permutations to find the vectors that satisfy the condition, but through an exponentially smaller value, i.e. $K^{elements-1}$. In this way, each process only iterates through a small part of *A*, i.e. from *A[global_id*elements]* to *A[elements * (global_id + 1) - 1]*, sums up the elements, and checks if their sum is equal to *SUM*. The code that checks the condition and counts the vectors is displayed.

```
tempSum = 0;
 for (j = 0; j < elements; j++) tempSum += a[i*elements+j];

 //check if the elements add up to SUM
 if (tempSum == sum)
      rez[i]++;
```


S. Loshkovska, S. Koceski (Editors): ICT Innovations 2015, Web Proceedings, ISSN 1857-7288

After each cycle, a value of *A* is incremented and the conditions are checked again and again in the *for* loop. The parallel increment of each part of *A* by its respective process is shown below.

```
j = 1; finish = 0;
while (!finish)
{
     a[i*elements+j]++;
     if (a[i*elements+j]>=k)
     {
            a[i*elements+j] = 0;
            j++;
            if (i*elements+j >= i*elements+elements)
               kraj = 1;
      }
      else finish = 1;
}
```

We are incrementing the values starting from the 2nd element, since the initial value for *j* is 1. The first element of the list remains unchanged for each process. By using dynamically allocated memory equal to *N*K*sizeof(int)* we get the results with the adequate speed up. The speed up that we get from the parallel execution of the code scales with *N* and *K*. Since the time for iterating through all the permutation is exponential, the bigger the values for *N* and *K*, the greater the speed up will be, after we minimize the exponent. Practical case where this can be used is in peer-assisted Video-On-Demand streaming, as explained in the introduction section. A few test cases are shown in Fig. 3 and Fig. 4 (the Y axis is in seconds).

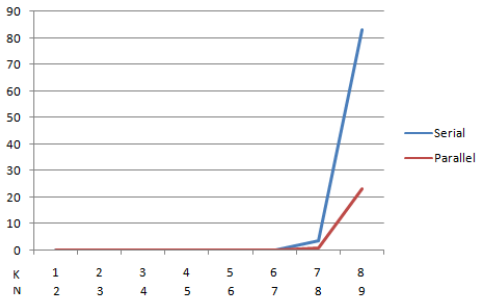

Fig. 3. The execution time for N & K up to 8 and 9

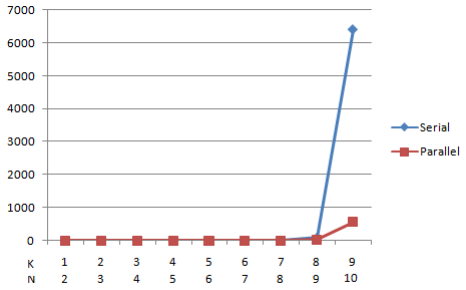

Fig. 4. The execution time for N & K up to 9 and 10

Both Fig. 3 and Fig. 4 are representing the execution time for the same algorithm (same scenario with an additional step in Fig. 4), but are split into 2 pictures because of the large difference in execution time in the next step (i.e. the execution time when

N & K are 9 & 8 in Fig. 4 looks similar, but in Fig. 3 we can clearly see the difference, which shows the exponential grow in the execution in every next step.)

Regarding the speed up, we can see that for the minimal values for *N* and *K* (between 2 and 5) there is not much difference in the time of execution, since the serial and the parallel execution time is virtually equal, but for values around 5 or greater we can already notice the speed up. For example, if *N* and *K* are 7 and 6, the parallel execution time is about 3 times faster than the serial (0.06s / 0.18s). When *N* and *K* are 9 and 8, the parallel execution time becomes 4 times faster (~20s / ~80s), and with the next case (10 and 9) it's already more than 10 times faster (~9min / >1.5h), which increases as *N* or *K* increase.

4 Conclusion

With great power comes great complexity. OpenCL, with its wealth of features, makes it possible to code routines capable of executing on devices ranging from graphics cards to supercomputers. The modern GPU is a massively parallel processor and with the help of OpenCL and CUDA programming model we are able to write scalable parallel programs to execute on GPU.

In this work, we have parallelized a modified version of the popular subset sum problem using OpenCL. By dynamically allocating memory, we minimized the memory used to solve the problem, and make an optimal number of work items execute in optimal number of work groups, executing optimal amount of work. Also, we achieved a speed up that largely scales with the problem size, and significantly decreases the time needed for execution for larger *N* and *K*.

References

1. "GPU" - http://www.webopedia.com/TERM/G/GPU.html
2. D.B. Kirk and W-m W. Hwu, "Programming Massively Parallel Processors", Morgan Kaufmann 1st edition, Feb. 2012.
3. Zlate Ristovski, Igor Mishkovski, Sasho Gramatikov, "Parallel Implementation of the Modified Subset Sum Problem in CUDA"
4. Soumendra Nanda, CS 105: Algorithms (Grad) Subset Sum Problem March 2, 2005
5. Martello, Silvano; Toth, Paolo (1990). "4 Subset-sum problem". Knapsack problems: Algorithms and computer interpretations. Wiley-Interscience
6. L. Wan, K. Li, J. Liu and K. Li, "A Novel CPU-GPU Cooperative Implementation of A Parallel Two-List Algorithm for the Subset-Sum Problem", Proceeding PMAM'14 Proceedings of Programming Models and Applications on Multicores and Manycores, 2014.
7. M. Fischetti, "Worst-case analysis of an approximation scheme for the subset-sum problem"
8. Nei Yoshihiro Soma, Alan Solon Ivor Zinober, Horacio Hideki Yanasse, Peter John Harley, "A polynomial approximation scheme for the subset sum problem"
9. Khronos Group – "The open standard for parallel programming of heterogeneous systems"

10. Erik Smistad – “Getting started with OpenCL and GPU Computing”
11. Dr.Dobb – “A Gentle Introduction To OpenCL”

S. Loshkovska, S. Koceski (Editors): ICT Innovations 2015, Web Proceedings, ISSN 1857-7288