DOI: http://dx.doi.org/10.26483/ijarcs.v8i8.4738

ISSN No. 0976-5697

Volume 8, No. 8, September-October 2017

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

A NOVEL AUTOMATIC C TO NVIDIACUDA CODE OPTIMIZATION FRAMEWORK

Chennupalli Srinivasulu
Research Scholar, JNTU, Kakinada
JNTU Kakinada, AP, India.

Dr. Niraj Upadhyay
Dean and Professor of CSE,
JBIT, Hyderabad, TS, India.

Dr A.Govardhan
Principal, JNTU Hyderabad,
TS, India

Abstract: With the continuous demand for high performance computing, the need for reducing time for executing the application is the current challenge of research. Nevertheless, the execution time for the application not only depends on the hardware or architecture, rather also depends on the algorithm design. Improvement of the hardware may lead to higher investments and the optimization of cost is also to be taken care. Henceforth, the major optimization task is to focus on the algorithm design. A number of algorithm design techniques are available and techniques have reached the maximum of optimization levels. Thus, not limiting to the improvement in the algorithm design, the use of parallel execution of the programs is also to be considered. GPUs are commonly used processing units to speed up the application execution in the domain of game development. The GPUs can be utilized to parallelize the application execution to reach the clock usage to the maximum. The major challenge is to design or re-design the application code from traditional serial programming languages to the parallel codes, which can take the advantages of GPU cores. Nonetheless, the code conversion is not easy and demands a higher understanding of parallel programming and the GPUs are transparent to understand for a beginner. Thus the final demand for the application development industry is to build a code conversion framework to automatically convert the source code into parallel programs. This work presents a novel C to NVIDIA Cuda code converter and gives the legacy programs a chance to run on parallel architecture. This work, to be presented, can be considered as a base line for further reach and be used for bench marking the applications. The results demonstrate a high reduction in execution time.

Keywords: Code Conversion, Parallel Execution, GPU, CPU, CUDA, NVIDIA, CUDA Stack

1. INTRODUCTION

The evaluations of the GPUs are primarily caused by the enhancement of high graphics in the game development industry and scientific applications demanding more processing capabilities. The 3D rendering of the graphics modules of the games need the highly parallel and programmable pipelined processors. These can deliver parallel execution in significantly low cost. The measures of the performance of graphics processing units are completely taken over the performance of the central processing units. The notable works by Shane Ryoo et. al. [1] have demonstrated the improvements of execution time for multi-threaded applications on GPUs compared to the CPUs. The surprising improvements of reducing execution time have forced multiple research organizations and processor architecting industries to build more sophisticated GPUs for general purpose floating point conversions and calculations. R. Kresch et al [2] have demonstrated the evaluation and scaling up of the general purpose GPUs from 1970 till the date. The preliminary focus for the development was to make the GPUs ready for general purpose calculations in order to cater the parallel processing benefits to the general purpose applications like scientific application or customer centric applications or the business applications or the financial applications. The recent advancements as demonstrated by D. L. N. Research [3] can delivery 500 Giga Flops, which is nearly four times improved, compared to the CPU cores available in the market.

Significant improvements demonstrated by the GPUs for the application development industry made a substantial impact among the researchers and the demands for programming in parallel languages have increased. Nevertheless, the programming in parallel languages that demands higher efficiencies, which is difficult to obtain due to invisibility of the GPU components, made the task challenging for application developers. In the other hand programming languages, which can take the benefits of parallel cores for any GPU, like CUDA has evolved. Yet, many legacy applications are built using C, a primary serial programming language, also demands to be upgraded to take the advantages of available GPU. Thus the conversion of the code is a primary task for the developers. This includes evaluation of kernels, an independent set of instruction finding, loop controlling and unrolling and finally the parallelization of the code using threads. Consequently, the bottleneck remains the same as demand for parallelization and building an expert development team.
Nonetheless, this leads to a demand for finding the rule sets to convert the source code to CUDA codes automatically and take the recompenses of general purpose GPUs.
This work presents a novel code conversion technique to convert legacy C source codes to the CUDA code. The rest of the work is organized such as in the Section – II literature is reviewed in order to understand the recent advancements of this domain of work, in Section – III CUDA architecture is reviewed to the possible extend in order to establish the framework theory for code conversion, in the Section – IV the algorithm is presented with the light of the mathematical

models, in the Section – V the results are obtained and compared and in the Section – VI this work presents the conclusion.

2. LITERATURE SURVEY

The landmark for the parallel architecture was introduced by NVIDIA in the year of 2006 called Computer Unified Device Architecture or CUDA [3]. This invention was to open the gate for high performance computing on GPU to leverage the execution time for scientific or high graphical applications. The architecture was widely accepted by researchers and developers as the architecture was made available to personal computing and as well as to the servers running low to medium to high computational loads. Another reason for this wide acceptance was the use of multicore processors and shared memory architecture. The notable proof of this concept was presented by Shane Ryoo et. al [4] on performing highly complex scientific applications such as Fast Fourier Transform optimization.
NVIDIA also developed a software development kit or SDK consisting of hardware simulation, drivers, libraries and device drivers for the benefit of the developers. CUDA software stack is composed of several layers: a hardware driver (CUDA Driver), an API and the runtime (CUDA Runtime), two higher-level mathematical libraries (CUDA Libraries) of general purposes [Fig 1].

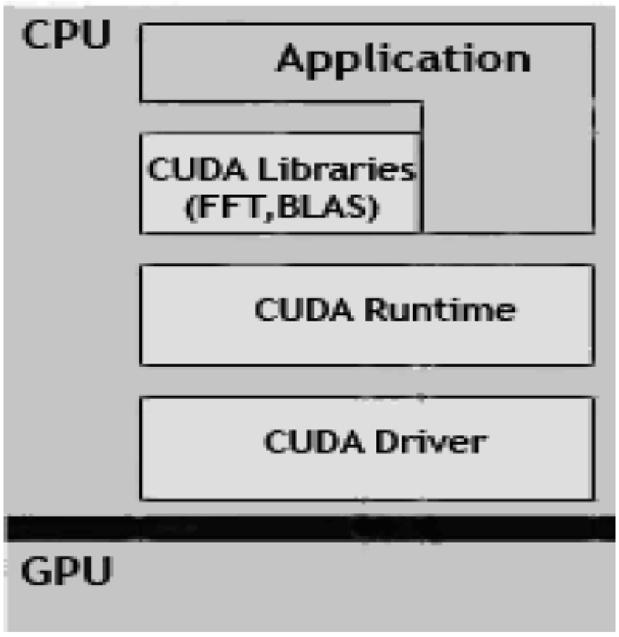


Fig. 1 CUDA Software Stack

The improvement of GPU performance over the traditional CPU architecture was evolved over the hardware organization. NVIDIA strongly recommended that in order to achieve the higher GPU utilization and optimal use of memory hierarchy are two major reasons for performance improvements of GPUs. The notable work by Christian Tenllado et. al [5] have founded the guidelines for a parallel code generation from a serial algorithm majorly focuses on this two principles.

Researchers represent several experiments aimed at analysing their relative importance. Results indicate that code transformations that target efficient memory usage are the major determinant of actual performance. Overall, they ensure the best performance even if some resources remain underutilized. Therefore, maximizing occupancy should be examined at a later stage in the compilation process, once data related issues have been properly addressed.

NVIDIA compiler NVCC can optimize code but the best optimized code is one should write at assembly level. But it looks very difficult in big algorithms and projects. So to find out occupancy is an important issue.

With the availability of the NVIDIA GPU, research focuses on the automatic code conversion techniques for serial codes into parallel. However, the automatic conversion is always debated due to lack of control during the code conversion. Henceforth, in order to overcome this designated problem, this work formulates all necessary guidelines formulated by various researchers by their notable works.

TABLE I: GUIDELINES FOR AUTOMATIC CODE CONVERSION

Researchers / Contributors	Years	Recommendations
B. R. Neha Patil [6]	2007	Focus on the task of parallelization of the algorithms rather than spending time on their implementation.
V. Rajaraman, C. Siva Ram Murthy [7]	2000	Support heterogeneous computation where applications use both the CPU and GPU
V. Rajaraman, C. Siva Ram Murthy [7]	2000	Serial portions of applications are run on the CPU, and parallel portions are run on to the GPU
Setoain, Christian Tenllado, Manuel Arenaz, and Manuel Prieto [1]	2013	Enable heterogeneous systems (CPU + GPU) CPU and GPU are separate devices with separate DRAMs
Setoain, Christian Tenllado, Manuel Arenaz, and Manuel Prieto [1]	2013	Generate a template based on calculated occupancy.
Setoain, Christian Tenllado, Manuel Arenaz, and Manuel Prieto [1]	2013	Conversion of C code in a way it ts in the CUDA C template.
B. R. Neha Patil [6]	2007	Optimize the source code and measure the performance GPU & CPU of the program

Henceforth, considering the notable recommendations from the renounced research outcomes, this work analyses the CUDA architecture and attempt to propose the code conversion algorithm.

4. ARCHITECTURE OF CUDA

The CUDA architecture is designed and developed by the NVIDIA and for the benefit of the application developers and researchers NVIDIA also provides the sufficient understanding of the programming model and the shared memory architecture to utilize the benefits of available GPU. In this section the work furnishes the understanding linking to the serial to parallel code conversion techniques. Understanding Programming Model for CUDA

The GPU is viewed as a compute device, that is a coprocessor to the CPU, has its own device Memory, and runs many threads in parallel [6] Data parallel portion of application are executed on the device as kernels which run in parallel on many threads. Difference between GPU and CPU thread [8] are:

- GPU threads are extremely lightweight and require very little creation overhead.
- GPU needs 1000s of threads for full efficiency whereas multicore CPU needs only a few.

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. Blocks are organized into a one-dimensional or two-dimensional grid of thread blocks as illustrated by Figure 1.

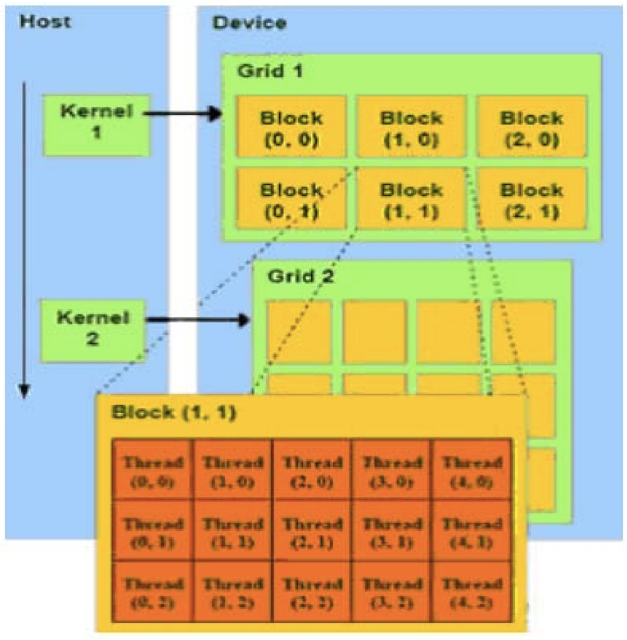


Fig. 2 Programming Model for CUDA

A. Understanding Memory Model for CUDA

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 3. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.

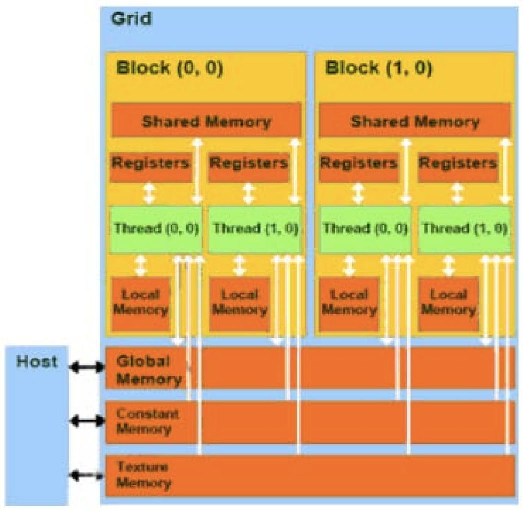


Fig. 3 Memory Model for CUDA

Henceforth with the detailed understanding, this work is now ready to propose the novel algorithm for the code conversion.

5. **NOVEL CODE CONVERSION ALGORITHM**

With the collected recommendations from various research attempts, this work proposes a novel algorithm to convert serial C programs, which is designed to run on the CPU, into a parallel CUDA C program, which can take the complete advantages of the benefits provided by GPUs.
The proposed algorithm is described into 2 individual components as Algorithm 1 and Algorithm 2. Here the Algorithm 1 takes care of the conversion of functions and independence check for the functions, finally converts the functions into CUDA syntax.
In the other hand, the second algorithm converts the basic syntaxes into CUDA C syntaxes and also converts the independent modules into CUDA threads to run on GPUs.
The steps of the algorithm are described here:

Algorithm – 1

```
Step-1. Read the C Source File
Step-2. Find the initial variables and kernel variables
        [Assumption: The variables are expected to be declared
        in the entry section of the source code]
          a. Find the global variable instances and library
             files
          b. Build the symbol table for all the notation
Step-3. Find the declared functions
          a. If the function is main method
               i. Maintain the syntax
          b. Else, In case of non-main method
               i. Convert the functions as global
                  function
              ii. Include __global__ clause for each
Step-4. Find the functional dependencies
          a. If the function has a dependency
               i. Write the function as a device
                  definition function
              ii. Repeat Step 3.b
          b. Else, In case of independent functions
               i. Continue
Step-5. Finalize the conversion as main.cu file
```

Algorithm – 2

```
Step-1. Read the main.cu file
Step-2. Analyse the code and evaluate the BlockGrid with
        BlockDim values
            a. Replace the variable declarations using
               CUDAMalloc and CUDAMemcpy
Step-3. Analyse the code for pragma kernel regons
            a. Convert each region into kernel calls
Step-4. Analyse the code using dependency checker
            a. If independent routines
                    i. Convert each independent routines
                       into threads
            b. Else, in case of dependent routine
                    i. Continue
Step-5. Compile the main.cu
Step-6. Execute main.cu
```

The results of this algorithm is also been discussed in this work in the next section.

6. RESULTS AND DISCUSSION

The intension of this work is to demonstrate the improvement of the performance for parallel application over serial application. The automatic conversion of the source code is always debated and hence this work provides much larger and concrete proofs for the demonstration of the improvements.
In this section, the results demonstrate the converted source code and analyses the serial and parallel execution time.

A. *Analysis of the Performance on Binary Search*

The first analysis is demonstrated on the popular binary search code. Firstly the source C program is converted into CUDA C automatically using the Novel Code Converter proposed in this work [Table – 2].

TABLE II: CUDA – C EQUIVALENT & CONVERTED CODE FOR BINARY SEARCH

Recommendations

```
#include<stdio.h>
__global__ void kernel(int *gpu_bb,int *gpu_nn,int *gpu_aa,int
*gpu_cc)
{
     int gpu_i=threadIdx.x+blockIdx.x*blockDim.x;
   if(*gpu_bb==gpu_aa[gpu_i])
       {
          *gpu_cc=1;
       }
}
int main()
{
   int a[90000],b,mid,n,i;
    int c=0;
  //Kernel Variables
    int *g_p,*g_n,*g_b,*g_a,*g_c;
   //CUDA GRID BLOCK SIZE AND NUMBER OF BLOCKS
    int block_size = 32;
    const int N = 90000;  // Number of elements in arrays
    int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
    size_t size = 90000 * sizeof(int);
// Memory Allocation
    cudaMalloc((void**)&g_p,sizeof(int)); // Allocate array on
devic
    cudaMalloc((void**)&g_b,sizeof(int)); // Allocate array on
devic
    cudaMalloc((void**)&g_c,sizeof(int)); // Allocate array on
devic
    cudaMalloc((void**)&g_a,size); // Allocate array on devic
    n=90000;
    for(i=0;i<n;i++)
    {
    a[i]=i;
    }
    b=100000;
// Copy Data to device from host
    cudaMemcpy(g_b,&b,sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(g_n,&n,sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(g_c,&c,sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(g_a,&a,size,cudaMemcpyHostToDevice);
// call kernel
    kernel<<<n_blocks, block_size>>>(g_b,g_n,g_a,g_c);
// Retrieve result from device and store it in host array
    cudaMemcpy(&bg_b,,sizeof(int),cudaMemcpyDeviceToHost);
    cudaMemcpy(&n,g_n,sizeof(int),cudaMemcpyDeviceToHost);
    cudaMemcpy(&c,g_c,sizeof(int),cudaMemcpyDeviceToHost);
    cudaMemcpy(&a,g_a,size,cudaMemcpyDeviceToHost);
 // Free GPU Variables
    cudaFree(g_b);
    cudaFree(g_n);
    cudaFree(g_c);
    cudaFree(g_a);
    if(c==0)
    {
       printf("The number is not found\n\n");
    }
    else
    {
       printf("The number is found\n\n");
    }
    system("pause");
   return 0;
}
```

Furthermore, the comparison for CPU time is also been analysed [Table – 3].

TABLE III: CPU VS GPU EXECUTION TIME FOR BINARY SEARCH

Diagnosis Session Duration	10 Seconds		20 Seconds		30 Seconds		40 Seconds		50 Seconds	
C on CPU	39	39	78	78	117	117	156	156	195	195
CUDA-C on GPU	29	29	58	58	87	87	116	116	145	145
Improvement (%)	74.36	74.36	74.36	74.36	74.36	74.36	74.36	74.36	74.36	74.36

The result is also been analysed visually for CPU [Fig – 4] & for GPU [Fig – 5].

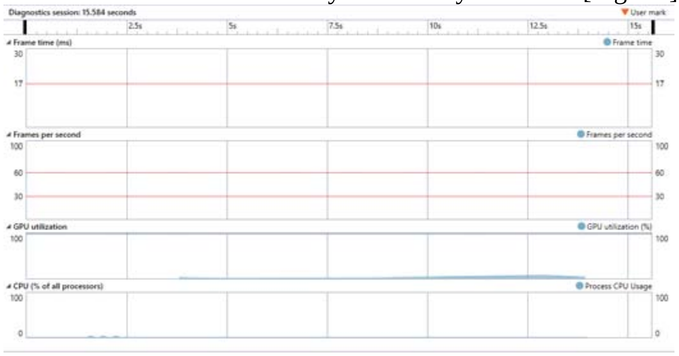


Fig. 4 CPU Analysis for Binary Search

Also this work analyses the Hot Path for Code Execution analysis [Table – 4].

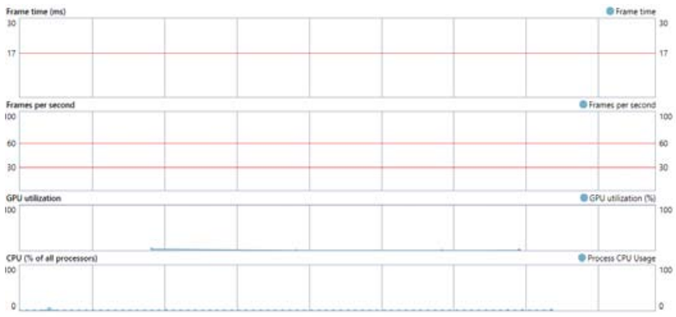
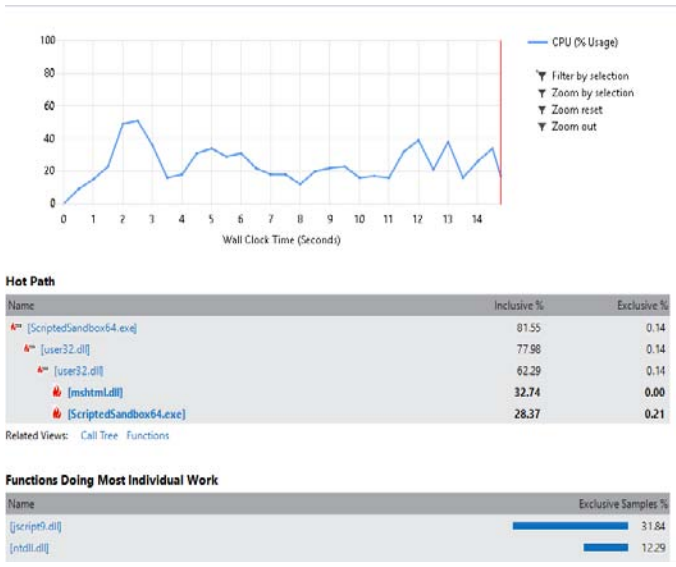


Fig. 5 GPU Analysis for Binary Search

TABLE IV: HOT CODE EXECUTION PATH UTILIZATION – BINARY SEARCH

Code Name	Serial Execution – Hot Path Utilization (%)	Parallel Execution – Hot Path Utilization (%)
Binary Search	81.55	79.46

The result is also been visually analysed for CPU [Fig – 6] and GPU [Fig – 7]



Hot Path

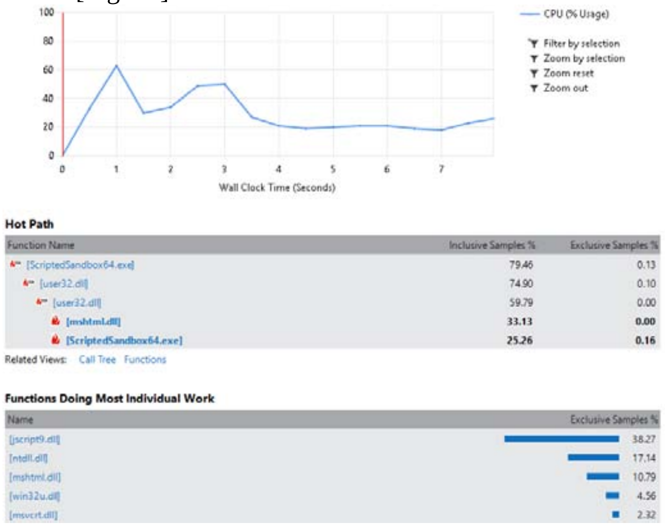
Name	Inclusive %	Exclusive %
[ScriptedSandbox64.exe]	81.55	0.14
[user32.dll]	77.98	0.14
[user32.dll]	62.29	0.14
[mshtml.dll]	**32.74**	**0.00**
[ScriptedSandbox64.exe]	**28.37**	**0.21**

Related Views: Call Tree Functions

Functions Doing Most Individual Work

Name	Exclusive Samples %
[jscript9.dll]	31.84
[ntdll.dll]	12.29

Fig. 6 Serial Hot Code Path for Binary Search



Hot Path

Function Name	Inclusive Samples %	Exclusive Samples %
[ScriptedSandbox64.exe]	79.46	0.13
[user32.dll]	74.90	0.10
[user32.dll]	59.79	0.00
[mshtml.dll]	**33.13**	**0.00**
[ScriptedSandbox64.exe]	**25.26**	**0.16**

Related Views: Call Tree Functions

Functions Doing Most Individual Work

Name	Exclusive Samples %
[jscript9.dll]	38.27
[ntdll.dll]	17.14
[mshtml.dll]	10.79
[win32u.dll]	4.56
[msvcrt.dll]	2.32

Fig. 7 Parallel Hot Code Path for Binary Search

B. Analysis of the Performance on KnapSack

Next, the analysis is demonstrated on the popular KnapSack code. Firstly the source C program is converted into CUDA C automatically using the Novel Code Converter proposed in this work [Table –5].

TABLE V: CUDA – C EQUIVALENT & CONVERTED CODE FOR KNAPSACK

Recommendations

```
__global__ void kernel(int *gpu_c,int *gpu_v,int *gpu_a,int
*gpu_last_added) {
        int gpu_i = threadIdx.x+1;
        int gpu_j;

            gpu_j = 0;

              if ((gpu_c[gpu_j] <= gpu_i) && (gpu_a[gpu_i] <
gpu_a[gpu_i-gpu_c[gpu_j]] + gpu_v[gpu_j])){

                  int s=gpu_i;
                  gpu_a[gpu_i] = gpu_a[gpu_i-gpu_c[gpu_j]] +
gpu_v[gpu_j];
                  gpu_last_added[s] = gpu_j;

              }
               gpu_j = 1;

              if ((gpu_c[gpu_j] <= gpu_i) && (gpu_a[gpu_i] <
gpu_a[gpu_i-gpu_c[gpu_j]] + gpu_v[gpu_j])){

                  int s=gpu_i;
                  gpu_a[gpu_i] = gpu_a[gpu_i-gpu_c[gpu_j]] +
gpu_v[gpu_j];
                  gpu_last_added[s] = gpu_j;

              }
               gpu_j = 2;

              if ((gpu_c[gpu_j] <= gpu_i) && (gpu_a[gpu_i] <
gpu_a[gpu_i-gpu_c[gpu_j]] + gpu_v[gpu_j])){

                  int s=gpu_i;
                  gpu_a[gpu_i] = gpu_a[gpu_i-gpu_c[gpu_j]] +
gpu_v[gpu_j];
```

```
            gpu_last_added[s] = gpu_j;

        }

}
int main(int argc, char *argv[]) {
    int n=3;
    int W=10;
    int i,j;
    int aux;

    //Kernel Variables
    int c[10];int *g_c;
    int v[10];int *g_v;
    int a[MAXWEIGHT];int *g_a;
    int last_added[MAXWEIGHT];int *g_last_added;

    //CUDA GRID BLOCK SIZE AND NUMBER OF BLOCKS
    int block_size = 10;
    const int N = 10;  // Number of elements in arrays
    int n_blocks = N/block_size + (N%block_size == 0 ? 0:1)+1;

    c[0] = 8;
    c[1] = 6;
    c[2] = 4;
    v[0] = 16;
    v[1] = 10;
    v[2] = 7;
    for (i = 0; i <= W; ++i) {
        a[i] = 0;
        last_added[i] = -1;
    }
    a[0] = 0;

    // Memory Allocation
    g_c= (int *)malloc(10 *sizeof(int)); // Allocate array on host
     cudaMalloc((void **) &g_c,10 * sizeof(int)); // Allocate
array on devic

    g_v= (int *)malloc(10 *sizeof(int)); // Allocate array on host
     cudaMalloc((void **) &g_v,10 * sizeof(int)); // Allocate
array on devic

    g_a= (int *)malloc(MAXWEIGHT *sizeof(int)); // Allocate
array on host
     cudaMalloc((void **) &g_a,MAXWEIGHT * sizeof(int)); //
Allocate array o

    g_last_added= (int *)malloc(MAXWEIGHT *sizeof(int)); //
Allocate array o

    cudaMalloc((void **) &g_last_added,MAXWEIGHT *
sizeof(int)); // Allocat


    // Copy Data to device from host
    cudaMemcpy(g_c, c,10 * sizeof(int),
cudaMemcpyHostToDevice);
    cudaMemcpy(g_v, v,10 * sizeof(int),
cudaMemcpyHostToDevice);
    cudaMemcpy(g_a, a,MAXWEIGHT * sizeof(int),
cudaMemcpyHostToDevice);
    cudaMemcpy(g_last_added, last_added,MAXWEIGHT *
sizeof(int), cudaMemcpyHostToDevice);

    // call kernel
    kernel <<< 1 ,block_size >>>( g_c, g_v, g_a, g_last_added);

    // Retrieve result from device and store it in host array
    cudaMemcpy(c, g_c,10 * sizeof(int),
cudaMemcpyDeviceToHost);
    cudaMemcpy(v, g_v,10 * sizeof(int),
cudaMemcpyDeviceToHost);
    cudaMemcpy(a, g_a,MAXWEIGHT * sizeof(int),
cudaMemcpyDeviceToHost);
    cudaMemcpy(last_added, g_last_added,MAXWEIGHT *
sizeof(int), cudaMemcpyDeviceToHost);

    // Free GPU Variables
    cudaFree(g_c);
    cudaFree(g_v);
    cudaFree(g_a);
    cudaFree(g_last_added);


    for (i = 0; i <= W; ++i) {
        if (last_added[i] != -1){
            printf("Weight %d; Benefit: %d; To reach this weight I
added object %d (%d$ %dKg) to weight %d.\n",i,a[i],last_added[i]
+ 1,v[last_added[i]],c[last_
added[i]],i - c[last_added[i]]);
        }

        else {
            printf("Weight %d; Benefit: 0; Can't reach this exact
weight.\n",i);
        }

    }
    printf("---\n");
    aux = W;
    while ((aux > 0) && (last_added[aux] != -1)) {
        printf("Added object %d (%d$ %dKg). Space
left: %d\n",last_added[aux] +
1,v[last_added[aux]],c[last_added[aux]],aux - c[last_added[aux]]);
        aux -= c[last_added[aux]];
    }
    printf("Total value added: %d$\n",a[W]);
    system("pause");
    return 0;

}
```

Furthermore, the comparison for CPU time is also been analysed [Table – 6].

TABLE VI: CPU VS GPU EXECUTION TIME FOR KNAPSACK

Diagnosis Session Duration	**10 Seconds**		**20 Seconds**		**30 Seconds**		**40 Seconds**		**50 Seconds**	
C on CPU	45	45	90	90	135	135	180	180	225	225
CUDA-C on GPU	31	31	62	62	93	93	124	124	155	155
Improvement (%)	68.89	68.89	68.89	68.89	68.89	68.89	68.89	68.89	68.89	68.89

The result is also been analysed visually for CPU [Fig – 8] & for GPU [Fig – 9].

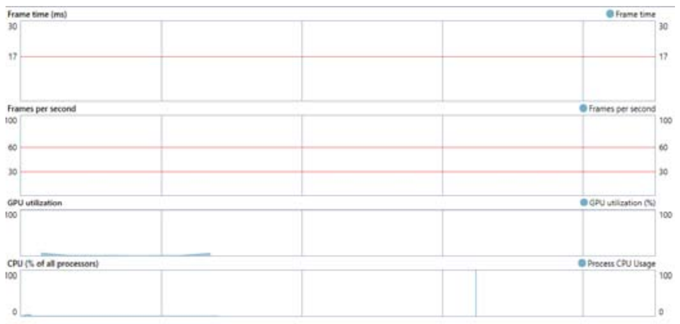


Fig. 8 CPU Analysis for KNAPSACK

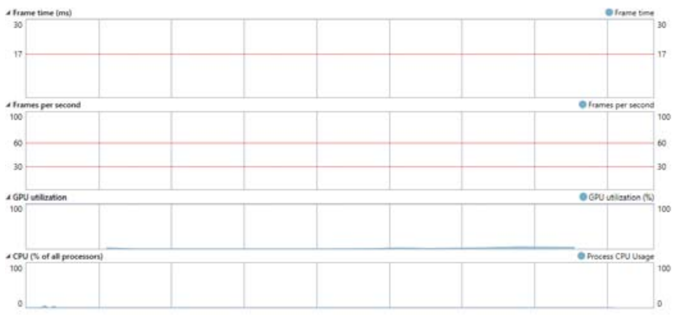


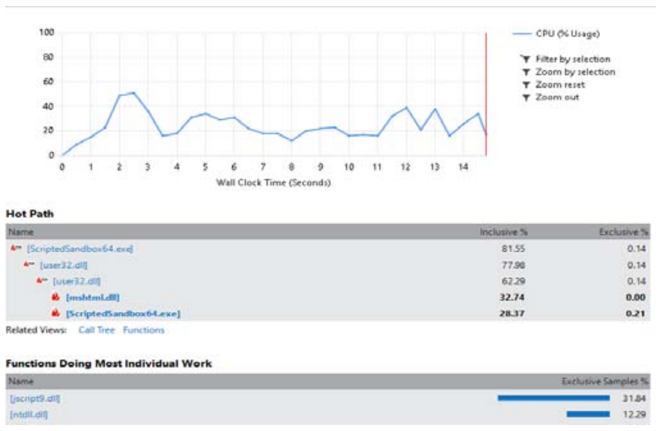
Fig. 9 GPU Analysis for KNAPSACK

Also this work analyses the Hot Path for Code Execution analysis [Table – 7].

TABLE VII: HOT CODE EXECUTION PATH UTILIZATION – KNAPSACK

Code Name	**Serial Execution – Hot Path Utilization (%)**	**Parallel Execution – Hot Path Utilization (%)**
KnapSack	87.25	87.25

The result is also been visually analysed for CPU [Fig – 10] and GPU [Fig – 11]



Hot Path

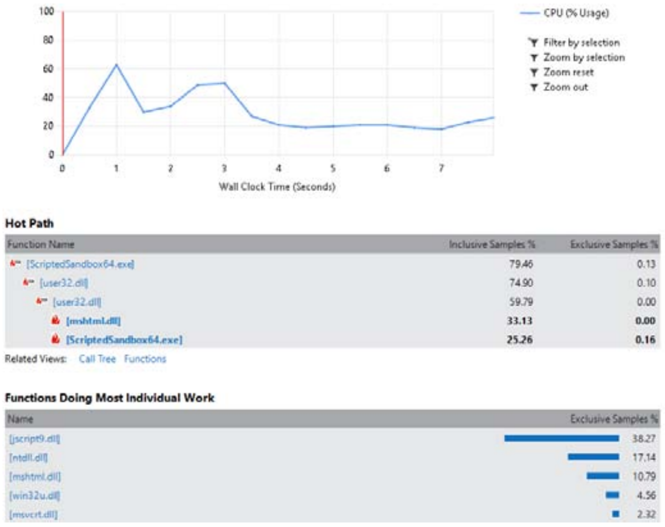
Name	Inclusive %	Exclusive %
[ScriptedSandbox64.exe]	81.55	0.14
[user32.dll]	77.98	0.14
[user32.dll]	62.29	0.14
[mshtml.dll]	32.74	0.00
[ScriptedSandbox64.exe]	28.37	0.21

Related Views: Call Tree Functions

Functions Doing Most Individual Work

Name	Exclusive Samples %
[jscript9.dll]	31.84
[ntdll.dll]	12.29

Fig. 10 Serial Hot Code Path for KNAPSACK



Hot Path

Function Name	Inclusive Samples %	Exclusive Samples %
[ScriptedSandbox64.exe]	79.46	0.13
[user32.dll]	74.90	0.10
[user32.dll]	59.79	0.00
[mshtml.dll]	33.13	0.00
[ScriptedSandbox64.exe]	25.26	0.16

Related Views: Call Tree Functions

Functions Doing Most Individual Work

Name	Exclusive Samples %
[jscript9.dll]	38.27
[ntdll.dll]	17.14
[mshtml.dll]	10.79
[win32u.dll]	4.56
[msvcrt.dll]	2.32

Fig. 11 Parallel Hot Code Path for KNAPSACK

C. *Analysis of the Performance on Vector Sum*

Next, the analysis is demonstrated on the popular Vector Sum code. Firstly the source C program is converted into CUDA C automatically using the Novel Code Converter proposed in this work [Table –8].

TABLE VIII: CUDA – C EQUIVALENT & CONVERTED CODE FOR VECTOR SUM

Recommendations

`__global__ void kernel(float *gpu_a,float *gpu_b,float *gpu_c) {`
`  int idx=threadIdx.x + blockDim.x*blockDim.x;`
 	
`int gpu_i=idx;`
`      gpu_c[gpu_i] = gpu_a[gpu_i] + gpu_b[gpu_i];`
 	
`}`
`int main(int argc, char *argv[]) {`
`        int n=3;`
`        int i;`
`        int j;`
 	
`        //Kernel Variables`
`        float a[1024];float *g_a;`
`        float b[1024];float *g_b;`
`        float c[1024];float *g_c;`
 	
`        //CUDA GRID BLOCK SIZE AND NUMBER OF BLOCKS`
`        int block_size = 1;`
`//        const int N = 10;  // Number of elements in arrays`
`        int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);`

```
        for (i = 0; i <= N; ++i) {
            a[i] = i;
            b[i] = i;
        }

        // Memory Allocation
        g_a= (float *)malloc(1024 *sizeof(float)); // Allocate array on
host
         cudaMalloc((void **) &g_a,1024 * sizeof(float)); // Allocate
array on device
        g_b= (float *)malloc(1024 *sizeof(float)); // Allocate array on
host
         cudaMalloc((void **) &g_b,1024 * sizeof(float)); // Allocate
array on device
        g_c= (float *)malloc(1024 *sizeof(float)); // Allocate array on
host
         cudaMalloc((void **) &g_c,1024 * sizeof(float)); // Allocate
array on device

        // Copy Data to device from host
        cudaMemcpy(g_a, a,1024 * sizeof(float),
cudaMemcpyHostToDevice);
        cudaMemcpy(g_b, b,1024 * sizeof(float),
cudaMemcpyHostToDevice);
        cudaMemcpy(g_c, c,1024 * sizeof(float),
cudaMemcpyHostToDevice);

        // call kernel
         kernel <<< n_blocks, block_size >>>( g_a, g_b, g_c);

        // Retrieve result from device and store it in host array
        cudaMemcpy(a, g_a,1024 * sizeof(float),
cudaMemcpyDeviceToHost);
        cudaMemcpy(b, g_b,1024 * sizeof(float),
cudaMemcpyDeviceToHost);
        cudaMemcpy(c, g_c,1024 * sizeof(float),
cudaMemcpyDeviceToHost);

        // Free GPU Variables
        cudaFree(g_a);
        cudaFree(g_b);
        cudaFree(g_c);
```

```
        for (i = 0; i < N; i++) {
            printf("A=%6.2f B=%6.2f A+B = %6.2f\n",a[i],b[i],c[i]);
        }
        system("pause");
        return 0;

}
```

Furthermore, the comparison for CPU time is also been analysed [Table – 9].

TABLE IX: CPU VS GPU EXECUTION TIME FOR VECTOR SUM

Diagnosis Session Duration	10 Seconds		20 Seconds		30 Seconds		40 Seconds		50 Seconds	
C on CPU	281	281	562	562	843	843	1124	1124	1405	1405
CUDA-C on GPU	274	274	548	548	822	822	1096	1096	1370	1370
Improvement (%)	97.51	97.51	97.51	97.51	97.51	97.51	97.51	97.51	97.51	97.51

The result is also been analysed visually for CPU [Fig – 12] & for GPU [Fig – 13].

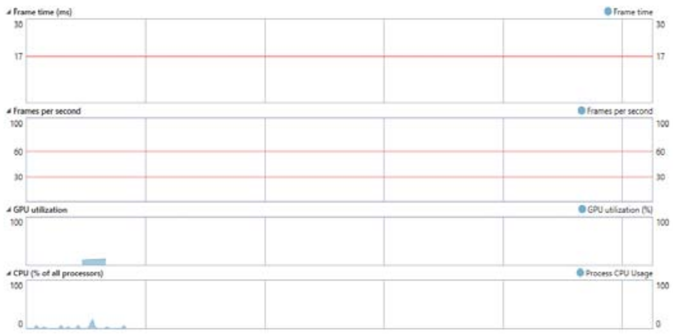


Fig. 12 CPU Analysis for VECTOR SUM

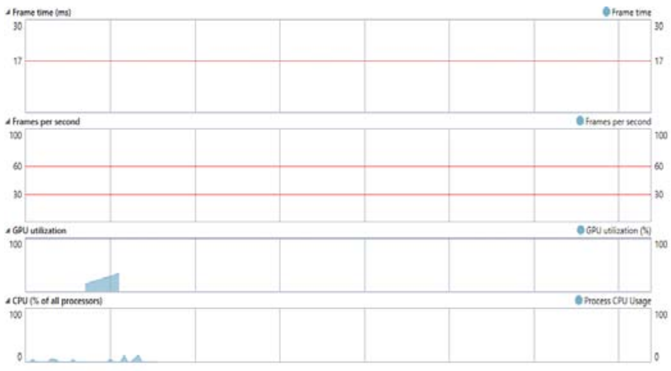


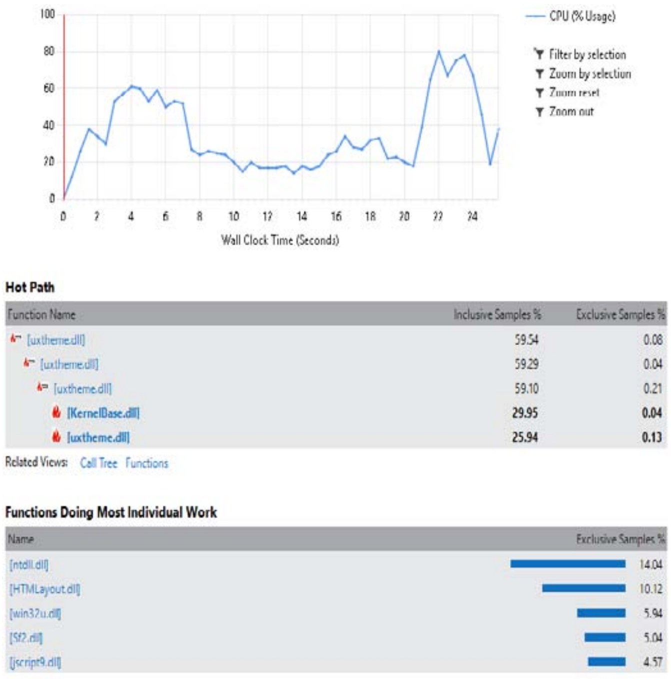
Fig. 13 GPU Analysis for VECTOR SUM

Also this work analyses the Hot Path for Code Execution analysis [Table – 10].

TABLE X: HOT CODE EXECUTION PATH UTILIZATION – VECTOR SUM

Code Name	Serial Execution – Hot Path Utilization (%)	Parallel Execution – Hot Path Utilization (%)
Vector Sum	59.54	60.59

The result is also been visually analysed for CPU [Fig – 14] and GPU [Fig – 15]



Hot Path

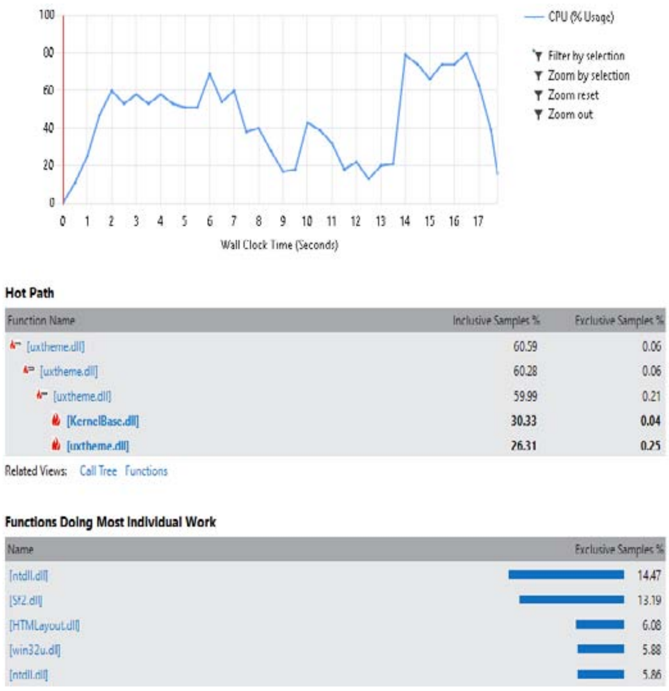
Function Name	Inclusive Samples %	Exclusive Samples %
[uxtheme.dll]	59.54	0.08
[uxtheme.dll]	59.29	0.04
[uxtheme.dll]	59.10	0.21
[KernelBase.dll]	29.95	0.04
[uxtheme.dll]	25.94	0.13

Related Views: Call Tree Functions

Functions Doing Most Individual Work

Name	Exclusive Samples %
[ntdll.dll]	14.04
[HTMLayout.dll]	10.12
[win32u.dll]	5.94
[Sf2.dll]	5.04
[jscript9.dll]	4.57

Fig. 14 Serial Hot Code Path for VECTOR SUM



Hot Path

Function Name	Inclusive Samples %	Exclusive Samples %
[uxtheme.dll]	60.59	0.06
[uxtheme.dll]	60.28	0.06
[uxtheme.dll]	59.99	0.21
[KernelBase.dll]	30.33	0.04
[uxtheme.dll]	26.31	0.25

Related Views: Call Tree Functions

Functions Doing Most Individual Work

Name	Exclusive Samples %
[ntdll.dll]	14.47
[Sf2.dll]	13.19
[HTMLayout.dll]	6.08
[win32u.dll]	5.88
[ntdll.dll]	5.86

Fig. 15 Parallel Hot Code Path for VECTOR SUM

Also, this work presents the average execution time improvement for parallel code over serial code [Table – 11].

TABLE XI: AVERAGE EXECUTION TIME IMPROVEMENT

Code Name	Improvement (%)
Binary Search Problem	74.36
KnapSack Problem	68.89
VectorSum Problem	97.51
Average Improvement	80.25

Thus with the light of the obtained results, this work presents the conclusions in the next section.

7. CONCLUSIONS

The demand for automatic conversion of the serial code to parallel codes in order to reduce the execution time and defeat the fact of higher efficiencies in the workforce is always under a focus for the research. This work deploys a novel algorithm to convert serial C codes into parallel NVIDIA CUDA codes to take the maximum benefits from the GPUs available. The automatic conversion framework, proposed and demonstrated in this work, not only reduces the time for the conversion, also reduces 80% of the execution time for legacy serial programs from various algorithmic approaches. The conversion framework demonstrated a 100% similar result upon execution and works for all programs written following the fundamental guidelines of the code developments. This work is to be seen as one of the baseline for further research and a contribution towards automatic code translation for legacy systems in order to make the computational support for modern developments.

REFERENCES

[1] Shane Ryoo, Sam S. Stone, "Optimization principles and application performance evaluation of multithreaded GPU using CUDA", Center for Reliable and high-performance Computing University of Illinois at Urbana-Champaign NVIDIA Corporation, 2009.

[2] R. Kresch and N. Merhav, "Fast DCT domain altering using the DCT and the DST," HPL Technical Report HPL-95-140, December 1995.

[3] D. L. N. Research, "NVIDIA gpu architecture & implications,", NVIDIA Corporation 2007.

[4] Shane Ryoo, Christopher I. Rodrigue, Sara S. Baghsorkhi, "Optimizing the Fast Fourier Transform on a Multi-core Architecture," 2006-2008.

[5] Setoain, Christian Tenllado, Manuel Arenaz, and Manuel Prieto, "Towards Automatic Code Generation for GPU architectures", Computer Architecture Group, Department of Electronics and Systems, University of A Coruna,Spain.

[6] B. R. Neha Patil, "SFast and parallel implementation of image processing algorithm using cuda technology on gpu hardware", ", tech. rep., Department of Electrical & Computer and Systems Engineering, Rensselaer Polytechnic Institute,Troy, NY 12180-3590.

[7] V. Rajaraman, C. Siva Ram Murthy, "Parallel Computers Architecture and Programming", Prentice Hall,2000,ISBN-81-203-1621-5.