IDENTIFICATION OF HOT PATH FOR CODE EXECUTION OPTIMIZATION

GRANDHI PRASUNA
Reasearch Scholar, Dept. Of CSE,
Acharya Nagarjuna University, AP
grandhiprasuna@gmail.com

Dr. **O. NAGA RAJU**
Asst. Professor & Head, Dept. Of
Computer Science, Govt. Degree College,
Macherla, AP

Abstract— Optimization of the source code is a widely accepted process for managing the performance of any software applications or components. The highest level of software optimization can be identified with the help of highly used control paths of the applications. These highly used control paths are considered as hot path for the application control flow. Number of parallel research outcomes have demonstrated significant outcomes for designing the hot path flow of the algorithm. Nevertheless, majority of the parallel research works have only identified the paths and not deployed the same concept for code optimization on the run time. Henceforth, this work demonstrates a novel algorithm for code optimization using hot path.

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

I. 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 timesimproved, 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, anindependent 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.

II. 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].

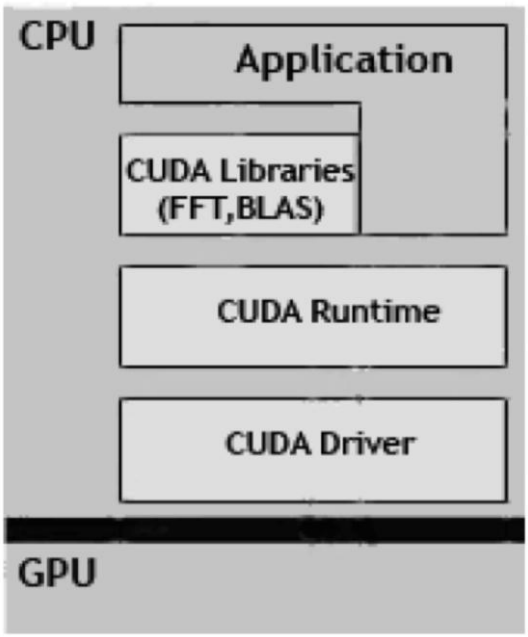

Fig. 1 CUDA Software Stack

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 these 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 is 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.

III. 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: Automatic Source Code Conversion for Optimization

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

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

IV. 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 - AS EXAMPLE

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. 2CPU Analysis for Binary Search

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

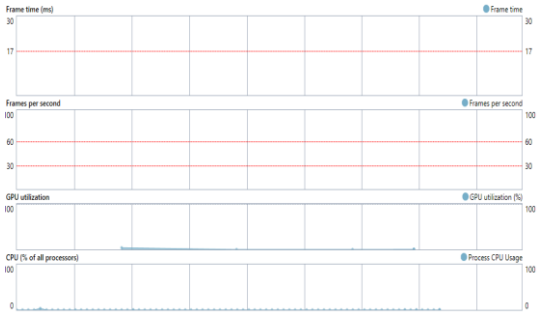

Fig. 3GPU Analysis for Binary Search

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]

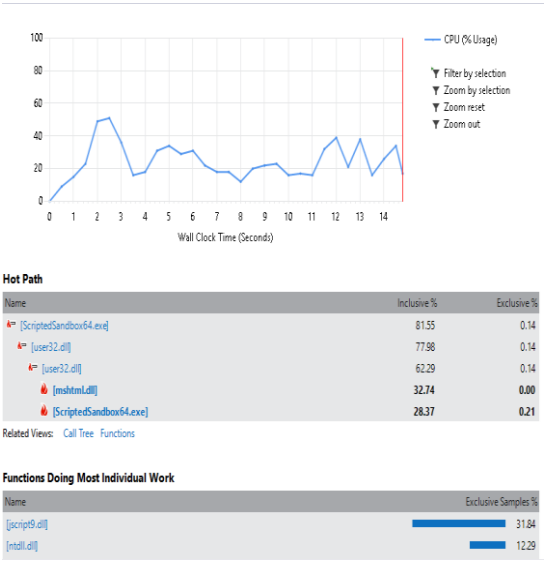

Fig. 4 Serial Hot Code Path for Binary Search

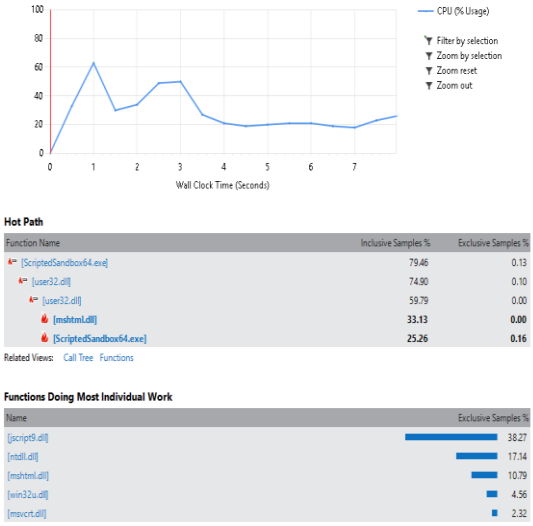

Fig. 5 Parallel Hot Code Path for Binary Search

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

V. CONCLUSIONS

The interest for programmed change of the sequential code to resemble codes so as to decrease the execution time and destruction the reality of higher efficiencies in the workforce is constantly under a concentration for the examination. This work sends a novel calculation to change over sequential C codes into equal NVIDIA CUDA codes to take the most extreme advantages from the GPUs accessible. The programmed change system,

proposed and exhibited right now, just lessens the ideal opportunity for the transformation, additionally decreases 80% of the execution time for heritage sequential projects from different algorithmic methodologies. The transformation system exhibited a 100% comparable outcome upon execution and works for all projects composed after the central rules of the code advancements. This work is to be viewed as one of the gauges for additional examination and a commitment towards programmed code interpretation for heritage frameworks so as to make the computational help for current improvements.

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.*