Automated Tool to Generate Parallel CUDA code from a Serial C Code

Akhil Jindal
Department of Computer Engineering, Delhi Technological University, Shahbad Daulatpur, Main Bawana Road, Delhi - 110042, India.

Nikhil Jindal
Department of Computer Engineering, Delhi Technological University, Shahbad Daulatpur, Main Bawana Road, Delhi - 110042, India.

Divyashikha Sethia
Department of Software Engineering, Delhi Technological University, Shahbad Daulatpur, Main Bawana Road, Delhi - 110042, India.

ABSTRACT

With the introduction of GPGPUs, parallel programming has become simple and affordable. APIs such as NVIDIA's CUDA have attracted many programmers to port their applications to GPGPUs. But writing CUDA codes still remains a challenging task. Moreover, the vast repositories of legacy serial C codes, which are still in wide use in the industry, are unable to take any advantage of this extra computing power available. Lot of attempts have thus been made at developing auto-parallelization techniques to convert a serial C code to a corresponding parallel CUDA code. Some parallelizes, allow programmers to add "hints" to their serial programs, while another approach has been to build an interactive system between programmers and parallelizing tools/compilers. But none of these are really automatic techniques, since the programmer is fully involved in the process. In this paper, we present an automatic parallelization tool that completely relieves the programmer of any involvement in the parallelization process. Preliminary results with a basic set of usual C codes show that the tool is able to provide a significant speedup of ~10 times.

General Terms

Auto parallelization, parallelization, C, CUDA, hiCUDA, GPU.

Keywords

Auto parallelization, parallelization, C, CUDA, hiCUDA, GPU.

1. INTRODUCTION

In the last decades, there have been great advancements in the field of Parallel Computing. With the introduction of General Purpose Graphical Processing Units (GPGPUs), attaining parallel processing capability has become simple and affordable. A typical GPU is a multi-core architecture with each core capable of running thousands of threads simultaneously. Hence, an application with a large amount of parallelism can use GPUs to realize significant performance benefits. SDKs and APIs such as Nvidia's CUDA [1], AMD's FireStream and Khronos Group's Open CL [2] have simplified the task of programming GPUs. Some of the areas where GPUs have been used extensively for General Purpose computing are: scientific computing [3][4], image processing [5][6][7], animation and simulation [8][9] and cryptography [10].

But the vast repositories of legacy serial C codes, which are still in use, are unable to exploit this extra computing power available to them. Manually updating all such codes is tedious and error-prone. Parallelizing even a single C code is not a trivial task. The programmer needs to have a complete knowledge of the code being parallelized and should be comfortable with the target parallel architecture. Also, even though APIs, such as those of CUDA, have attracted many non-graphics programmers to port their applications to GPGPUs, the process still remains very challenging. In particular, CUDA places on the programmer the burden of packaging GPU code in separate functions, of explicitly managing data transfer between the host memory and various GPU memories, and of manually optimizing the utilization of the GPU memory [11].

Due to the reasons mentioned above, we have undertaken the task to develop "Automated Tool to Generate Parallel CUDA code from a Serial C Code". The tool is aimed at enabling easy portability of existing serial softwares to parallel architectures. This should be possible without the user having any knowledge whatsoever of the algorithm and the architecture.

Though the quality of automatic parallelization has improved in the past several decades, fully automatic parallelization of sequential programs by compilers remains a grand challenge due to its need for complex program analysis and the unknown factors (such as input data range) during compilation. Attempts have been made at simplifying the process of manual parallelization by allowing programmers to add "hints" to their programs to guide compiler parallelization, such as High Performance Fortran (HPF) [12] for distributed memory systems and OpenMP [13] for shared memory systems. Another approach has been to build an interactive system between programmers and parallelizing tools/compilers. Notable examples are Vector Fabrics' vfAnalyst [14], SUIF Explorer [15] (The Stanford University Intermediate Format compiler), the Polaris compiler [16], and ParaWise (formally CAPTools) [17].

There exist some directive based auto-parallelization tools for CUDA such as PGI Accelerator, CAPS HMPP, Goose, NOAA F2C Fortran/C to CUDA C auto-parallelization compiler and hiCUDA [18]. But the drawback in all these tools is that the programmer has to understand and learn the specific compiler directive syntax. Our proposed tool goes a step further than these tools in simplifying the process for the user by automatically generating the parallel code from input serial code without any additional input from the user.

The tool works in two phases. In the first phase, the input serial code is parsed to identify independent portions of code which can be executed in parallel. Identifiers are then automatically inserted at appropriate positions to mark these parallelizable portions. In the second phase, an equivalent CUDA code is generated which parallelizes the portions identified in Phase 1.

The parallel code obtained might not be as efficient as hand-tuned programs but can still lead to tremendous speedups with a quick production phase.

The paper is organized as follows. Section 2 presents the system architecture while the major issues in identifying and then parallelizing portions of serial code are discussed in section 3. Section 4 analyses the results and Section 5 ponders over some future work that can lead to better results.

2. SYSTEM ARCHITECTURE

The proposed tool takes a serial C code as input and generates an equivalent parallel code as output, as illustrated by the flowchart in Figure 1. The generated output code can be compiled and executed on any machine with a CUDA enabled graphics card.

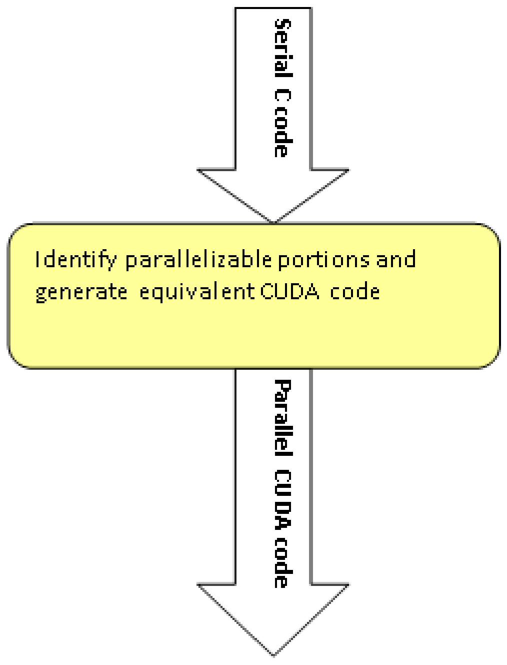
Serial C code
Identify parallelizable portions and generate equivalent CUDA code
Parallel CUDA code

Fig 1: Flowchart showing the input and output with the tool as a black-box

Internally, the tool works in two phases (Figure 2). hiCUDA is used as the intermediate language between them:

• In the first phase, the input serial code is parsed using a Perl script to identify independent portions of code which can be executed in parallel. Identifiers (hiCUDA pragmas) are then automatically inserted at appropriate positions to highlight these parallelizable portions.

• In the second phase, the hiCUDA compiler is used to generate an equivalent CUDA code using the hiCUDA pragmas inserted in Phase 1.

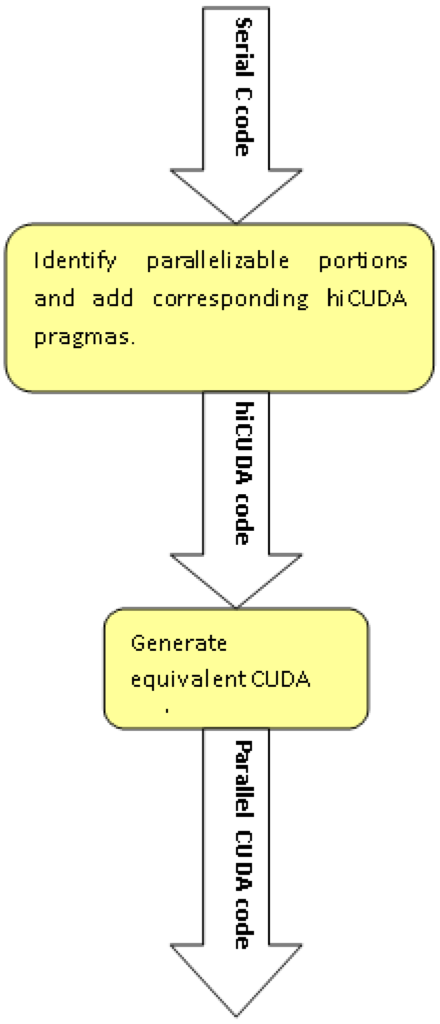
Serial C code
Identify parallelizable portions and add corresponding hiCUDA pragmas.
hiCUDA code
Generate equivalent CUDA
Parallel CUDA code

Fig 2: Representation of the internal phases

3. IDENTIFY AND PARALLELIZE PARALLELIZABLE PORTIONS

As most of the execution time of a program takes place inside some form of loop, we intend to focus most on them and will try to split each loop so that each of its iteration can be executed on separate processors concurrently.

Also, since GPUs are optimized for executing SIMD type instructions, they are guaranteed to give maximum gain.

For example, consider the following code snippet:

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

Code 1. Single for loop

Since instructions in each iteration are independent of each other, this code can be easily parallelized by using n threads running in parallel, each operating on one element of vector a.

3.1 When to parallelize

Not all for loops can be parallelized. Hence, we need to have a set of rules defining when a for loop is parallelized and when not. For our purpose, we do not parallelize in the following cases:

- Presence of an I/O instruction in a loop.
- Presence of a break/return/goto statement in a loop.
- A scalar is being Written after Read (WAR).
- Same element of an array being written in each iteration of the loop.
- 2 different elements of same array being accessed in each iteration (at least one being written).

3.2 Handling Nested Loops

In case of nested loops, a separate read and write list is created for each loop independently and the analysis as explained above is done to determine whether the loop is parallelizable.

To handle nested loops, we create a GPU kernel for each bunch of nested loops. Each loop is analyzed independently and a kernel is created if at least one of the nested loops can be parallelized.

For example, consider the following snippet:

```
for( i = 0; i < n; i++) {
        for( j  = 0; j < n; j++) {
                sum[i] = sum[i] + a[i][j];
        }
}
```

Code 2. Nested for loop

In this case, the outer loop will be parallelized while the inner loop will not be, which is indicated by the hiCUDA pragma just before the outer loop only:

```
#pragma hicuda loop_partition over_tblock over_thread
for( i = 0; i < n; i++) {
        for( j  = 0; j < n; j++) {
                sum[i] = sum[i] + a[i][j];
        }
}
```

.Code 3. Nested for loop with a hiCUDA pragma for outer loop

3.3 Determining number of threads (block size and number of blocks)

The number of threads required for the parallel execution of a loop are determined by the number of iterations of each loop.

For example, consider the following code snippet:

```
for( i = 0; i < n; i++) {
        sum = sum + a[i];
}
```

Code 4. Code for computing the sum of all elements of an array

Here, the number of threads required are (n - 0). Hence, for a block_size = 512, number_of_blocks = (n – 0)/512.

For nested loops, the dimensionality of block_size and number_of_blocks changes accordingly.

For example, consider the following code snippet:

```
for( i = 0; i < n; i++) {
        for( j = 0; j < m; j++) {
                a[i][j] = i * j;
}
}
```

Code 5. Sample code for initializing a 2-D array

Here, the number of threads required are (n - 0) * (m – 0). Hence, for a block_size = (16, 16), number_of_blocks = (((n – 0)/16), ((m – 0)/16)), which is indicated by the hicuda pragma in the following code:

```
#pramga hicuda kernel kernel_name tblock(((n – 0)/16), ((m –
0)/16)) thread(16, 16)
#pramga hicuda loop_partition over_tblock over_thread
for( i = 0; i < n; i++) {
#pramga hicuda loop_partition over_tblock over_thread
        for( j = 0; j < m; j++) {
                a[i][j] = i * j;
}
}
```

Code 6. Nested for loops with hiCUDA pragmas

3.4 Memory allocation/de-allocation on GPU

Whenever a kernel is created, memory needs to be allocated (and then de-allocated) on the GPU, for all data variables which are accessed (read/write) inside the kernel. Hence additional information about the dimensionality of each array is maintained.

All variables accessed inside the loop instructions are categorized into two lists: the read list (if the variable is read from) and the write list (if the variable is written onto). These lists are then utilized to determine which variables will be "copyin" and which variables will be "copyout" from GPU memory.

For example, consider the following serial code:

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

Code 7. Sample code for calculating square of each element of array

Here a[] will only be "copyin" to the GPU memory while b[] will be both "copyin" and "copyout" from it, which is indicated by the following pragmas:

```
#pragma hicuda global alloc a[*] copyin
#pragma hicuda global alloc b[*] copyin
#pramga hicuda kernel kernel_name tblock((n – 0)/16)
thread(16)
```

```
#pramga hicuda loop_partition over_tblock over_thread
for( i = 0; i < n; i++) {
        b[i] = a[i] * a[i];
}
#pragma hicuda kernel_end
#pragma hicuda global copyout b[*]
#pragma hicuda global free a
#pragma hicuda global free b
```

Code 8. The complete hiCUDA code

4. RESULTS AND ANALYSIS

To test the effectiveness of this tool, it is important to quantify the following:

1. Performance of the generated parallel code v/s original serial code.
2. Performance of the automatic parallel code v/s a hand written best optimized parallel code.

The codes are run on an Intel Core2Duo 1.6 Ghz processor with 2GB RAM. NVIDIA's GeForce 8400 GS graphics card is used for GPU's.

4.1 Measuring speedup obtained by parallelization

To measure the speedup obtained by parallelizing serial codes, the execution time of various input serial C codes are compared with the execution time of the corresponding auto generated parallel CUDA codes.

4.1.1 Matrix Multiplication

First of all, the standard problem of matrix multiplication is considered. Two matrices, a and b are initialised as follows:

$a[i][j] = (i + j) \wedge N$
$b[i][j] = (i - j) \wedge N$
where, a and b are of size N*N.

Then, the resultant c matrix is obtained by multiplying matrices a and b.

$$c[i][j] = \sum_{k=1}^{N} a[i][k] * b[k][j]$$

Figure 3 and Table 1 show the speedup obtained by parallelizing a serial code for this problem. For a moderate matrix size, N = 1024, the speedup obtained is 5-6 times.

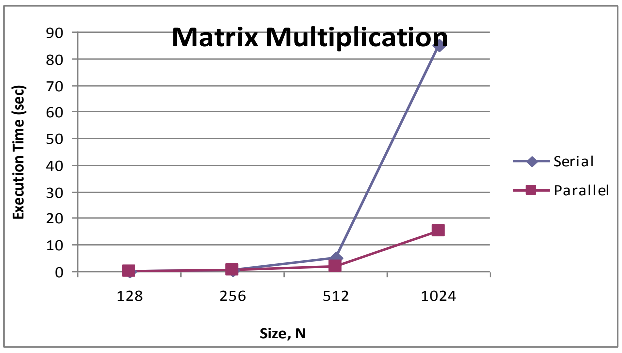

Fig 3: Performance comparison of serial and parallel matrix multiplication codes

Table 1. Execution times for serial and parallel matrix multiplication codes

Size, N	Serial	Parallel
128	0.08	0.15
256	0.6	0.4
512	5	2
1024	85	15

4.1.2 Compute Power Array

The problem of computing the power vector is considered next. Each element of vector a is calculated by using the following formula:

$a[i] = i \wedge N$
where, N is the size of the vector a.

Since, each element is independent of others, this task can easily be parallelized. The proposed tool is used to parallelize the serial code and the speedup obtained is indicated in Figure 4 and Table 2.

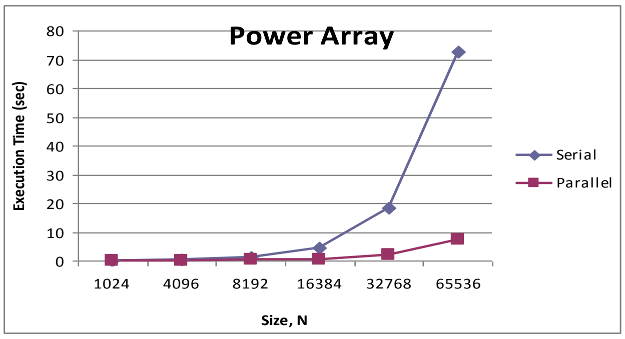

Fig 4: Performance comparison of serial and parallel power array codes

Table 2. Execution times for serial and parallel power array codes

Size, N	Serial	Parallel
1024	0.02	0.141
4096	0.287	0.172
8192	1.142	0.243
16384	4.552	0.584
32768	18.201	1.942
65536	72.8	7.347

In this case, the speedup obtained is ~10 times.

The speedup, in this case, is greater than that for matrix multiplication, which is as expected. A high cost is paid in transferring data from CPU to GPU (and vice-versa), so the computation for each GPU thread should be long enough to justify the overhead transfer costs. In matrix multiplication, for the second kernel (computing c[i]), each thread performs only a single operation of multiplying a single element of matrix a[] with that of b[], which is unable to compensate for the cost of threads and kernel creation.

4.1.3 Calculate the Prime divisors

Next, we consider a modular code, where each task is divided amongst various functions. We consider the problem of finding the prime divisors of a given number n. Different functions are responsible for finding whether an integer 1 < i < n is a prime number and whether it completely divides the number n. This tool successfully parallelises this code and the speedup obtained is shown by Figure 5 and Table 3.

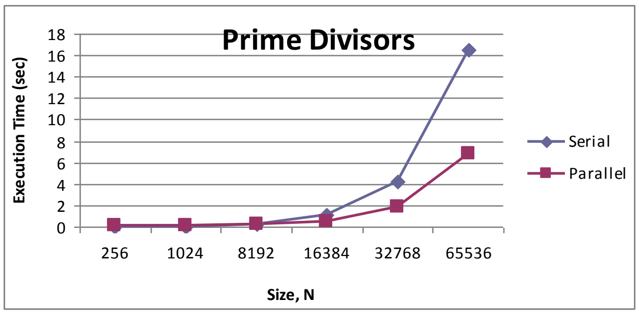

Fig 5: Performance comparison of serial and parallel prime divisors codes

Table 3. Execution times for serial and parallel prime divisors codes

Size, N	Serial	Parallel
256	0.002	0.101
1024	0.006	0.128
8192	0.278	0.212
16384	1.056	0.554
32768	4.162	1.815
65536	16.55	6.82

4.2 Comparing generated parallel code with hand-written code

To get hand written CUDA code, we use the hiCUDA compiler to generate CUDA code from a hand written hiCUDA code.

A hand written parallel code is expected to outperform the automatically generated parallel code in a few cases, but here we document by how much they outperform and the reasons for the same.

4.2.1 Matrix Multiplication

The first code we consider in this section is the matrix multiplication code (which is the same code as in section 4.1.1). When we write the code manually for this specific problem, we can gain some advantage by utilizing the shared memory on GPU's, which is ignored by the generic tool. The data needed by all threads in a thread block can be loaded into the shared memory before they are used, reducing access latency to memory. This can be done by using the following hiCUDA pragma's:

```
#pragma hicuda shared alloc A[*][*] copyin
#pragma hicuda shared remove A
```

Since the amount of data is too large to fit in the shared memory at once, it must be loaded and processed in batches. For this problem, we store 32 elements of each matrix in the shared matrix at a moment. The difference in the automatically generated and the hand-written code is shown in Codes 9 and 10.

```
for(kk=0; kk<N2; kk+=32) {
#pragma hicuda shared alloc A[i][kk:kk+32] copyin
#pragma hicuda shared alloc B[kk:kk+32][j] copyin
#pragma hicuda barrier
        for (k = 0; k < 32; ++k) {
                sum = sum + A[i][kk+k] * B[kk+k][j];
                sum = sum % 100000000;
}
#pragma hicuda barrier
#pragma hicuda shared remove A B
}
```

Code 9. The hand-written code

```
for(k=0; k<N2; k++) {
        sum = sum + A[i][k] * B[k][j];
        sum = sum % 100000000;
}
```

Code 10. The automatically generated code

Figure 6 and Table 4 show a comparison of the running times of both the codes.

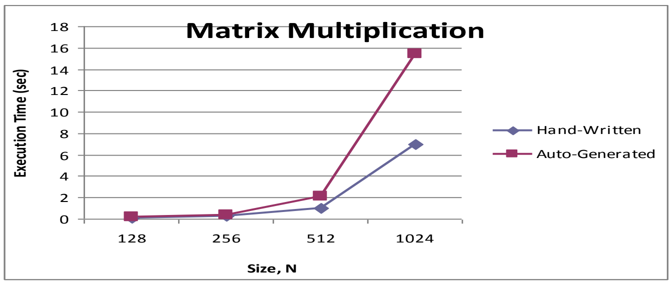

Fig 6: Performance comparison of hand-written and auto-generated matrix multiplication codes

Table 4. Execution times for hand-written and auto-generated matrix multiplication codes

Size, N	Hand-Written	Auto-Generated
128	0.122	0.177
256	0.252	0.344
512	1.002	2.073
1024	6.984	15.47

4.2.2 Vector multiplication

Another area where a hand written code can beat the automatically generated code is by combining multiple kernels together. We use the standard vector multiplication algorithm in this section.

The first kernel, initializes a[] and b[] vectors(or arrays) as power arrays:

a[i] = i ^ N;
b[i] = (N - i) ^ N;

The next kernel calculates c vector as follows:

*c[i] = a[i] * b[i];*

While these tasks are performed by different kernels in the automatically generated parallel code, they are combined into a single kernel in the hand written code. Performance gains are obtained by eliminating the memory transfer instructions. In the automatically generated code, a[] and b[] vectors are first copied from GPU memory into the CPU memory after the execution of first kernel. These are then successively copied back to the GPU memory for the beginning of the second kernel. A comparison of both these codes reveals that the performance gain is not very significant, as shown in Figure 7 and Table 5.

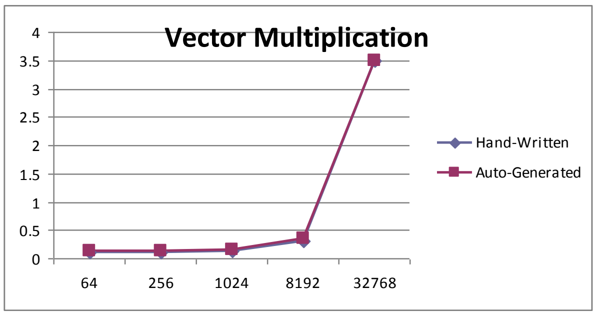

Fig 11: Performance comparison of hand-written and auto-generated vector multiplication codes

Table 5. Execution times for hand-written and auto-generated vector multiplication codes

Size, N	Hand-Written	Auto-Generated
64	0.105	0.139
256	0.103	0.136
1024	0.124	0.143
8192	0.313	0.344
32768	3.488	3.499

5. CONCLUSION AND FUTURE WORK

A working end-to-end tool has successfully been developed and tested over a wide range of codes. The performance results obtained are very satisfying with speedup gains obtained of up to 10 times.

The *automatic* parallelization, we believe, is a very significant step forward and would help the industrial community immensely. It being a *generic* tool capable of handling most kinds of C codes increases its worth.

This good performance only motivates us to improve it further. The ultimate aim is that the tool should be able to parse all legacy C codes. The parsing technique needs to be improved for this. We can also look at some dedicated parsing tools available for C code such as Elsa [19].

Also, the tool uses static analysis to detect data independency, that is, it reads the code as a simple text. On the other hand, if it used dynamic analysis, wherein the code is actually executed and the runtime memory accesses monitored, it would have enabled handling pointers too. But that would have also resulted in the execution time of the serial code being a bottleneck in the parallelization process.

Presently, each set of nested for loops in the C code are combined together to form an independent kernel for the GPU. In cases, such as in matrix multiplication, where we have two consecutive kernels one directly after the other and when the output of one is an input for the next, we can combine the two kernels and remove the unnecessary memory transfer instructions in between them. This would require another pass over an intermediate hiCUDA code.

To obtain maximum speed up, the tool will have to deal with the shared and textured memory of GPU as well. It, at present, deals with only the global GPU memory

6. ACKNOWLEDGMENT

The authors would like to thank Mr. Tarun Beri (IIT Delhi) for sharing his invaluable knowledge of the field and discussing the ideas of the authors and even giving some of his own.

7. REFERENCES

[1] NVIDIA, NVIDIA CUDA Compute Unified Device Architecture-Programming Guide, Version 3, 2010.
[2] Stone, J.E., Gohara, D., Guochun Shi, “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems”, Computing in Science and Engineering, Vol. 12, Issue 3, pp. 66-73, May 2010
[3] E. Alerstam, T. Svensson and S. Andersson-Engels, "Parallel computing with graphics processing units for high speed Monte Carlo simulation of photon migration" , J. Biomedical Optics **13**, 060504 (2008).
[4] Larsen E. S., Mcallister D., “Fast matrix multiplies using graphics hardware”, Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, Nov. 2001, pp. 55.
[5] Vladimir Glavtchev, Pinar Muyan-Ozcelik, Jeffrey M. Ota, John D. Owens, "Feature-Based Speed Limit Sign Detection Using a Graphics Processing Unit", IEEE Intelligent Vehicles, 2011.
[6] Woetzel J., Koch R., “Multi-camera realtime depth estimation with discontinuity handling on PC graphics hardware”, Proceedings of the 17th International Conference on Pattern Recognition (Aug. 2004), pp. 741–744.
[7] Rumpf M., Strzodka R., “Level set segmentation in graphics hardware”, Proceedings of the IEEE International Conference on Image Processing (ICIP ’01), Oct. 2001, vol. 3, pp. 1103–1106.

[8] Purcell T. J., Buck I., Mark W. R., Hanrahan P., “Ray tracing on programmable graphics hardware”, ACM Transactions on Graphics 21, 3 (July 2002), pp 703–712.

[9] Knott D., Pai D. K., “CInDeR: Collision and interference detection in real-time using graphics hardware”, Proceedings of the 2003 Conference on Graphics Interface, June 2003, pp. 73–80.

[10] Svetlin A. Manavski, "Cuda compatible GPU as an efficient hardware accelerator for AES cryptography" Proc. IEEE International Conference on Signal Processing and Communication, ICSPC 2007, (Dubai, United Arab Emirates), November 2007, pp.65-68.

[11] T. D. Han and T. S. Abdelrahman, "hiCUDA: High-Level GPGPU Programming", IEEE Transactions on Parallel and Distributed Systems, Jan. 2011, vol. 22, no. 1, pp. 78-90.

[12] David B. Loveman, “High Performance Fortran”, IEEE Parallel & Distributed Technology: Systems & Technology, February 1993, v.1 n.1, pp 25-42.

[13] Leonardo Dagum and Ramesh Menon, “OpenMP: An industry-standard API for shared-memory programming”, IEEE Computational Science and Engineering, 5(1):46–55, January–March 1998.

[14] VectorFabrics. vfAnalyst: Analyze your sequential C code to create an optimized parallel implementation. http://www.vectorfabrics.com/.

[15] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S.-W. Liao, E. Bugnion, and M. Lam, “Maximizing multiprocessor performance with the SUIF compiler”, IEEE Comput. 29, 12, Dec. 1996, pp 84–89.

[16] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. “Advanced Program Restructuring for High-Performance Computers with Polaris”, IEEE Computer, December 1996, Vol. 29, No. 12, pages 78- 82.

[17] Johnson, S.P., Evans, E., Jin, H., Ierotheou, C.S., “The ParaWise Expert Assistant—Widening accessibility to efficient and scalable tool generated OpenMP code”, WOMPAT, pp. 67–82 (2004).

[18] T.D. Han, “Directive-Based General-Purpose GPU Programming”, master’s thesis, Univ. of Toronto, Sept. 2009.

[19] Elsa: The Elkhound-based C/C++ Parser. http://www.scottmcpeak.com/elkhound/sources/elsa/