

Modeling Parallel Programs for Heterogeneous Computing

Ferosh Jacob (Advisor: Dr. Jeff Gray)

Department of Computer Science

University of Alabama, AL- 35487

fjacob@crimson.ua.edu

ABSTRACT

With the growing interest in multicore processors and Graphics Processing Units (GPUs), heterogeneous computing is an emerging necessity to fully utilize computing resources. In the traditional approach to parallel programming, programmers are required to manage the sequential part of a program while rewriting the parallel part to a more efficient parallel programming model. Our evaluation across several benchmarks suggests that there are cases for various problem sizes and system configurations that are challenging even for an expert parallel programmer to adapt (i.e., predict which architecture will perform better without actually executing the program). This paper presents a new approach to model parallel programs such that programmers can more easily separate the parallel part of a program from its sequential part. The programmer is given the flexibility to rewrite the parallel part using the parallel programming model of his/her choice. We provide support for merging the parallel code with the existing sequential code. This allows the programmer to focus on the parallel nature of the program and also offers the flexibility of switching between parallel libraries based on the platform and the specific problem context. As an experimental assessment of our approach, we identified and studied 58 parallel programs written by expert programmers.

General Terms

Algorithms, Performance, Languages

Keywords

Parallel programming, Pmodel, Software Modeling

1. INTRODUCTION

Multicore processors have gained much popularity recently as semiconductor manufactures battle the “power wall” by introducing chips with two (dual) to four (quad) processors, as well as Graphics Processing Units (GPUs) that have numerous processors per chip. The expectations of increasing clock speed are no longer satisfiable, which is driving the recent trend toward an increase in the number of cores per chip. For a multicore processor with low clock speed to outperform a single core processor with higher clock speed, software must be written in a parallel manner to take advantage of the additional processing capabilities. For some selected parallel programs, a comparison of the x1900 series (GPU from AMD/ATI) to the dual core AMD Opteron CPU processors shows that there can be an order of

magnitude improvement in performance when programs are parallelized. For next generation applications, programmers will be required to adapt to a new style of programming to utilize the parallelism in the processors available.

Parallel programming can be defined as the creation of code for computations that can be executed simultaneously. Due to their highly parallelized structure, GPUs provide an excellent platform for executing parallel programs. For example, NVIDIA’s Computation Unified Device Architecture (CUDA), Microsoft’s Direct Compute, and Khronos Group’s OpenCL are the most commonly used frameworks for General-Purpose GPU (GPGPU) programming. Parallel or GPU programming still requires skill beyond that of an average programmer. Currently, in order to write a program that will execute a block of code in parallel, a programmer must learn a parallel programming Application Programming Interface (API) that can be used to describe the computation. Even after the execution, the programmer must use other APIs or frameworks to evaluate the performance of their parallel program. As revealed from our analysis, the execution time of these programs depends not only on the platform, but also on the problem size.

The dependencies that emerge from the execution profile of a program usually define the complexity of the parallel code. A survey of general-purpose computation on graphics hardware reveals that GPGPU algorithms continue to be developed for a wide range of problems [1]. To use GPGPUs outside of their intended context, much work is required to make such algorithms accessible to a broader range of software developers. Abstractions in parallel programming languages and directives or annotations in sequential code have shown initial promise in reducing some of the burden of parallel programming, as noted in Section V. However, even with all of these advances, parallel programming still requires skill beyond that possessed by an average application programmer.

In this paper, we describe our analysis of programs written by expert programmers using different parallel programming models, investigate what makes such programs hard to adapt, and propose a new way to retarget parallel programs. Section II describes a performance study of programs written in different platforms, with an analysis of selected OpenMP programs. A solution to some of the challenges of retargeting parallel programs is explained in Section III. Two examples are introduced in Section IV to demonstrate our approach. Related work is overviewed in Section V and a conclusion is offered in Section VI that enumerates possible extensions of the current work.

2. A STUDY ON PARALLEL PROGRAMS

In this section, we investigate the factors that make parallel programming hard. The study is based on two important factors: 1) Execution time: evaluate the performance of the same programs written in different platforms for various problem sizes, and 2) Parallel code: analyze the parallel code from parallel programs collected from different domains.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Conference’10, Month 1–2, 2010, City, State, Country.
Copyright 2010 ACM 1-58113-000-0/00/0010...\$10.00.

2.1 Execution analysis of NAS Benchmarks

The NAS Parallel Benchmarks (NPB) are a set of benchmarks developed and maintained by the NASA Advanced Supercomputing (NAS) group. For our experiment, we selected four benchmarks: 1) CG (Conjugate Gradient), a program to estimate the smallest eigenvalue of a matrix using the conjugate gradient method for solving linear equations, 2) FT (Fourier Transform), a program to solve three-dimensional partial differential equations using the Fast Fourier Transform (FFT), 3) EP (Embarrassingly Parallel), a program to generate independent Gaussian random variables using the Marsaglia polar method, and 4) BT (Block Tridiagonal), a program to solve non-linear partial differential equations using the block tridiagonal method.

Each problem was executed for different class sizes, denoted as (S, W, A, B). As an example, for the FT benchmark the class S is 64x64x64, class W is 128x128x32, class A is 256x256x128, and class B is 512x256x256. Each class was executed for 1, 2, 4, and 8 instances (threads in OpenMP and processes in MPI) to analyze the execution time. In total, there were 128 executions (4 (benchmark) * 4 (classes) * 4 (instances) * 2 (MPI and OpenMP)). The benchmark code was used without any modification for this experiment.

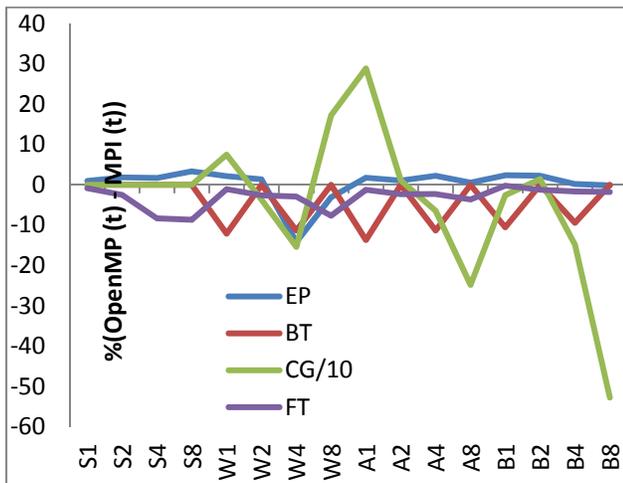


Figure 1 Execution time comparison of OpenMP, MPI for NAS benchmarks

The benchmarks were executed on an Intel Quad-Core i5 CPU with 4.0GHz running on Ubuntu 10.10. The results of the execution are shown in Figure 1. In this plot, we compare the execution time of the MPI program with the counterpart in OpenMP. The percentage difference in execution time is plotted for all of the four benchmarks across all four classes, each with varying instance. As an example, “A4” denotes the problem when executed with problem size A and instance 4 (4 threads in the case of OpenMP and 4 processes in the case of MPI). The MPI implementation of the BT benchmark expects the number of processes to be a square, and hence the execution time of the BT MPI benchmark for all classes with instance 2 and 8 was assumed to be the same as OpenMP. Graphs that are positive (EP) show that OpenMP execution time is higher for all classes and for all instances. Hence, in that case MPI is preferred. Graphs that are negative (BT, FT) show that the MPI implementation takes longer. Hence, for those problems OpenMP is a better option. Graphs that cross over the X axis (as in the case of CG and EP), suggest that both OpenMP and MPI can yield different results based on the size of the problem.

2.2 Parallel blocks in parallel program

An analysis was conducted for ten OpenMP programs collected from various domains. In an OpenMP program, a parallel block is defined by a compiler directive starting with `#pragma omp parallel`. The details of the analysis are shown in Table 1. The first column of the table shows the name of the program, second column shows the total Lines Of Code (LOC), third column shows the total LOC of the parallel block, and the last column shows the number of parallel blocks in each program. The LOC of the parallel blocks to the total LOC of the program ranges from 2% to 57%, with an average of 19% for the selected ten OpenMP programs.

Table 1 Parallel sections in OpenMP programs

No	Program Name	Total LOC	Parallel LOC	No. of blocks
1	2D Integral with Quadrature rule	601	11 (2%)	1
2	Linear algebra routine	557	28 (5%)	4
3	Random number generator	80	9 (11%)	1
4	Logical circuit satisfiability	157	37 (18%)	1
5	Dijkstra’s shortest path	201	37 (18%)	1
6	Fast Fourier Transform	278	51 (18%)	3
7	Integral with Quadrature rule	41	8 (19%)	1
8	Molecular dynamics	215	48 (22%)	4
9	Prime numbers	65	17 (26%)	1
10	Steady state heat equation	98	56 (57%)	3

2.3 Analysis conclusion

There are programs that work better in OpenMP for a given problem size, and others that are better in MPI for a different problem size. Because performance of the program is a function of the size of the problem and the programming model, there is often a need for having different programs that target multiple parallel programming models, such as OpenMP and MPI.

To create a different execution environment for a parallel program, more than 50% of the total LOC of code would need to be rewritten for most of the programs that we studied. The traditional approach to retargeting a parallel program is to manually copy/paste or rewrite the sequential section in the parallel program. To our knowledge, there is no existing support for creating or maintaining the sequential section while rewriting the parallel program for a new platform.

To compare the performance in other parallel programming models requires additional code to setup the execution, in addition to actually specifying the execution. In the case of OpenMP programs, additional code is required for the library declarations; for MPI, the library declarations are initialization and finalization code for process instance and process size variables; for GPUs,

additional code is required to copy a variable from CPU memory to GPU memory before execution, and copy the results back after the execution.

3. SOLUTION APPROACH: PPMODEL

Models are often created as a higher level abstraction of some system design [2]. Our research has led us to the realization of the benefits of adopting a modeling approach to address the challenges of parallel programming. The result of our work is a modeling environment called PPmodel, which has two goals: 1) to separate the parallel sections from the sequential parts of a program, and 2) to define a new execution strategy for the computation intensive part of the program without changing the flow of the program. Using PPmodel, the parallel part of the program can be separated from the sequential part of the program, re-designed, and then regenerated. With PPmodel, programmers can switch between technical solution spaces (e.g., MPI, OpenMP, CUDA and OpenCL) without actually changing the program. Our approach allows a programmer to concentrate more on the essence of the parallelization, rather than focusing on the accidental complexities of language-specific and library-specific details. A high level design diagram of PPmodel is shown in Figure 2. As shown, PPmodel acts as a Code Extractor as well as Code Refactorer.

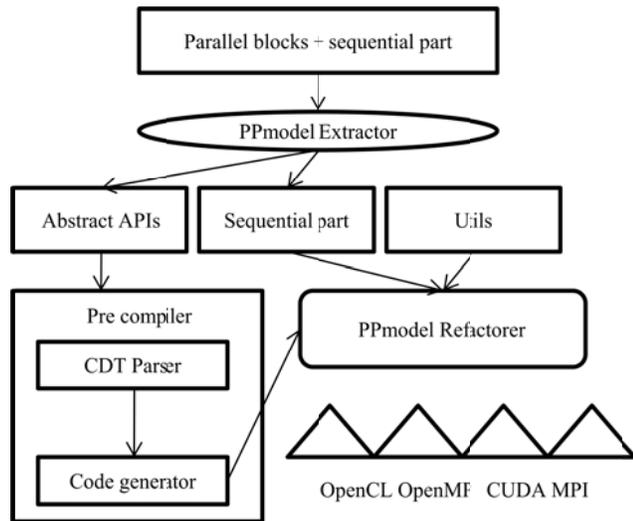


Figure 2 Design diagram of PPmodel with Abstract API

3.1 Code Extractor

In the Code Extractor stage, the parallel blocks from a parallel program are separated from its sequential part. Currently, PPmodel supports only 'C' programs written in OpenMP. These parallel blocks are later converted into AbstractAPIs. For MPI programs, AbstractAPIs are similar to an "Extract Function" [3] refactoring. In the case of a GPU programming language like CUDA, an AbstractAPI can abstract the code needed for copying the required variables from the CPU to the GPU, executing the kernel, and copying the results back to the GPU. More details about AbstractAPIs are mentioned in [4]. Generating AbstractAPIs from sequential code is explained in [5].

3.2 Code Refactoring

At the code refactoring stage, PPmodel gathers all of the information from AbstractAPIs and the sequential code to generate the final code for the new programming model. As an example, in the case of a matrix transpose, the code specified within a parallel block is parsed and the variables involved in the

block are extracted (e.g., `idata` is the input variable and `odata` is the output variable). Other details, such as the size and type of each variable, are also extracted and stored. With this information, the block can be replaced by a method call followed by a set of assignment statements. In the case of this example, the AbstractAPIcode generated would be:

```

copyin(idata, odata);
call(main, idata, height, width);
copyout(odata);

```

The first parameter of the call is the name of the parallel block and the remaining parameters represent the list of variables used in the block. Parameters for `copyout` are the variables in which the result is accumulated, and the parameters in `copyin` are the reference variables used. The variables declared within the parallel block (`xIndex`) are not included in any of the parameter list. These three lines of code form the abstract API are responsible for generating the host code in GPU programs or other parallel programming models. In the case of MPI, the `copyin` and `copyout` methods have no effect, but the call method is replaced by a function call that has an MPI implementation. In the case of GPU modeling, programmers implement the kernel, but in the case of MPI, an MPI method is implemented.

3.3 Modeling

We applied the concepts of Domain-Specific Modeling (DSM) [6] to facilitate the PPmodel modeling tool implementation. Considering the description about the structure of parallel programs as a specific application domain, the formal definition of the key abstractions for this domain, called the metamodel, must be defined first. The metamodel [7] specifies the entities, associations and constraints for the domain, and can be used to generate a modeling environment that enables users to build visual models that specify the structure of parallel programs. The models conform to the definition of the metamodel and can be used in computation, analysis and generation of software artifacts.

We used the Eclipse Modeling Framework (EMF), a metamodeling tool in Eclipse, to support the implementation of PPmodel. The metamodel consists of three components: 1) the abstract syntax of the structure of a parallel program is captured in the domain model, 2) the concrete syntax (i.e., the visualization with icons) is specified in the graphical model, and 3) the tooling model defines the functions of the editing environment (e.g., the palette, creation buttons, actions). The metamodel is used to generate the PPmodel modeling editor. The separation of domain model, graphical model and tooling model realizes an extensible framework, so that any changes to the visualization will not affect the domain concept definition and the editing environment, and any alteration of the domain model or tooling model will not force the other two definitions to change.

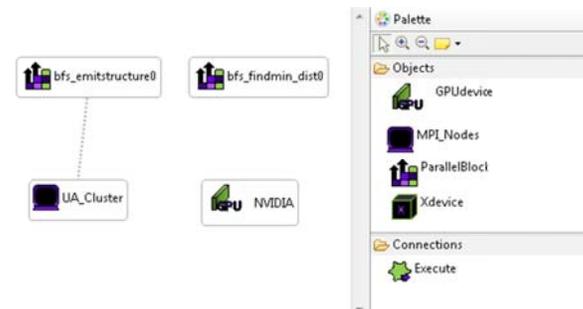


Figure 3 Modeling environment for parallel programs

4. PERFORMANCE ANALYSIS

This section describes an analysis of the execution time of two programs, each in two different programming models. A detailed explanation is given about the process for the first example. An online demo of the first example is available at the PPmodel project page¹.

4.1 Circuit Satisfiability problem

The Circuit Satisfiability problem has only one parallel block, with each thread determining whether the current value satisfies the given circuit equation. The parallel program written in OpenMP can be executed in a cluster using MPI by rewriting the parallel part of the program while keeping the sequential part of the program untouched. The following sections introduce the three stages of using PPmodel.

4.1.1 Model creation for Satisfiability problem

The model is created through a two-step process: 1) Model representation of the program is generated by right-clicking the program “Satisfy.c” in “Project Explorer” and selecting “ModelMe,” and 2) From the model representation, a visual representation of the program for a generated model is created by right-clicking the generated file, “_satisfy.parallelsystem” and selecting “DrawMe.” The model representation illustrates the parallel blocks of the program and the visual representation links the program to the target environment. It is possible to have different visualizations for the same program. The visualization model is a representation of the program in a particular configuration, specifying the target platform for each block.

4.1.2 Modeling the Circuit Satisfiability problem

Modeling helps the programmer to specify the automatically detected blocks to execute in a different platform. The modeling environment as shown in Figure 3 has a palette that consists of Objects and Connections. The Objects represent the set of nodes for modeling and Connections link a parallel block node with any of the execution devices. In Figure 3, an execution device can be a GPU device or MPI nodes. After creating a link between an execution device and a parallel block, a new file is created in the “generated” folder. In this file, a programmer can specify the implementation of the new execution strategy. In the case of a GPU node, this would be a kernel file or a C function.

4.1.3 Code generation for Satisfiability problem

In the editor, upon selecting “satisfy.parallelsystem_diagram” there is an option for code generation. This integrates the code written in “main_MPI.c” with the program in “_satisfy.c”. Code integration involves replacing the OpenMP code with the newly added MPI code, adding libraries to execute MPI code, and some code to initialize MPI-specific variables (e.g., process identifier, number of processes).

Execution time comparison

As noted in Section 2, the code was executed on an Intel Quad-Core i5 CPU with 4.0GHz running Ubuntu 10.10. The optimum performance can be achieved by using four threads for OpenMP and 4 processes in MPI. The execution time is plotted for a circuit satisfiability expression having 19 to 24 variables in Figure 4. As shown in the Figure, the MPI and OpenMP solutions perform the same for smaller problem sizes. However, as the problem size increases, MPI performs better than the OpenMP counterpart.

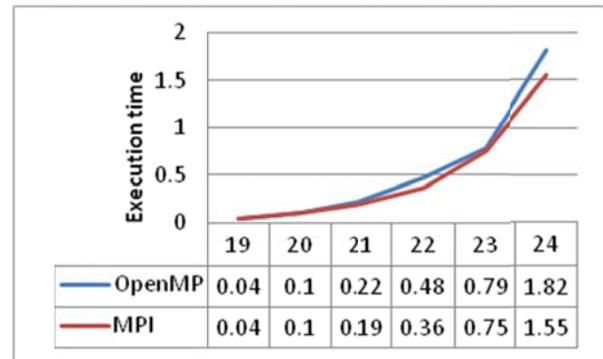


Figure 4 Execution plot of Satisfiability problem

4.2 Matrix multiplication and transpose

For matrix multiplication, the OpenMP programs were converted to a GPU programming model like CUDA. The kernel implementations were done similar to CUDA examples from the NVIDIA installation package, but using the PPmodel tool.

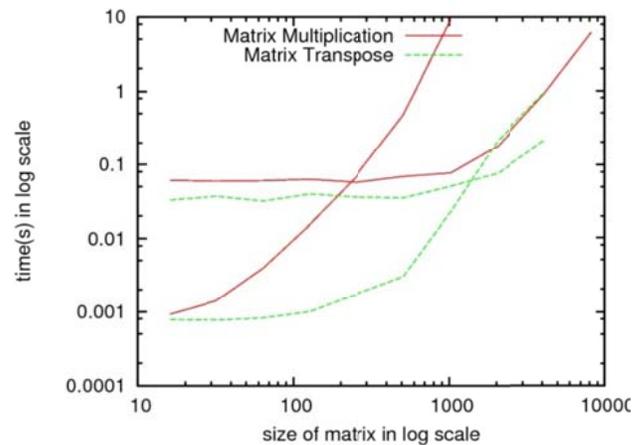


Figure 5 Execution plot for two matrix operations

The graph shown in Figure 5 is plotted for both shared memory (using OpenMP) and GPU (using CUDA) for two applications: matrix multiplication and matrix transpose. As expected, the shared memory (OpenMP) solution performs better for small problem sizes compared to using a GPU, which has a high threshold because of the expensive data transfer operations. As the problem size increases, the GPU programs become faster than shared memory programs. The problem size for which a GPU performs better than a CPU differs with the application, as seen in Figure 5 (i.e., the crossover size for matrix transpose is larger than the corresponding matrix size for matrix multiplication). This can be attributed to the problem complexity: In matrix transpose, no computations are involved; only data transfer. These differences in data size and configuration parameters of the problem make it challenging to select the correct language and platform for expressing the parallel computation.

5. RELATED WORK

There have been a few modeling efforts in the parallel programming domain using graphical programming languages, such as CODE [8] and GASPARD [9]. Rather than providing abstraction for a language from one parallel programming model to the other, modeling the parallel part of the program makes our work unique compared to these other efforts. OpenMP to GPGPU [10] converts OpenMP programs to CUDA code. However, the goal of our work is to express the parallel part of a program in a

¹ <http://cs.ua.edu/graduate/fjacob/ppmodel/>

way separated from the sequential part so as to allow the programmers to focus more on the parallel problem than the program as a whole. Other related works include program transformations from sequential to parallel and abstractions in parallel programs. Many of the sequential to parallel converters [11] use data dependency and refactoring approaches that are similar to our current implementation. Many efforts [12], [13], [14], [15] were done on the abstraction of GPU programs. Most of the work was concentrated on a particular device or language; for example, [12], [13], and [14] all target CUDA. CGiS [15] provides support for multiple devices. Some of the features include parallel control structures and special vector operators.

6. CONCLUSION

In this paper, we presented a tool named PPmodel that can separate the parallel part from the sequential part of a program. Using the modeling framework, programmers can execute the parallel blocks in a different platform without actually rewriting the program. The framework supports merging the newly written code back to the sequential part. Detailed analysis was conducted to determine that there is a need for rewriting the programs in different platforms to find out the optimum execution time of a program. An OpenMP program written to solve the Circuit Satisfiability problem was redesigned to execute on multiple nodes using MPI. Two matrix operations (matrix multiplication and matrix transpose) written in OpenMP were executed in CUDA by a programmer writing only the kernel part.

PPmodel currently can model only C OpenMP programs and generate target code for the MPI library and CUDA. Similar implementations can be created for other programming languages and platforms. The programming language determines the refactoring framework to use and the platform decides the code to be inserted or refactored.

7. REFERENCES

- [1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, pp. 80-113, 2007
- [2] J. M. Atlee, R. France, G. Georg, A. Moreira, B. Rumpe and S. Zschaler. "Modeling in Software Engineering," *Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, May 2007, pp. 113-114.
- [3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional,
- [4] F. Jacob, R. Arora, P. Bangalore, M. Mernik, and J. Gray, "Raising the level of abstraction of GPU-programming," *Proceedings of the 16th International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, pp. 339-345.
- [5] F. Jacob, D. Whittaker, S. Thapaliya, P. Bangalore, M. Mernik and J. Gray, "CUDA: A tool for CUDA and OpenCL Programmers," *Proceedings of the 17th International Conference on High Performance Computing*, Goa, India, December 2010, 11 pages.
- [6] J. Gray, J. Tolvanen, S. Kelly, A. Gokhale, S. Neema, and J. Sprinkle, "Domain-specific modeling," *Handbook of Dynamic System Modeling*, CRC Press, 2007
- [7] A. Lédeczi, A. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing domain-specific design environments," *IEEE Computer*, vol. 34, no. 11, 2001, pp. 44-51.
- [8] J.C. Browne, M. Azam, and S. Sobek, "CODE: A unified approach to parallel programming," *IEEE Software*, vol. 6, no. 4, 1989, pp.10-18.
- [9] F. Devin, P. Boulet, J. Dekeyser, P. Marquet, "GASPARD: A visual parallel programming environment," *Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, Warsaw, Poland, September 2002, pg. 145.
- [10] S. Lee, S. Min, J. Seung. And R. Eigenmann, "OpenMP to GPGPU: A compiler framework for automatic translation and optimization," *SIGPLAN Notices*, 44, February 2009, pp. 101-110.
- [11] R. Allen, and K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 4, 1987, pp. 491-542.
- [12] S. Ueng, M. Lathara, S. Bagsorkhi, and W. Hwu, "CUDA-Lite: reducing GPU programming complexity," *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, Edmonton, Canada, July 2008, pp. 1-15.
- [13] J. Breitbart, "CuPP: A framework for easy CUDA integration," *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009, pp. 1-8.
- [14] T. D. Han and T. S. Abdelrahman, "hiCUDA: A high-level directive-based language for GPU programming," *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units*, Washington, D.C, March 2009, pp. 52-61.
- [15] N. Fritz, P. Lucas, and P. Slusallek, "CGiS: A new language for data-parallel GPU programming," *Proceedings of the 9th International Workshop on Vision, Modeling, and Visualization*, Stanford, CA, November 2004, pp. 241-248.