HAMP –A Highly Abstracted and Modular Programming Paradigm for Expressing Parallel Programs on Heterogenous Platforms

Srinivas Balasubramanian, Prakash Raghavendra

Abstract— **With the start of the parallel computing era, due to power and thermal considerations, there is a growing need to bridge the gap between parallel hardware and software. The unintuitive nature of parallel programming and the high learning curve often prove a bottleneck in the development of quality parallel software. We propose HAMP – A Highly Abstracted and Modular Programming paradigm for expressing parallel programs. We provide the developer with high level modular constructs that can use to generate hardware specific optimized code. HAMP abstracts programs into important kernels and provides scheduling support to manage parallelism. By abstracting the scheduling and hardware features from the developer, we cannot only, considerably reduce the learning curve, but also increase software lifetime**

Keywords- Bottleneck code, highly abstracted, modular programming, parallel constructs, customized code

I. INTRODUCTION

Large scale parallelism is a given not only in GPU's but also CPU's in the near future[1][2][3] and the need of the hour is to develop efficient and scalable code that can slow down the software aging process. The transition from developing serially optimized code to scalable parallel code is not trivial and the associated learning curve proves to be a hindrance to most potential developers.[5] [6]

In the past, programming to exploit the architectural properties of a parallel machine was limited to a narrow field of experts. As the hardware industry shifted towards parallel computing, however, a wider field of programmers and algorithm developers needed to take advantage of these architectural innovations. With the right techniques, multicore architectures may be able to continue the exponential performance trend, which elevated the performance of applications of all types for decades. [7]

Srinivas Balasubramanian is with Department of Information Technology, NITK – Surathkal, Manglore, India. (email: srinivasbalasubramanian@acm.org)
Prakash Raghavendra is with the Department of Information Technology, NITK – Surathkal, Manglore, India. (email: spr@nitk.ac.in)

This paper addresses the gap between parallel hardware and parallel software development techniques and aims to bridge the same. The current approaches for auto parallelization of serial code, including static compile time analysis, seems to have hit a rough patch. Moreover the overhead associated with speculative parallelism [4], which involves monitoring the runtime behaviour of the program, leads us to believe that a highly abstracted and modular paradigm (HAMP) for parallel programming may hold the solution.

HAMP provides the following features to help bridge this gap. High level constructs that can be used to express the program. This is done by abstracting the program into important kernels and providing scheduling support to manage parallelism. HAMP differs from standard libraries as HAMP kernels are optimized for the target hardware and depending on the hardware where it needs to run, the compiler generates code, this ensures enhanced software lifetime and portability. HAMP uses extensive profiling of its kernels in order to determine optimal allocation of resources during simultaneous launch of kernels.

II. CURRENT CHALLENGES

Writing parallel code is a huge transition for most developers due to the instinctive serialized thought process. Due to the high cost involved with developing serial code and the large amount of legacy serial code, researchers started exploring automatic parallelization to make the development of scalable parallel code feasible. An effective framework that can help in development of quality parallel systems also needs to be addressed even though it only paves the path for the way ahead.

A. *Explicit Parallelism*

Automatic parallelization techniques that extract threads from single-threaded programs without programmer intervention do not suffer from these limitations and seemed as an obvious choice at first. Static analysis of the code at compile-time, looking for suitable ways to extract parallelism, was the initial trend. But the fact that even highly parallel applications could not be parallelized, due to lack of awareness of parallel programming in the part of the developer, led to search for alternative solutions.

B. Speculative Parallelism

Speculative parallelism does not need the code to have rigid constraints regarding dependencies as is the case in the static analysis. Dependencies that manifest in static analysis are speculated upon at runtime to extract as much parallelism as possible from them. If the speculation affects the correctness of the program suitable rollbacks are made to ensure a correct output is given. Speculative parallelism provided the solutions to most of the problems but the cost of speculation and lack of hardware support to roll back made it an expensive option. Software transactional memory was needed and the overhead and complication that spawned seem to have no elegant solution. Even though most of the instruction in a loop could be run in parallel 99 times out of 100 without altering the output, the 1 case where it failed proved to be very expensive and required large amount of effort and checks to ensure the output was validated [4].

C. Semi-Automatic Parallelism

Often many apparently easily parallelizable programs are unable to be parallelized automatically. The dependencies that inhibit parallelism are only rarely those required for correct program execution, more commonly manifesting from artificial constraints imposed by sequential execution models. This is because there exist alternative ways to express most code and most of the legacy code has not been written keeping a parallel architecture in mind. There is a simple alternative way of representing the same program finding a bypass around the dependencies which prevent parallelization in most cases. Semi automatic parallelization or tools which help to locate hot loops and suggest parallel transforms provide a temporary solution to this problem. These tools however still require the developer to have adequate knowledge of parallel programming in order to achieve scalable parallel programs.
Continuing exponential growth in transistor density and diminishing returns from the increasing transistor count have forced processor manufacturers to pack multiple processor cores onto a single chip. Multi-core processors generally do not improve the performance of single-threaded applications. There is a need to make parallel programming of applications feasible from a software engineering perspective. Automatic parallelization that is static has limited application while speculative parallelism has an associated overhead that makes it unattractive. The need to develop scalable parallel programs for applications is imminent.

III. Proposed System

Large scale parallelism is here to stay and the need of the hour is to develop efficient and scalable code. The transition from developing serially optimized code to scalable parallel code is non trivial and the associated learning curve proves to be a hindrance to most potential developers [5].The current approach for auto parallelization of serial code seems to have hit a rough patch. The limited scope of static analysis and the overhead associated with speculative parallelism [4] leads us to believe that a new approach is needed and we propose a highly abstracted and modular paradigm (HAMP) for parallel programming that we think may hold the solution.
HAMP takes its inspiration from design patterns. A variety of scientific applications today tend to utilise an overlapping set of operations. The main idea or inspiration for HAMP is that it's easier to find an implementation for a problem at an abstracted level then at the lower level. HAMP provides hardware specific implementations for a set of important kernels. The same HAMP code morphs itself depending on the target hardware we believe that feature is what separate HAMP from other high level heterogeneous programming languages. While generating the binary files HAMP does so based on the target hardware selected on the compiler. Our results show that higher the level of abstraction easier it is for the compiler to find implementations that are more efficient for a given problem. Also a higher level of abstraction reduces the learning curve. We also observed that code tuned for one hardware setup does not necessarily perform optimally on another hardware setup; therefore code generated should be hardware specific.

We aim to build a highly modular and abstracted system which makes extraction of functional parallelism easy. Data-parallelism will be implicit by having implementations of functions that inherently exploit the same.

The fact that serial programs that can easily be parallelized, can be coded in an obfuscated way that makes the detection of this inherent parallelism extremely difficult, made us think that if we can provide a framework like HAMP that provides the developer with enough to express his idea and leaves the best implementation of the idea to the compiler we have a solution to our problem. HAMP is inspired by the ideas of design patterns and the fact that there are a sizeable amount of problems that repeatedly show up in the industry [4] have a finite number of design patterns that can be used to represent and solve these problems . We believe that if we can map the users idea on to a finite number of computational methods that can be used to solve most of today's problems, the task of parallel programming becomes much easier. We try to show the benefits of the programming model of HAMP by exploring its benefits on scientific applications, the modules provided by HAMP can be compared to those provided by Mathworks- Matlab in the sense various well known functions have an efficient implementation that can be used in a simple and easy way.

The modular nature of the HAMP language allows us to have optimized implementations that can be customized and targeted for specific hardware. Parallel programming and especially GPGPU computing is facing a lot of resistance due to the lack of portability of optimized code [17] .Code that has been optimized for a particular hardware configuration may not show expected behaviour on other hardware configurations. From the perspective of the software industry there is a need to make the development process more standardized and streamlined while acknowledging the diversity of hardware that the application may eventually run on. Heterogeneous computing only adds to the developers woes in this respect; while the raw computational power made available is undeniable a feasible method to harness this power still eludes us. Not only is “the free lunch over” [5], the road ahead demands constant reinvention of the wheel for the multitudes of terrain that we might encounter. Even though the Holy Grail, “auto-parallelization” seems like the best solution, it is important that we acknowledge the paradigm shift in the nature of programming from a sequential to a parallel model. The non trivial nature of the problem becomes evident when we understand the fundamental difference in the two programming models.

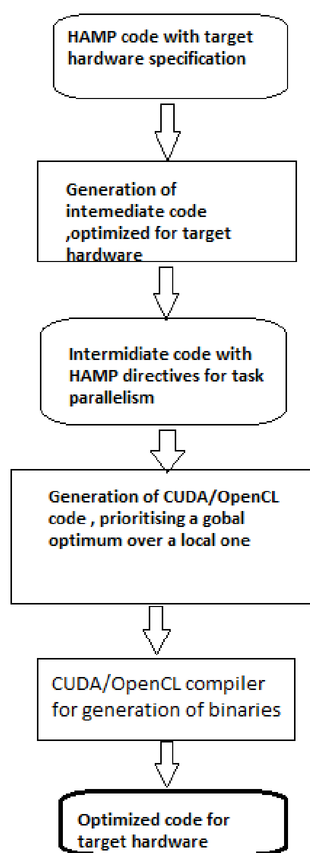

Figure 1 HAMP Workflow

Since each module of HAMP is standardized its behavior can undergo intensive profiling. The scalability of each module under predefined hardware conditions can be obtained and used to find an optimal scheduling as described in the following section.

An optimal scheduling of independent jobs: HAMP consists of different predefined modules that can be used to express a problem. The modular nature of the language makes it easier to identify coarse grain parallelism. Since different tasks have different levels of parallelism that can be extracted from them the scheduling of these independent tasks is of importance from the point of view minimizing the overall execution time of an application.

We define a bottleneck section as a section between two explicit global sync statements. We further analyze the scheduling of m independent jobs across n processors that are available for our bottleneck section. It is evident that the rate of completion of a task in the bottleneck section is determined by the maximum time taken by any of the m independent jobs.

BottleNeck_Tc= maximum (Ji_Tc) for i from 1 to m.

Where BottleNeck_Tc is the time taken to complete the bottleneck section and Ji_Tc is the time taken to complete the ith job. HAMP uses a longest job first greedy approach for allocation of the n processors across the m jobs. We assume large scale parallelism is available, implying the n is sufficiently larger than m. To begin with we allocate a single processor to each job and then allocate the next processor to the job currently taking the maximum time for completion. If there are multiple jobs with the same completion time the processor is allocated to the job which has the maximum reduction in its running time due to the addition of the processor. The behavior of the jobs on being provided additional processing power is made available by extensive profiling that has been carried out beforehand.

As shown in Figure 1, we specify the target hardware which allows us to generate intermediate code targeted for the hardware. We use information about the cache, number of processors, registry size etc to come up with suitable constants that give us values such as unroll factor, work group size etc. In the second stage we scan for task parallelism and alter these constants based on extensive profiling. We aim to get the best perform under the limited resources that has been made available to us. After necessary processing we come up with new constants which improve the performance of the group of tasks as a whole. Depending on our targeted hardware we then generate binaries using CUDA/ OpenCL or other third party compilers.

ISBN: 978-988-19252-1-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

The simultaneous launch of kernels that has become a new feature in heterogeneous computing also helps us exploit task parallelism. Parallel tasks that are competing for the same resource pool are scheduled internally using the longest job first scheduling algorithm. These parallel tasks are nothing but HAMP kernels that are independent of each other.

IV. Illustration of Concept with an Example

Consider the following example suppose we want to find the Number of paths in an undirected graph given its adjacency matrix. Further we impose a condition that we only want those paths whose length is a prime less than say a constant c and our graph is assumed to be a sparse graph. Writing this program in MPI, CUDA or Open CL code can be quite tedious but using HAMP our code will look something like this:

```
ReadSparseVector(A);
ReadConstant( c) ;
Sync;
K=GererateUniquePrimeLessThan ( c );
If(K) // there is a prime number not yet generated
and its value is K
{
B=MatrixPower(A,K); // Exponentiate matrix
S=MartixScanSum(B,K);// If B(I,j) is equal to K
add it to S
Answer=Answer+S ;
}
Else
END
Sync;
```

In the program flow the developer need only specify the order of invocation of various tools functions at his disposal as seen above. Current challenges like scheduling the threads on the processors, which make the code very cumbersome can be avoided. Further by abstracting the implementation we can ensure that a suitable solution for the problem which extracts maximum data parallelism in incorporated automatically in the code.

We notice that here GererateUniquePrimeLessThan(), Matrix power(), MartixScanSum(B,K) and the updating of the answer are all independent tasks that are sandwiched between sync statements. Each of these tasks can be performed independently and we may need to share the computational resources. This may alter the hardware specific optimizations that are present in the intermediate code which assumes that the entire hardware is at the disposal of each task. We use the method described in the previous section in order to allocate resources in an effective manner and thus come up with a solution that reduces the combined running times of the jobs listed above.

In the above example generating prime numbers less than a certain number provides scope for both pipeline parallelism and data parallelism. While choosing the next suitable number can be pipelined efficiently testing the primarily of a number can exploit both data and pipeline parallelism. Further synchronization and scheduling which requires an in-depth understating of parallel programming are hidden from the developer thus ensuring a low learning curve.

Our primary inspiration remains the fact that there are frequently occurring problems in the computer world and by documenting theirs solution; we can avoid reinventing the wheel. Even the reuse of code requires understanding of ideas behind parallel programming.

Sparse matrices are encountered very often in scientific applications today; there remain a variety of methods to go about sparse matrix multiplication depending on the distribution of non-zero elements. In our program as we have defined our input vector as a sparse matrix in line 1, the matrix power operation in line 5 is aware of the sparse matrix representation and uses a suitable algorithm to exploit the nature of the input while carrying out the exponentiation.

Further the exponentiation can be carried out in logarithmic time as opposed to the less effective trivial linear method. These ideas have already been proved to be computationally effective and will provide substantial scale-up in the application.

V. Results

Optimization techniques are numerous and the same technique may lead to varying results depending on the target hardware. General optimization techniques in a GPU involve modifying mappings, altering access patterns and modifying algorithms. Sometimes use of alternative functions may also provide an increase in code efficiency. We discuss our findings for the example considered in the previous section and the implication of our results as well. For generation of prime numbers we used the sieve of Eratosthenes here we noticed that even a parallel implementation is not necessarily better. A naive parallel

ISBN: 978-988-19252-1-3
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

implementation for the GPU in fact is outperformed by its serialized CPU counterpart. This shows that one needs to understand parallel algorithms well in order to harness its powers. The naive GPU version spawns off a thread for striking out the multiples of each number but this fails to exploit the parallelism offered by the GPU and is limited by the number of multiples of 2. We then modify the algorithm to overcome this implicit bottle neck.

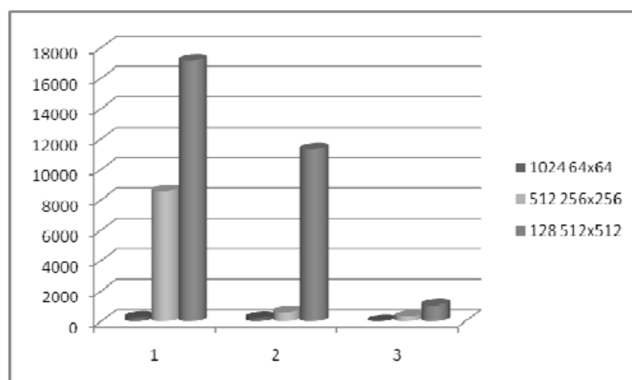

Figure 2. Time taken for Matrix Multiplication (ms)(lower is better)

Table 1. Time taken for Matrix Multiplication (ms)

Power	Matrix Size	Naïve MatMul	Optimized MatMul	Modified MatExp
1024	64x64	268	247	8
512	256x256	8589	635	323
128	512x512	17179	11338	1073

Table 2. Time for Sieve (ms)

Problem Size	CPU Sieve	Naive GPU Sieve	Modified GPU Sieve
2^{10}	11	54	5
2^{25}	1240	2240	231
2^{50}	66210	97520	4521

The use of various hardware dependent features like shared memory, texture memory etc to enhance caching effects. The use of optimal instructions per threads by use of methods such as loop unrolling or use of advanced instructions which modify number of cycles for execution can help reduce the execution time of the kernel. Choosing algorithms that best suit your hardware is also critical. Different algorithms have different levels of parallelism that can be extracted and different number of computations to carry out a specified task. We illustrate the above by implementing modules that help calculate the number of prime paths in a graph. Suitable implementations of different modules or building blocks that have been fine tuned for NVidia Tesla 1060c to carry out the above task help us illustrate our concept.

As seen in Table 1 hardware specific optimizations can reduce the running time by up to 35 percent. Use of an optimal work group size, exploitation of different levels of memory hierarchy and spawning an optimal number of threads specific to the hardware seems to have a positive effect on the efficiency of the code. The modified MatMul is an alternate algorithm that greatly reduces the number of computations needed to evaluate the Nth power of a matrix and gives a tremendous amount of speedup, almost 30 times, compared the Naïve MatMul available in the CUDA 3.2 Toolkit. The comparison of the running times is illustrated clearly in figure 2.

Table 2 illustrates a prime number sieve. We notice that it is not always necessary that a parallel algorithms out performs its serial counterpart. A naïve parallel implementation in fact takes more time than its CPU counterpart due to the fact that the amount of parallelism available is very less. But by modifying the access patterns we were able to a speedup of almost 14 times.

The modified MatExp shows that abstracting the task further, provides the compiler the opportunity to perform more optimizations and thus present a far superior solution. We also notice that even if individual components are optimized their combination need not be optimum. The task discussed in the example before when written using HAMP shows a 21.8 times speedup on a Tesla 1060c as compared to a naïve implementation. This illustrates the power of HAMP and strengthens our belief that it is a step forward in the right direction.

VI. Conclusion and Future Work

The results clearly indicate the advantages of parallel programming and the pitfalls it might pose to novice developers. This further strengthens our belief in HAMP which abstracts such details from developers and provides an easy and efficient way to develop good parallel code. We will continue to add more modular building blocks and fine tune it for multiple hardware platforms. HAMP is just in its inception stage, we hope to uncover more features that can help HAMP bridge the gap between existing technology and the need of the industry to exploit the same.

In the future we will be trying to further increase the scope of HAMP by expanding the modules and the targeted hardware that it is applicable for. We hope to expand into other domains and also provide optimized code for heterogeneous computational units using OpenCL. We will also need to carry out extensive profiling of existing components for different hardware configurations before using then in order to come up with a effective scheduling. Constant innovation in the front of GPGPU architecture will always leave us with scope for further improvement.

References

- [1] Amdahl’s Law in the Multicore Era, Mark D. Hill, University of Wisconsin-Madison, Michael R. Marty, Google.
- [2] From a Few Cores to Many: A Tera-scale Computing Research Overview by: Jim Held, Jerry Bautista, Sean Koehl
- [3] The Landscape of Parallel Computing Research: A View from Berkeley: Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick
- [4] A Design Pattern Language for Engineering (Parallel) Software:The key to writing high-quality parallel software is to develop a robust software design. By Kurt Keutzer and Tim Mattson, Dr. Dobb's Journal May 18, 2010
- [5] The Free Lunch Is Over :A Fundamental Turn Toward Concurrency in Software. By Herb Sutter
- [6] Understanding Parallel Performance. By Herb Sutter, October 31, 2008
- [7] Analysis of Benefits of various MATLAB functions for multithreaded processing. http://www.mathworks.com/support/solutions/en/data/1-4PG4AN/?solution=1-4PG4AN
- [8] When to use GPU for Matrix Operations. http://www.mathworks.com/help/toolbox/distcomp/bsic3by-1.html.
- [9] U. Bondhugula, M. Baskaran, A. Hartono, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Towards effective automatic parallelization for multicore systems. In Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, pages 1-5, April 2008.
- [10] B. Armstrong and R. Eigenmann. Application of Automatic Parallelization to Modern Challenges of Scientific Computing Industries. In ICPP, pages 279– 286,2008.
- [11] Programming Multicores: Do Applications Programmers Need to Write Explicitly Parallel Programs? Arvind, David I. August, Keshav Pingali, Derek Chiou, Resit Sendag, and Joshua J. Yi IEEE Micro, Volume 30, Number 3, May 2010.
- [12] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In Proceedings of the International Conference on Compiler Construction (ETAPS CC'10), lncs, Cyprus, March 2010. Springer-Verlag.
- [13] H. Vandierendonck, S. Rul, and K. D. Bosschere. The paralax infrastructure: Automatic parallelization with a helping hand. In Proc. of PACT, 2010.
- [14] S. Sethumadhavan, N. Arora, R. B. Ganpathi, J. Demme, and G. Kaiser. COMPASS: Community Driven Parallelization Advisor for Sequential Software. In 2nd Intl. Workshop on Multicore Software Engg., 2009.
- [15] LAMPVIEW: A Loop-Aware Toolset for Facilitating Parallelization,Thomas Rorie Mason, Master's Thesis, Department of Electrical Engineering, Princeton University, August 2009.
- [16] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The Common Case Transactional Behavior of Multithreaded Programs. In the Proceedings of the 12th Intl. Conference on High- Performance Computer Architecture (HPCA), Austin, TX, Feb. 2006.
- [17] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, "Evaluating performance and portability of OpenCL programs," in The Fifth International Workshop on Automatic Performance Tuning (iWAPT2010), 2010.