Automatically Tuned Dense Linear Algebra for Multicore+GPU

Xing Fu, Xue Li, Gregory D. Peterson
Department of Electrical Engineering and Computer Science
The University of Tennessee - Knoxville
[xfu1, xli44, gdp]@utk.edu

ABSTRACT

The Multicore+GPU architecture has been adopted in some of the fastest supercomputers listed on the TOP500. The MAGMA project aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures processors like Multicore+GPU. However, to provide portable performance, manual parameter tuning is required. This paper presents automatically tuned LU factorization. The key parameter of LU factorization is tuned automatically to optimize performance for a particular GPU platform. Moreover, we propose a work stealing scheme and GREEN-synchronization to decrease the power consumption of the LU factorization and accelerate the entire application.

KEYWORDS

Multi-core, GPU, automatic tuning, power efficiency

I. INTRODUCTION

Recently, supercomputers based on the Multicore+GPU architecture (e.g. Nebulae Dawning which is No.2 on TOP500 list [1]) have been deployed. The Multicore+GPU architecture is a promising technology trend for future supercomputers. In the MAGMA project [2], LAPACK is ported to Multicore+GPU. It is reported [3] that highly optimized LU, QR, and Cholesky factorization can achieve about 300Gflops for large matrices on GTX280 +Core2 Duo. The optimized implementation of the three factorizations achieves this rate due to a deeper performance analysis and tuning which can be very labor-intensive. Motivated by the ATLAS project, in this work we develop the technique to automatically identify key parameters of the functions implemented in MAGMA. Specifically, we present the automatically tuned LU factorization. Although we investigate only one function, the methodology can be applied to other functions and paves the way for future automatically tuned functions.

A significant contribution of this work is that the automatically tuned LU factorization is not only optimized for higher flops for various Multicore+GPU platforms. The performance acceleration [5][6] is the first priority, but the power issue is also important for supercomputers. We proposed a power-efficient synchronization mechanism called GREEN-synchronization between Multicores and GPUs. Power management techniques have been investigated extensively. In this work, we propose a control-based implementation of synchronizations to reduce power consumption.

Section II presents our methodology to tune the LU factorization for higher throughput. Section III presents our control-theoretic approach to tune the LU factorization for less power consumption. Section IV concludes the paper.

II. PERFORMANCE TUNING

Our automatically tuned methodology is based on the current fastest implementation of LU factorization on Multi-core and GPU platforms [3]. The source code of the LU factorization [7] is omitted due to page limits. The following table lists the main functions inside the LU factorization and where they are executed.

Function names	CPU	GPU
gpu_transpose_inplace		●
gpu_transpose		●
gpu_strsmRUNU		●
gpu_sgemmNN		●
sgetrf	●	
gpu_batch_sswap		●

Table 1 Breakdown of LU factorization

When the matrix order is large, most of the LU decomposition time is spent in functions executing on the GPU. However, gpu_strsmRUNU and gpu_sgemmNN are non-tunable. For the functions gpu_transpose_inplace and gpu_transposem, the BLOCK_SIZE parameter needs to be tuned manually. This parameter is used for the memory stride and the dimension of thread blocks. The source code excerpt is as follow.

(1) for(int i = 0; i < 32; i += BLOCK_SIZE)

(2) dim3 threads(32, BLOCK_SIZE, 1);

This parameter is critical to the performance of the LU function. The similar SWAPS_PER_RUN and VL parameters can be found in gpu_batch_sswap. These parameters are platform specific. To provide platform portability, it is necessary to configure the values according to the results of scripts. We vary the BLOCK_SIZE of gpu_transpose to transpose a 16384 × 16384 dense square matrix.

Figure 1 shows that the optimal BLOCK_SIZE for a Quadro FX 3800 GPU. Suboptimal parameters may lead to 10Gflops performance loss. It is important to tune the parameters of dense linear algebra for different platforms to get best performance. Applying the technique to other subroutines is left to our future work.

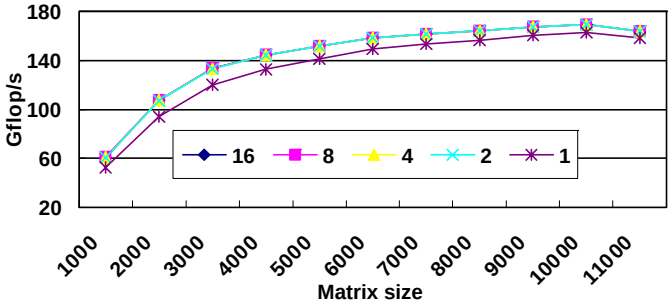

Figure 1 Tuning of BLOCK_SIZE for optimal performance

III. POWER TUNING

A short-coming of the state-of-art LU factorization in [3] is that the CPU launches GPU kernels, and then waits for the completion of the kernels. The CPU is completely idle in this period. We exploit this period to save the power consumption of the CPU. One common supported power management mechanism is dynamic frequency and voltage scaling (e.g. Enhanced Intel® Speedstep Technology [8]). We can tune to CPU to the lowest power consumption level (lowest frequency level) right before launching the GPU kernel and then back to the highest performance level (highest frequency level) after the GPU kernels complete. The power consumption will be approximated as a cubic function of CPU frequencies [9] and is shown in Figure 2. As we can see, by running the CPU at the lowest frequency can significant (about 70%) power consumption of the CPU.

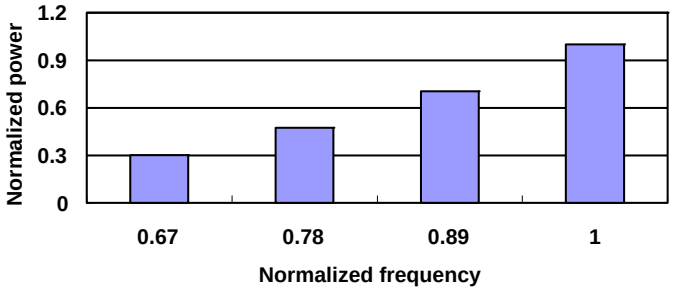

Figure 2 Impact of frequency on CPU power consumption

In this work, we propose a work stealing scheme to accelerate the application and GREEN-synchronization to save the power consumption of CPU. The schemes work as follows:

(1) Instead of simply shifting all work from CPU to GPU, the CPU spawns another CPU thread which steals a small fraction of GPU work to accelerate the entire application. In contrast, for matrix multiplication in the highly optimized LU factorization in [3], all multiplications are done in the GPU while the CPU does nothing.

(2) However, after stealing work from the GPU, the CPU and GPU need to be synchronized to ensure all work is done at some time points. We assign work so that the CPU will finish its part earlier than the GPU because the CPU is less computationally efficient than the GPU. The GPU needs to do as much as work as possible.

(3) Since the CPU will finish its part earlier and become idle, we can exploit the early completion of the CPU to save power consumption by designing a control loop.

The goal of the control loop is to predict the frequency level used to make the job stolen by the CPU to finish at the same time as the job executed on the GPU. We call the controller loop a frequency predictor. The key *advantage* of this control-theoretic approach is that we do not need the exact relationship between execution time and CPU frequency. By the control loop, the LU factorization can be self-tuned to save CPU power consumption while accelerating the entire application by dividing work among the GPU and CPU.

Note that how much work the CPU steals is a parameter set specifically by the end-user. The more work is stolen, the earlier the entire application will finish. However, less power will be saved. The extreme case is that the CPU and GPU finish their parts at the same time when the CPU always runs at the highest frequency. The other extreme case is that all work is done on the GPU, the entire application will finish later, but the CPU can always run at the lowest frequency which saves significant power.

The frequency predictor illustrated in Figure 3 will be invoked when the LU factorization function is invoked. The desired execution time s is selected to be the execution time of the job executed on the GPU. This information can be obtained very easily by execution time profiling functions provided by the CUDA SDK. We set the frequency of the CPU to be the highest frequency when the LU factorization is invoked for the first time. After first invoking the LU factorization, we will get the actual execution time of the stolen job on the CPU. We calculate the error $e(k) = s - y(k)$ and $k = 1$ which denotes the first invocation of the LU factorization function. The $e(k)$ will be input to the controller. Following standard control theory design, we design a controller which is only characterized by one parameter, so the overhead of the controller is very small. The controller will predict the frequency $f(k)$ and throttle the CPU frequency accordingly at the next time when the LU factorization function is invoked.

The system diagram of the control loop is:

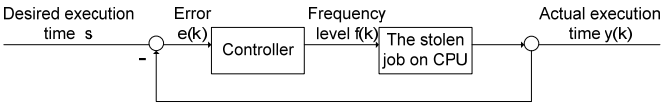

Figure 3 the system diagram of the frequency predictor.

It can be shown theoretically that the predicted frequency will guarantee the job stolen by the CPU and job executed on the GPU finish at the same time. It can also be proved that no matter the function, only one controller is needed for many dense linear algebra functions. We do not need to design many controllers. All those dense linear algebra functions

incorporated with controllers can be self-tuned to save power consumption of CPU while accelerating the entire application. The detailed analysis is omitted due to page limits. Interested readers can refer to [9].

IV. CONCLUSION

In this paper, we explore automatically tuned dense linear algebra for Multicore+GPU. As an example, we present an automatic way to identify key parameters of LU factorization to provide platform portability. We proposed a work stealing scheme and GREEN synchronization to speed up the LU factorization based on the state-of-art implementation [3] to save the CPU power consumption. This work paves the way for automatically tuned dense linear algebra which is highly desired by the Multicore+GPU architecture of future supercomputers.

V. REFERENCES

[1] TOP500 supercomputer site. http://www.top500.org/

[2] Matrix Algebra on GPU and Multicore Architectures http://icl.cs.utk.edu/magma/index.html

[3] V. Volkov and J. Demmel, Benchmarking GPUs to tune dense linear algebra, ACM/IEEE conference on Supercomputing ,2008, Piscataway, NJ, USA.

[4] Automatically Tuned Linear Algebra Software (ATLAS) http://math-atlas.sourceforge.net/.

[5] Akila Gothandaraman, Rick Weber, Gregory Peterson, Robert Hinde, and Robert Harrison, Architectural Comparisons for a Quantum Monte Carlo Application, Symposium on Application Accelerators in High Performance Computing, 2009 , Urbana, IL, USA.

[6] David Jenkins and Gregory Peterson, Accelerating the Stochastic Simulation Algorithm, Symposium on Application Accelerators in High Performance Computing, 2009, Urbana, IL, USA.

[7] LU, QR and Cholesky factorizations using GPU http://forums.nvidia.com/index.php?showtopic=89084

[8] Enhanced Intel® SpeedStep® Technology for the Intel® Pentium® M Processor ftp://download.intel.com/design/network/papers/30117401.pdf

[9] Xing Fu, Xiaorui Wang, and Eric Puster, Dynamic Thermal and Timeliness Guarantees for Distributed Real-Time Embedded Systems, IEEE International Conference on Embedded and Real-Time Computing Systems and Applications,2009, Beijing, China