Exploiting Multi- and Many-core Parallelism for Accelerating Image Compression

Cheong Ghil Kim
Dept. of Computer Science
Namseoul University
Cheonan, Chungnam, Korea
cgkim@nsu.ac.kr

Yong Soo Choi
GSIMS
Korea University
Seoul, Korea
ciechoi@korea.ac.kr

***Abstract*—With the help of recent development on semiconductor design and process technologies modern processors can provide a great opportunity to increase the performance of processing multimedia data by exploiting task- and data-parallelism in heterogeneous system consisting of multi-core CPUs and many-core GPUs (graphical processing units). This paper presents an optimization of 2D DCT (discrete cosine transform), a computation intensive signal processing algorithm widely used in compression standards, on speed in multi-core CPUs and many-core GPUs. Two optimization techniques using Intel TBB (threading building blocks) and OpenCL are discussed in detail. OpenCL is a recent open standard proposed to provide universal APIs and programming paradigms for various GPUs and accelerators; it can provide massively parallel processing suitable for data intensive multimedia applications with very low cost. The simulation result that the parallel DCT implementations are performed considerably faster than the serial ones, max 4.8 and 6.9 times speedup as for TBB and OpenCL, respectively. Especially, OpenCL implementation on GPU shows a linear speedup, a typical characteristic of massively parallel processing, as the increase of 2D data sets.**

Keywords-mult-core; GPU; TBB; OpenCL; task-level parallelsim; massively parallel programming; DCT

I. INTRODUCTION

Over the past few decades, modern microprocessors have competed with increasing clock rates and putting more transistors in single core processors. However, this trend is facing the problem of power dissipation which is proportional to both clock speed and transistor count. In consequence, multi-core CPUs and dedicated accelerator processors such as GPUs have become the dominant alternative solution for improving processor performance. This trend enables a single chip to increase the performance capability with neither requiring a complex system nor increasing the power requirements. Therefore, a major consequence is that parallel computing techniques such as incorporating multiple processing cores and other acceleration technologies are increasing in importance [1]. In order to take advantage of multi-cores, programs could be just written to accomplish their tasks using multiple parallel threads of execution. However, in order to make use of GPUs with many cores of simple processing capability, a different parallelism, massively parallel programming, is required. That is, a solution of easy and efficient parallel programming for heterogeneous computing systems consisting of multi-core CPU and many-core GPUs is strongly required.

OpenCL [2] is a new industry standard for task-parallel and data-parallel heterogeneous computing on a variety of modern CPUs, GPUs, DSPs, and other microprocessor designs. This is a software development infrastructure in the form of parallel programming languages and subroutine libraries that can support heterogeneous computing on multiple vendors' hardware platforms [3]. In order to program GPUs people have to use a specific hardware dedicated graphic API such as Cg, BrookGPU, Sh library and Stream, and CUDA. Of course, there are languages which allow general parallel programming, for example, OpenMP and MPI. Unfortunately, they are designed for specific platforms and hard to be used in GPUs; that is, they are not suitable for these GPU-based heterogeneous system platforms. Therefore, an open standard, OpenCL, is proposed for programming with different GPUs and accelerators including multi-core CPU without modifying computing kernels.

In this study, we describe parallel implementations of DCT (discrete cosine transform), a computation intensive signal processing algorithm that are widely used in compression standards, using multi- and many-core parallelism. This computation can be characterized as data intensive tasks accompanied by heavy memory access; on the other hand, its computational complexities are relatively low. In order to show the performance improvements of DCT utilizing these parallelisms, the programs are implemented using Intel TBB and OpenCL and targeted on multi-core CPU and several GPUs on desktop PC. Various different implementations with several computation attributes are included to demonstrate the capabilities of OpenCL. The experimental results present that cheap GPUs are able to lead better performance than much more expensive servers when using OpenCL paradigms.

The organization of this paper is as following. In Section 2, multi- and many-core processor architectures are reviewed. Section 3 describes two parallel programming paradigms including the introduction on DCT, and compares their difference. In section 4, the experimental results will be discussed; finally the conclusion will be addressed in Section 5.

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (KRF 2010-0 0028047).

978-0-7695-4470-0/11 $26.00 © 2011 IEEE
DOI 10.1109/MUE.2011.13

II. RESEARCH BACKGROUND

A. Multi-core CPUs

The first 32-bit microprocessor was introduced based on wide-issue superscalar architecture, shown on Figure 1(a), for deep ILP (Instruction-Level Parallelism) in the mid-1980s. This architecture allowed more than one instructions to be executed during a clock cycle. Another way of increasing on-chip parallelism is SMT (Simultaneous Multi-Threading) technique[5], which permits multiple independent threads of execution to better utilize the resources for TLP (Thread Level Parallelism). SMT enabled processors to issue multiple instructions from multiple threads in one cycle. Figure 1(b) shows the architecture of SMT. Intel's implementation of SMT is known as Hyper-Threading Technology which makes a single processor appear, from software's perspective, as multiple logical processors.

The next architectural trend was the shift toward multi-core processors using chip multiprocessing (CMP). Especially, the advanced manufacturing technology enables a single processor to be implemented two or more execution cores. As a result, in recent years, there has been a steady increase in the number of processing cores available on a processor chip; dual and quad core systems are already becoming popular; furthermore, six and eight cores are being introduced. Their architectures become different according to number of core processors and number of levels of cache on chip, and amount of shared cache. Figure 2 shows the architecture of Intel Core Duo using two

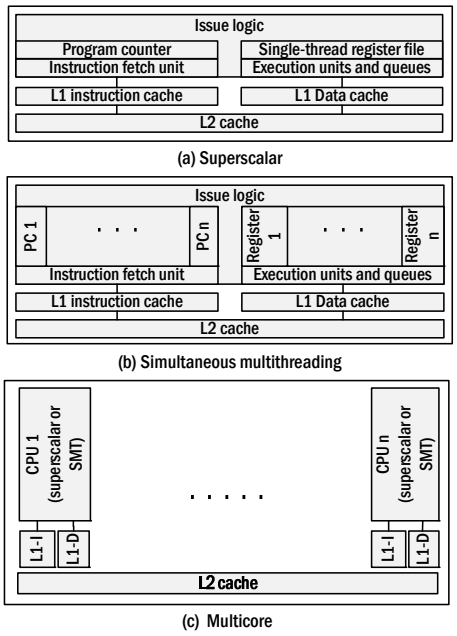

Figure 1. Processor architectures [4]

x86 superscalar cores, shared L2 cache, and dedicated L1 cache per core. Intel Core i7 uses simultaneous multi-threading (SMT) with four x86 SMT processors with dedicated L2 and shared L3 cache.

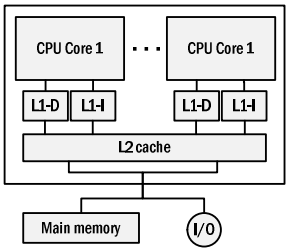

Figure 2. Intel Duo core [4]

To extract high performance from such architectures, Intel provides TBB [6], an open source runtime C++ library that targets desktop shared memory parallel programming. The TBB programming environment allows programmers to express concurrency in terms of parallel tasks rather than parallel threads. Here, tasks are special regions of code that perform a specific action or function when executed by the TBB runtime library. Moreover, TBB is able to significantly reduce load imbalance and improve performance scalability through task stealing, allowing applications to exploit concurrency with little regard to the underlying CMP characteristics (i.e. number of cores). Because it is a library, not a new language or language extension, it integrates into existing programming environments with no change to the compiler. An additional advantage, compared to depending on a language or extension for parallelism is that most programmers can more readily modify a library than a compiler. Hence a library permits more rapid evolution and customization [7].

B. Many-core GPU

In the past few years, GPUs from ATI (AMD) [8] and nVidia [9], have greatly achieved the improvements on computation power and programmability; they have emerged as a high performance parallel computing solution in desktop PCs. Differently from multi-core CPUs, a GPU consists of many simple processing units helping for massively parallel computing; so that a GPU can process many threads simultaneously enabling high computational throughput across large amounts of data, namely SIMT (Single Instruction Multiple Thread) [10].

At the beginning GPUs were a graphic rendering device as an add-on coprocessor board to process graphic related jobs from CPUs with no flexibility for general purpose programming. The architecture of GPUs evolved in 2001, when for the first time programmers were given the opportunity to program individual kernels in the graphics pipeline by using programmable vertex and fragment shaders. However, GPUs were very difficult to use because programmers had to use the specific APIs to access the processor cores, for example, OpenGL or Direct3D. Later GPUs (as the G80 series by nVidia) substituted the vertex and fragment processors with a unified set of generic processors that can be used as either vertex or fragment processors depending on the programming needs. On each new generation, additional features are introduced that move the GPUs one step closer to wider use for general purpose computations. These

improvements in programmability also make them potentially useful for more general computation-centric tasks.

Accordingly, recent GPUs can provide tremendous computational power over CPUs, and the performance gap between them continues to grow larger over time. In 2004, the fastest Pentium4 achieved 7 GFLOPs, but ATi Radeon X850 achieved 66 GFLOPs. In present, the six-core Intel Core i7-980XE achieves over 100 GFLOPs while the GPU AMD Radeon HD5870 achieves 2720 GFLOPs. This computing power can be comparable with low level of supercomputing environments with 2756 GFLOPs, costing 1/500 of supercomputer price. So how to utilize the capabilities of these increased computing powers from modern GPUs will become a major challenge in modern high performance supercomputing applications [11].

Figure 3 shows the architecture of Radeon HD 6970 GPU with two graphics engines. Each graphics engine is composed of many stream processors, depicted as ultra-threaded dispatch processor. They're organized in groups of four, with 16 of these groups per SIMD Engine (or stream processors cluster). Therefore, it has 24 SIMD engines; each engine is configured with 16 stream processors; each stream processor equips with 4 ALUs. That is, this architecture allows the GPU to have 1,536 processing elements enough for massively parallel processing. The Radeon HD 6970 is clocked at 880MHz while the memory runs at 5.5Gbps. Furthermore, the card comes with 2GB memory that will be run through a 256-bit memory interface.

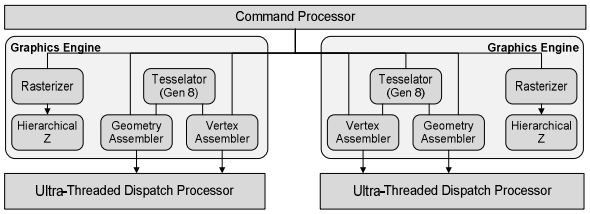

Figure 3. Radeon HD 6970

III. PARALLEL PROGRAMMING OF DCT

In general, one of the simplest ways to speed up processing an algorithm in computer systems is having the capability of simultaneous processing on data and tasks. Current PCs are configured with multi-core CPU and high performance GPU which allow parallel executions in software by providing a platform that supports the simultaneous execution of multiple threads as well as massive data parallelism. This section overviews two parallel programming technologies which enables to exploit task- and data-level parallelism. in image compression. Also, this section will introduce the algorithm, DCT, to be implemented for the performance evaluation.

A. Overview of DCT

The demand for high-speed computing architectures and algorithms for DCT has been increased continuously due to the dominant popularity of digital signal processing and video compression. The primitive operation of DCT is based on matrix computations in which parallel processing techniques must be considered for real time processing with large 2D data sets. This operation is characterized as data intensive tasks accompanied by heavy memory access; on the other hand, their computational complexities are relatively low. Thus, it naturally maps onto massively parallel architecture with distributed memory.

2D DCT can be described as a transform from a 2D matrix of pixels to that of spatial frequency information. The mathematical model and related coefficients used in this paper are as following. For an input matrix $x(m, n)$ and an output matrix $z(k, l)$ with $\{0 \le m, n, k, l \le N\text{-}1\}$, the forward $N \times N$ 2D DCT is defined as

$$z(k,l) = \frac{2}{N}\alpha(k)\alpha(l)\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} x(m,n)\cos\frac{(2m+1)\pi k}{2N}\cos\frac{(2n+1)\pi l}{2N} \quad (1)$$

where $$\alpha(k) = \alpha(l) = \begin{cases} \frac{1}{\sqrt{2}}, & if\ k = l = 0 \\ 1, & otherwise. \end{cases}$$

Equation (1) can be rewritten in matrix form as

$$Z = AXA^T \quad (2)$$

where X is the source pixel (spatial domain) data, Z is the DCT output coefficients (frequency domain), and A is an orthogonal matrix defined as

$$a(u,v) = \sqrt{\frac{2}{N}}\alpha(u)\cos\frac{(2v+1)\pi u}{2N} \quad (3)$$

A naive implementation of (1) requires N^4 multiplications. Alternatively, the 2D DCT can be computed by applying 1D DCT by rows (columns) and then, by columns (rows) due to separable property of 2D DCT. This approach is called the row-column decomposition method and requires $2N$ instances of N-points 1D DCT to implement an $N \times N$ 2D DCT, resulting in $2N^3$ multiplications [12].

In this paper, we have implemented four different serial DCT versions on CPU to establish the basis for comparison. For this purpose, the implementation focuses on DCT computation itself. The first version is the cosine function version, denoted as DCT-1. Here, (1) is implemented directly with serial processing. The second one is the enhanced version of DCT-1 by computing the coefficient in advance and reading them while processing the algorithm. Originally, the cosine function of DCT is very computationally expensive; however DCT-2 can improve the processing speed a lot by pre-computing the coefficients. The next is DCT-3 using direct matrix multiplication based on (2). Usually, matrix multiplication is a good candidate for parallel computing, in which a same program can be executed on many data elements, called data parallel computation. Fortunately, modern GPU has many processing resources as a streaming processor which allows us different data elements on different channel in parallel. Finally, DCT-4 is the implementation of row-column decomposition method. The reduction of computation complexity was already demonstrated analytically from N^4 multiplications to $2N^3$.

B. TBB

In accordance with the increasing popularity of multi-core CPUs, tools enabling easy and quick parallel coding are required with the form of parallel runtime systems and libraries that aim at improving application portability and programming

```
const size_t L = 150;
const size_t M = 225;
const size_t N = 300;

void SerialMatrixMultiply( float c[M][N], float a[M][L], float b[L][N] ) {
       for( size_t i=0; i<M; ++i ) {
            for( size_t j=0; j<N; ++j ) {
                float sum = 0;
                for( size_t k=0; k<L; ++k )
                    sum += a[i][k] * b[k][j];
                c[i][j] = sum
            }}}
```

Figure 4. Serial code block of matrix multiplication

efficiency. TBB [6] is an open source runtime C++ library that targets desktop shared memory parallel programming.

TBB provides programmers with APIs used to exploit parallelism through the use of tasks rather than parallel threads. Moreover, TBB is able to significantly reduce load imbalance and improve performance scalability through task stealing, allowing applications to exploit concurrency with little regard to the underlying CMP characteristics (i.e. number of cores) [7]. Because it is a library, not a new language or language extension, it integrates into existing programming environments with no change to the compiler. An additional advantage, compared to depending on a language or extension for parallelism is that most programmers can more readily modify a library than a compiler. Hence a library permits more rapid evolution and customization [13,14].

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range2d.h"
using namespace tbb;

const size_t L = 150;
const size_t M = 225;
const size_t N = 300;

class MatrixMultiplyBody2D {
    float (*my_a)[L];
    float (*my_b)[N];
    float (*my_c)[N];
public:
    void operator()( const blocked_range2d<size_t>& r ) const {
        float (*a)[L] = my_a;
        float (*b)[N] = my_b;
        float (*c)[N] = my_c;
        for( size_t i=r.rows().begin(); i!=r.rows().end(); ++i ){
            for( size_t j=r.cols().begin(); j!=r.cols().end(); ++j ) {
                float sum = 0;
                for( size_t k=0; k<L; ++k )
                    sum += a[i][k]*b[k][j];
                c[i][j] = sum;
            }
        }
    }

    MatrixMultiplyBody2D( float c[M][N], float a[M][L], float
        b[L][N] ) :
            my_a(a), my_b(b), my_c(c)
        {}
};

void ParallelMatrixMultiply(float c[M][N], float a[M][L], float b[L][N]){
    parallel_for( blocked_range2d<size_t>(0, M, 16, 0, N, 32),
                MatrixMultiplyBody2D(c,a,b) );
}
```

Figure 5. Parallel implementation of matrix multiplication using TBB

The code blocks in Figures 4 and 5 show the usage of TBB with matrix multiplication. The former shows its serial version and the latter the corresponding parallel version which uses blocked-rabge2d to specify the iteration space. Header starts with #include "tbb/blocked_range2d.h" statement. The blocked_range2d enables the two outermost loops of the serial version to become parallel loops. The parallel for recursively splits the blocked_range2d until the pieces are no larger than 16 × 32. It invokes MatrixMultiplyBody2D::operator() on each piece.

C. OpenCL

OpenCL [3] is a recent open standard to supports parallel execution on single or multiple different kinds processors including CPU, GPU, DSP, and other special purpose co-processors. This is a runtime-system, API, and programming language for data- and task-parallel computing model. For this purpose, it abstracts the specifics of underlying hardware and provides programmers with low level control capabilities with OpenCL APIs and kernel language.

OpenCL software stacks are:

- Platform layer allows hardware abstraction by querying and selecting compute devices in the system, initializing a compute device(s), and create compute contexts and work-queues.
- Runtime layer is in charge of executing compute kernels and managing resources of the selected devices.
- Compiler is a subset of ISO C99 with appropriate language additions; it compiles and build compute program executables.

In order to initiate a program using OpenCL, the program

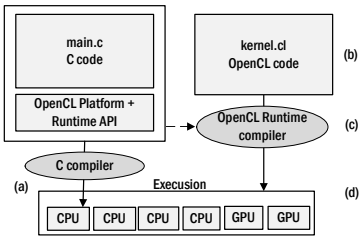

Figure 6. OpenCL execution [26]

have include OpenCL header files to gain access to the platform and runtime API. OpenCL kernels are compiled at run time. Figure 6 shows the sequence of an application under execution to build and initiate an OpenCL kernel. The labels of the sequences, (a) through (d), in the figure are described as follows:

(a) Execution of main.c program. OpenCL header files are included so OpenCL platform- and runtime-calls can be made.

(b) Pure OpenCL source-code loaded from file into memory by the main.c program under execution.

(c) The OpenCL source code is built into an executable for target devices(s) attached to the OpenCL context, and stored in a memory object.

(d) Input and output data locations (pointers), and corresponding types, are set up right before kernel execution - making sure the kernel running on the device(s) gets its data and knows where to store results. Then, the memory object containing correct executable(s), according to OpenCL context, is handed over to the OpenCL runtime and thereby executed on device(s).

In most implementations the OpenCL source code is first compiled into an intermediate representation which is device independent. This intermediate code is optimized as much as possible, before the final code for the selected device is generated by the device's code generator (as part of the device's OpenCL driver/runtime infrastructure) [15].

TABLE I. EXECUTION TIMES

Options		256	512	1024	2048
1	DCT-1	0.15462136	0.61671960	2.46247641	9.78384881
	DCT-2	0.01110100	0.04423459	0.17555091	0.69865293
	DCT-3	0.00236362	0.00913958	0.03690136	0.14571443
	DCT-4	0.00184406	0.00717728	0.03043594	0.11938342
2	TBB-1	0.042652245	0.172499383	0.641383637	2.532382177
	TBB-2	0.003071616	0.011547762	0.044960016	0.183345607
	TBB-3	0.000645335	0.001996582	0.008383777	0.034291673
	TBB-4	0.000486851	0.002124348	0.008041399	0.030636395
3	DCT-2	0.01110100	0.04423459	0.17555091	0.69865293
	TBB-2	0.003071616	0.011547762	0.044960016	0.183345607
	OCL-2	0.000755398	0.001225752	0.002958031	0.010160484

IV. SIMULATIONS

For the simulation, we estimate the computation speed of different DCT implementations, DCT-1, 2, 3, and 4, as mentioned in Section 3, on different image sizes, 256 × 256, 512 × 512, 1024 × 1024, and 2048 × 2048. They are denoted as 256, 512, 1024, and 2048, respectively. All versions are coded with C++ with 8 × 8 DCT coefficient and compiled by Microsoft Visual Studio 2010 running on Windows 7. And then they are implemented on TBB and OpenCL to exploit multi- and many-core parallelism, respectively. In porting them to the TBB environment, we use version 3.0 of Intel TBB library [16] for Windows and they are denoted as TBB-1, 2, 3, and 4, respectively. In case of OpenCL 1.1 [2], currently only DCT-2 is implemented and denoted as OCL-2. We use ATI stream SDK v2.0.1 with OpenCL support. We perform experiments on the Radeon HD 6970 GPU at 850MHz. The counterpart CPU is Intel Core 2 Quad Q9550 running at 3.41GHz with overclocking; main memory is 4GB DDR3 RAM. To obtain the execution time in terms of CPU clock cycles, we use Pentium's RDTSC (Read Time Stamp Counter) [17] instruction to measure the execution time of the different implementation versions. Execution times were obtained by executing each implementation 400 times inside a loop, and then summed averaged execution time was selected.

The experiment result is summarized in following tables. Table 1 shows the various implementations of DCT categorized into three groups. Group 1 consists of serial implementations on CPU. Here, DCT-4, the row-column decomposition method reveals the fastest result with over 80 times speedup. Figure 7 shows the result of them excluding the slowest DCT-1. Group 2 shows the execution times of TBB implementations; the result shows that TBB can achieve average 3.8 speedup over serial implementations, shown in Figure 8. As for TBB implementation, particularly, the TBB version of DCT-3, the direct matrix multiplication method, shows the greatest performance improvement. Finally, Group 3 shows the performance comparisons between three different technologies, CPU serial, CPU with TBB, and GPU with OpenCL for the implementation of DCT-2, shown in Figure 9. The implementation on OpenCL outperforms the one on TBB; it can achieve over 70 times speedup over the serial implementation while TBB having 4 times speedup, shown in Figure 10. Especially, it shows a linear speedup, a typical characteristic of massively parallel processing, as the increase of 2D data sets.

V. CONCLUSION

Nowadays modern PCs are equipped with multi-core CPU and many-core GPU, which can provide a great opportunity to increase the processing performance of an application by exploiting task- and data-parallelism in heterogeneous system. This paper presented an optimization of 2D DCT on speed in multi-core CPUs and many-core GPUs. For this purpose two parallel programming techniques of Intel TBB (threading building blocks) and OpenCL, were discussed in detail. The simulation result reveals that the parallel DCT implementations are performed considerably faster than the serial ones. Especially, OpenCL implementation on GPU shows a linear speedup, a typical characteristic of massively parallel processing, as the increase of 2D data sets. It means that GPUs will help us to have supercomputing power with very low cost on our desktop PCs.

REFERENCES

[1] K.A. Hawick, A. Leist, and D.P. Playne, "Mixing Multi-Core CPUs and GPUs for Scientific Simulation Software,", Computer Science, Massey University, Tech. Rep. CSTN-102, 2009.

[2] The OpenCL Specification 1.0 rev.48, Khronos OpenCL Working Group, Oct. 2009. Available at http://www.khronos.org/opencl/

[3] J.E. Stone, D. Gohara, and S, Guochun, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems, Computing in Science & Engineering, vol. 12, no. 3, May-June 2010 pp. 66–73.

[4] W. Stallings, Computer Organization and Architecture 8/E: Designing for Performance, Prentice Hall, 2009.

[5] D.M. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," In Proceedings. 22nd Annual Int'l Symposium on Computer Architecture, 1995. ISCA-22, 1995. pp. 392-403.

[6] J. Reinders, Intel Threading Building Blocks, O'Reilly, Sebastopol CA, 2007.

[7] G. Contreras and M. Martonosi, "Characterizing and Improving the Performance of Intel Threading Building Blocks," In Proceedings. IEEE Int'l Symposium on Workload Characterization), Sep. 2008, pp. 1-10.

[8] AMD corp. AMD Graphics for Desktop PCs. Available at http://www.amd.com/us/

[9] NVIDIA corp. NVIDIA GeForce Family. Available at http://www.nvidia.com/object/geforce_family.html

[10] W. Zhu and J. Curry, "Parallel ant colony for nonlinear function optimization with graphics hardware acceleration," IEEE Int'l Conference on Systems, Man and Cybernetics, 2009, pp. 1803-1808.

[11] S.L. Chu, and C.C. Hsiao, "OpenCL: Make Ubiquitous Supercomputing Possible," 12th IEEE Int'l Conference on High Performance Computing and Communications (HPCC), 2010, pp. 556-561.

[12] C.G. Kim, S.J. Lee, and S.D. Kim, "2-D Discrete Cosine Transform (DCT) on Meshes with Hierarchical Control Modes," Lecture Notes in Computer Science, 3522, June 2005, pp. 675-682.

[13] A. Robison, M. Voss, and A. Kukanov, "Optimization via Reflection on Work Stealing in TBB," IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008, pp. 1-8.

[14] S. Akhter and J. Roberts, Multi-Core Programming: Increasing Performance through Software Multi-threading, Intel Press, 2006.

[15] A. Fagerlund, Multi-core programming with OpenCL: performance and portability- OpenCL in a memory bound scenario, Master thesis, Norwegian University of Science and Technology, 2010. Available at http://daim.idi.ntnu.no/

[16] http://threadingbuildingblocks.org/

[17] http://developer.intel.com/design/xeon/applnots/241618.htm