Survey on Discrete Wavelet Transforms Using Parallel Architectures

Samuel Li

April 2015

1 Introduction

Wavelet Transforms originated from signal processing community as a new tool method to perform signal analysis. Researchers soon expanded the use of wavelet transforms to digital signals, resulting in the calculation of discrete wavelet transform (DWT). While DWTs are still used in digital signal analysis, one of the most prominent uses has been data compression, as well as providing a *progressive data access* to data analysis tasks. With progressive data access, a data analysis process starts with a coarser version of the data, which takes only a fraction of the raw data, and then decides the region of interest to continue investigation, during which more detailed data is loaded for that particular region. As examples, the JPEG 2000 image standard makes use of wavelet transforms to achieve data compression [5], and the VAPoR software package uses wavelet transforms to provide progressive data access [7].

The calculation of DWT is a computation intensive task. Researchers have been always adopting the latest developments in parallel computing to accelerate this calculation. Some of the architectures that researchers have ported DWT to include single instruction multiple data, shared memory architecture, and distributed memory systems. In the recent years, the general-purpose graphics processing units (GPGPU) have emerged as a new powerful tool to achieve high performance. In this survey paper, I am going to briefly introduce some successful implementations of DWT on above parallel architectures. I especially pay attention on how inter-processor communication is handled in these implementations, since data movement consumes a great portion of total execution time in parallel computing settings. GPGPU is a little special among all these

architectures, because it has many more processing units than other architectures (hundreds to thousands). Data partitioning then requires more attention than the other architectures, which we will focus to discuss in the survey.

This survey is organized as following: Section 2 talks about background knowledge of wavelet transforms. Section 3 to 7 discuss implementation concerns for architectures single instruction multiple data machine, shared memory architecture, distributed memory systems, GPGPU with CUDA, and GPGPU with OpenCL respectively. Section 8 concludes this survey paper.

2 Background of Wavelet Transforms

Discrete Wavelet Transform (DWT) on a signal $x[n]$ $(0 \leq n < N)$ is essentially passing the signal through a filter function, f, in a convolution operator. This filter function is also called the *kernel* of this DWT. The results of this DWT are *coefficients* of the signal. The coefficients can be further categorized as two separate groups: one representing an approximation of the original signal, namely *scale coefficients*; and another one contains the detailed information to reconstruct the original signal from the approximation, namely *detail coefficients*. Normally each of these two groups of coefficients has a size $\frac{N}{2}$.

DWTs can also be applied on the coefficients from previous DWTs. When applying another round of DWT, transforms are applied on the two groups of coefficients separately. As a result, the original signal $x[n]$ is represented as 4 groups of coefficients; each has a size $\frac{N}{4}$. This iteration finishes when each of the coefficient groups are small enough, or the number of iterations reaches the maximum value set by the user. Figure 1 illustrates an example of applying DWTs for three iterations on an 8-element signal.

From the perspective of programming models in parallel computing, the calculation of DWT is like applying a stencil pattern — each output coefficient is calculated from a few input elements. The kernel function of the DWT is the applied stencil in parallel programming.

We can also perform DWTs on multi-dimensional data. The calculation of multi-dimensional DWT is based on one-dimensional DWT: we essentially perform many one-dimensional DWTs on each of the dimensions of a multi-dimensional data. For example, the calculation of DWT on a two-dimensional image is basically performing 1D DWTs on each row of the image, following by 1D DWTs on each column of this image.

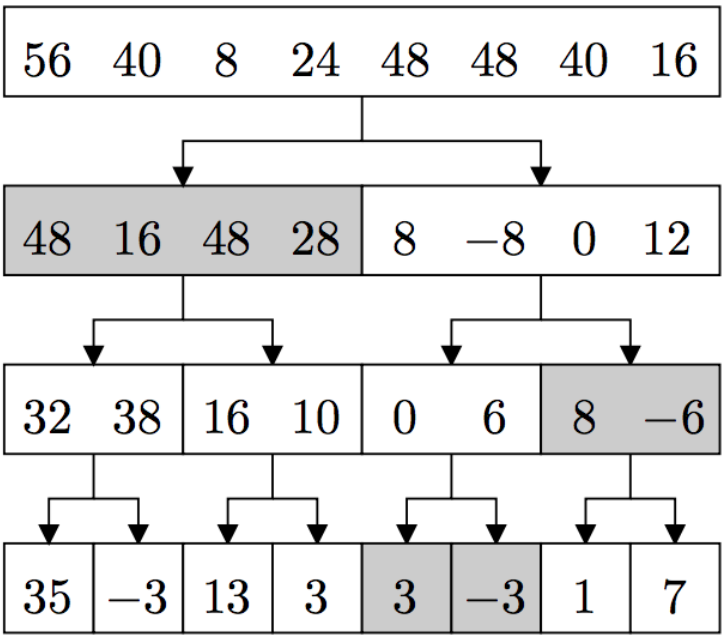


Figure 1: An example of DWT on a signal with 8 elements.

3 Single Instruction Multiple Data

The Single Instruction Multiple Data (SIMD) architecture was popular in the 1990s. With SIMD architecture, all processing elements (PE) execute identical instructions cycle by cycle, but they access different local data from their private local memory. All PEs are also connected in a net structure, allowing communication between them. Figure 2 illustrates an example SIMD architecture.

Lee et. al. implemented DWT on a SIMD machine, the MasPar system [1]. MasPar has 2048 PEs organized in a 64x32 mesh topology, as shown in Figure 2. This implementation takes use of the row structures of available PEs, by mapping an one-dimensional input signal onto a row of PEs to process. More specifically, a row of PEs take input values and performe calculations together, and passes the output coefficients to the next row of PEs. The next row of PEs then performs the next iteration of DWT on the input coefficients. At the same time, the first row of PEs takes in new input to calculate on. This process is illustrated in Figure 3.

From the perspective of parallel execution, this implementation uses a pipeline pattern to achieve great parallelism. As we discussed in class, the amount of

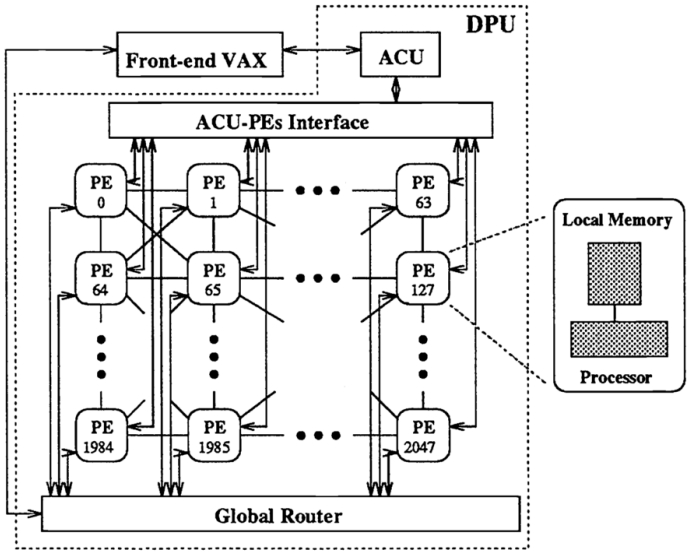


Figure 2: An example of the SIMD architecture: the MasPar system.

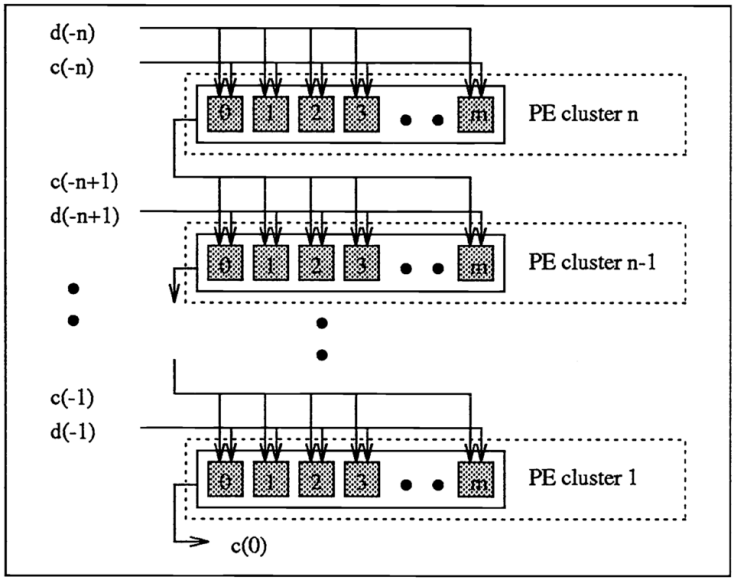


Figure 3: The pipeline of rows of PEs executing same instructions. The output of previous row serves as input for the next row of PEs to perform the next iteration of DWT.

parallelism achieved depends on the number of iterations of DWT we could perform on the input signal. This implementation might not be able to make use of all PEs if run on a very large system with many rows of PEs.

The experiment shows that the parallel implementation has a better performance than that of a serial implementation on the SPARC 1 workstation, whose processor is at least 100 times faster than that of the MasPar. Scalability wise, this pipeline implementation is able to keep good speedups when the problem size grows. This is because larger problems naturally enable more iterations of DWT to apply on. As a result, the execution pipeline has more steps, and can run more efficiently.

4 Shared Memory Architecture

When performing DWT, the calculation of each coefficients depends only on a few input elements from its neighbors (see stencil pattern discussion in Section 2). This property makes the calculation of DWT fit in the Shared Memory Architecture (SMA) very well, by assigning each PE a small chunk of the input signal array to operate on. Research works from Kutil et. al. and Lucka et. al. both employ this parallel scheme [3, 4].

Experiments by Kutil et. al. on the SGI POWERChallengeGR machine showed a linear speedup up to 10 PEs, while the parallel efficiency drops dramatically with more PEs. The researchers than decided that the efficiency was hurt by cache misses when more PEs were involved in. These observations and analyses are confirmed by Lucka et. al. Their experiments and analyses showed similar speedup patterns and performance impact by cache misses.

5 Distributed Memory Systems

New challenges arise when performing DWT on distributed memory systems, since job decomposition and inter-processor communication are both controlled by the program. Load balance and communication costs then become the biggest concerns regarding DWT on distributed memory systems.

Chadha et. al. analyzed two straightforward strategies of decomposing the DWT on two-dimensional data array: stripe decomposition and checkerboard decomposition [6]. These two decomposition strategies require different ways to communicate with neighbor processors. On the one hand, stripe decompo-

sition requires communicating to two neighbor processors, while checkerboard decomposition requires four. On the other hand, stripe decomposition requires communicating larger amount of data, compared to checkerboard decomposition. Chadha et. al. showed that the stripe decomposition provides better performance than the checkerboard partitioning. More specifically, the theoretical speedup for stripe decomposition is nearly linear with larger number of processors.

Nielsen et. al. independently revealed linear speedup when performing two-dimensional DWT, using the stripe decomposition [2]. Moreover, this research analyzed two approaches of data communication in two-dimensional DWT. The first approach performs a matrix transpose in between of DWTs on two the dimensions. This approach keeps good data locality for each single calculation, but results in large amount of data transferring when switching the calculation to the other dimension. The second approach keeps data local on each processor, and exchange values with its neighbor processors. This approach has minimal amount data to transfer, but the data transfer happens more frequently. Figure 4 illustrates these two approaches. Analysis showed the second approach, which saves a massive transpose operation, performs better with larger number of processors.

6 CUDA architecture

Traditionally, GPUs are used for fast processing on graphics-related calculations. A GPU usually consists of hundreds or thousands of processing elements, though each is simply focusing on floating point operations. As a result, a GPU has very high FLOPS to carry out graphics-related calculations, but lack the ability to perform sophisticated operations like a CPU.

More recently, the GPU venders have developed APIs that allow users to program general-purposed programs on the GPUs. As a result, the GPUs have become an important architecture in parallel computing community. The newest supercomputer ranking, top500, reports that three out of top ten supercomputer systems are equippted with GPUs.

Despite the same vision of using GPUs to accelerate the computation, different venders of GPUs provide different APIs. The most successful one, Compute Unified Device Architecture (CUDA), is currently from NVIDIA. Our focus will be specifically on the CUDA architecture in the rest of this section.

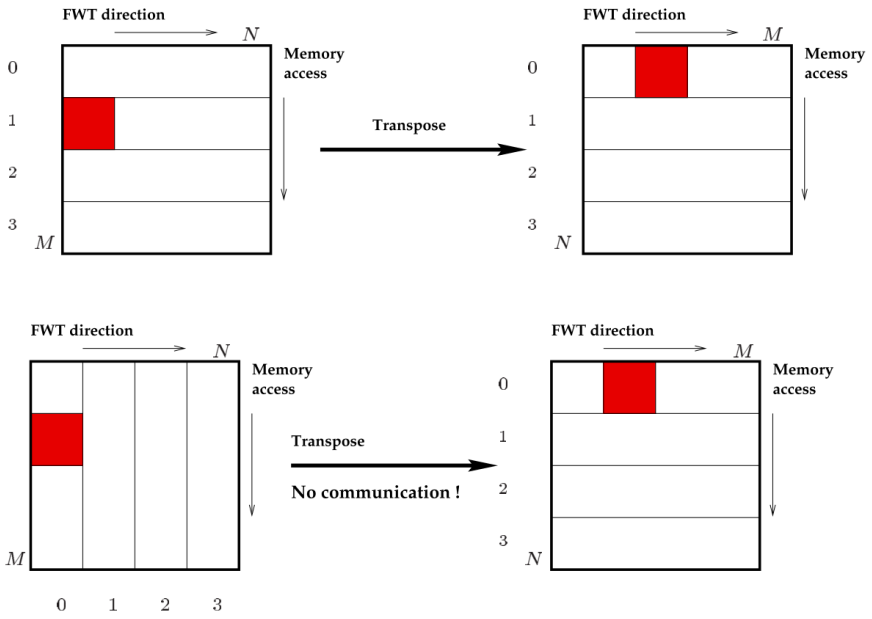


Figure 4: Two approaches of data communication: the top approach involves a transpose for the entire data, while the bottom approach involves boundary communication between neighbor processors.

6.1 Hardware Organization and Programming Model

With hundreds to thousands of cores and the associated memory, a CUDA GPU is organized in its unique way. Franco et. al. provides a good overview of such architecture [8].

CUDA cores are organized in two hierarchies in a CUDA GPU: first a certain number of CUDA cores are grouped together as a multiprocessor, and second a set of multiprocessor are put together on a CUDA GPU. Here each multiprocessor has a SIMD architecture.

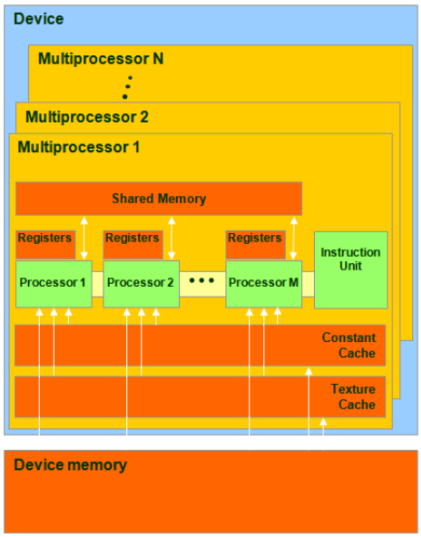
The memory organization on a CUDA GPU is also in hierarchy. In the most fine-grained level, there are registers attached to each CUDA core. The next level memory is *shared memory*, which is shared by all cores in a multiprocessor. *Device memory* is on the GPU card, and is shared by all the CUDA cores. In addition to shared memories and device memories, there are also dedicated memories for particular uses, namely constant cache and texture cache. We are not going to cover these dedicated memories in this survey. At last, the hardware model of a CUDA GPU is shown in Figure 5a.

In the CUDA programming model, a multi-threaded task is partitioned into many *blocks*. Threads in the same group are executing the same instructions. From the perspective of the program, there could be as many blocks as necessary. Eventually, each thread is identified using its *threadID* in the block, and the *blockID* of that particular block. When executing, each block of threads is mapped to run on one CUDA multiprocessor. This mapping is managed by CUDA to be scalable, meaning that each CUDA multiprocessor is always executing a thread block, until the task is finished [13]. This CUDA programming model is illustrated in Figure 5b.

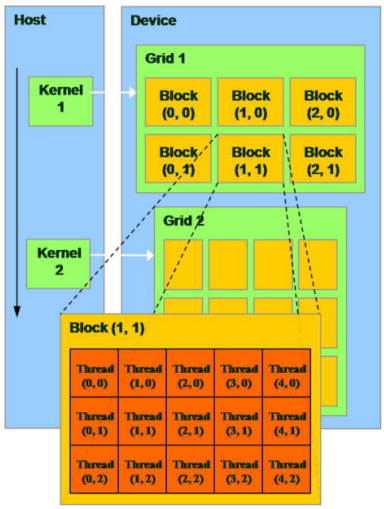
6.2 Naive Parallelism using CUDA

Franco et. al. [8] parallelized two-dimensional DWT following the similar approach illustrated in the top diagram of Figure 4. More specifically, the researchers identified two sections of DWT that could be parallelized: the one-dimensional DWT on rows, and the matrix transpose. When implementing such algorithms using CUDA, the program specifies sections to run on the GPU as *kernels*, and pass the data as well as operations onto the GPU to execute. The calculation results from GPU executions are then copied back to continue this algorithm for further operations.

Experiment results show that the CUDA implementation of two-dimensional



(a) Hardware model of a CUDA GPU. CUDA cores, multiprocessor, and the GPU card make a hierarchy together.



(b) Programming model on a CUDA GPU. Threads, threadsblocks, and grids make a programming hierarchy.

Figure 5: CUDA hardware and programming model.

DWT on an NVIDIA Tesla C870 GPU achieves 10x to 21.7x speedups, compared to the optimized implementation on an Intel Core 2 Quad CPU. A breakdown of the total calculation time further shows that the two data transfer steps, which move data from system main memory to the GPU to perform DWT and matrix transpose, take around half of the total elapsed time. This result indicates that reducing data movement between the GPU and system main memory is a direction to further increase performance.

6.3 To Make Better Use of Blocks in CUDA

Lann et. al. investigated better use of the blocks in CUDA, and proposed a different way to partition the input data to fit in this model [10]. This section is going to briefly introduce two different approaches the researchers use to perform DWT on rows and columns.

When an algorithm designer makes decisions to move multi-threaded tasks to the GPU, one always needs to partition the problem carefully, with the consideration of memory operations. Specifically, the *coalesced* reads and writes, meaning that reads and writes of a continuous chunk of memory, are always

preferable. In a two-dimensional data set that is stored in a row-oriented fashion, accessing data row by row naturally implies coalesced reads and writes, but column by column results in discrete random data access. As a result, Lann et. al. designed different data partitioning schemes for DWT on rows and on columns.

In the first case of DWT on rows, the researchers partitioned the two-dimensional data into many 1D arrays, each representing one row from the original data set. Each array is then mapped onto one block in the CUDA programming model. Figure 6a illustrates this approach: each thread block performs DWT on one row of input data. The black squares indicate that one thread moves along the row as the block calculates DWT in the one-dimension fashion.

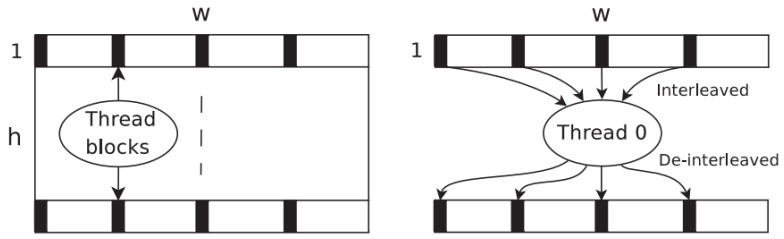
In the second case of DWT on columns, a simple partition of every column would not work well, because of the discrete memory access. The researchers then decided to partition the data in "slabs," rather than single columns, to map to the programming blocks. Figure 6b compares this partitioning with the partitioning for DWT on rows.

Experiments showed that DWT on columns using this "slab"-like partitioning performs only 10% to 20% slower than DWT on rows. The researchers also compared this approach to the one in Section 6.2, which involves a matrix transpose and then performs DWT on columns just in consecutive memories. The results are still in favor of their approach — the total execution time is 2x to 2.5x faster than the approach that involves matrix transpose.

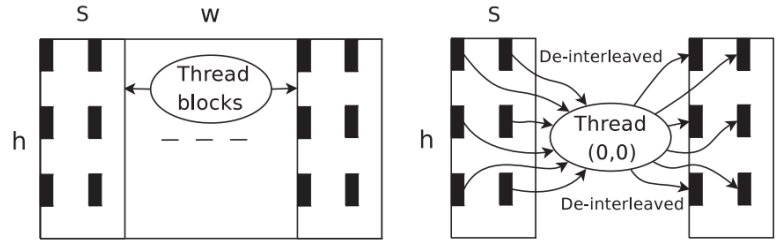
6.4 Optimization for DWT on GPUs

Bernabe et. al. proposed an automatic technique to optimize the DWT on GPUs [12]. The goal of the optimization is to maximize the occupancy of each CUDA multiprocessor. The optimization algorithm takes input from the physical specifications of the GPU card, the CUDA occupancy calculator, the routine to execute on the GPU, and a few heuristic rules. The key parameter to optimize is the block size, and then further the grid size based on optimized block sizes.

The researchers tested this automatic optimization technique on three test data sets, and three GPU cards. Test results showed that the occupancy of multiprocessor do vary on different GPU cards, and on different test data sets. The optimization technique is able to choose the best block size for each problem.



(a) Data partition of the two-dimensional data set onto blocks in a GPU programming model to perform DWTs on rows. Each row is mapped to a block in this figure, and the black square shows a thread process data along this row.



(b) Data partition of the two-dimensional data onto blocks in a GPU programming model to perform DWTs on columns. Each block contains S columns together in this figure.

Figure 6: Different data partitioning schemes for DWT on rows and columns.

An interesting observation is that though occupancy of a same GPU testing on different data set varies, the optimal block size seems to be the same. For example, both Tesla C870 and Fermi C2050 GPUs have 192 as the optimal block size, while the other tested card has 256 as optimal block size. It should be interesting to see more test results and decide if a single block size value can fit all problems on one specific GPU card.

7 Open Programming Language

In addition to the CUDA framework on NVIDIA GPUs, the Open Computing Language (OpenCL) provides another standard for programming on GPUs. OpenCL is supported by more venders, including Apple, AMD, NVIDIA, Intel, IBM, etc., because it is an open standard. The main advantage of OpenCL is its portability across multiple devices and capability for heterogeneous computing. In the following subsections, I will briefly introduce the programming model of OpenCL, then provide a use case of performing DWT on OpenCL, and finally compare OpenCL with CUDA.

7.1 OpenCL Programming Model

Sharma et. al. provide an overview of the OpenCL programming model [9]. This programming model shares the same structure as the CUDA model. In the finest level, each thread is named as a "work-item". In the coarser level, many work-items are put together as a work-group. The work-groups here are similar to the blocks in a CUDA model.

In terms of the memory model, the global memory is shared between all work-groups, while the local memory is shared by work-items in a particular work-group. Each work-item, then, has its private memory, which are essentially registers. Figure 7 illustrates this memory model.

7.2 Comparison Between OpenCL and CUDA

As an open standard, OpenCL is supported by both NVIDIA and ATI. Sharma et. al. [9] and Bernabe et. al. [11] studied the performance comparison between OpenCL and CUDA

In the performance comparison study, the researchers implemented DWT using the same algorithm, in both OpenCL and CUDA framework. The test

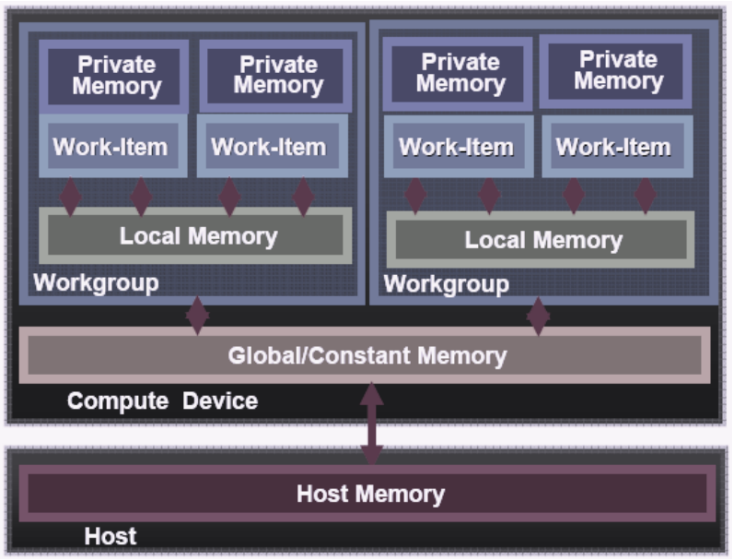


Figure 7: The OpenCL memory model

was run on an NVIDIA card, since that is the only GPU vender that supports both frameworks. The performance of memory access and kernel calculation is evaluated separately.

The first set of experiments compare different memory access patterns. The overall results suggest that OpenCL memory accesses are slower than CUDA, with great variation from some API calls to others. More specifically, the OpenCL performs as bad as 6x slower than CUDA using some API calls, while exhibits same results in some other API calls.

The second set of experiments compare the kernel codes that actually perform the calculation. Similar to the memory access tests, OpenCL implementations perform poorer than CUDA implementations. But interestingly, the slow down of OpenCL has much smaller variance — it is up to 72% slower than CUDA.

While all the test results are strongly in favor of CUDA rather than OpenCL, I notice that all these tests are run on NVIDIA GPUs. Given the test results that some OpenCL APIs perform better (e.g. close to CUDA) than some others, it is quite possible that NVIDIA puts more effort optimizing those good-performing API calls. More experiments and studies are needed to draw a more thorough conclusion on the comparison between CUDA and OpenCL.

8 Conclusions

We surveyed discrete wavelet implementation on parallel architectures. The parallel architectures cover a wide time range from 1990s to 2010s, including single instruction multiple data machine, shared memory architecture, distributed memory systems, GPGPU with CUDA, and GPGPU with OpenCL. We focused on how data movement is optimized to fit the parallel architectures, as well as how data partition affects the performance in the GPGPU architectures.

References

[1] Hung-ju Lee, Jyh-Charn S Liu, Andrew K Chan, and Charles K Chui. Parallel implementation of wavelet decomposition/reconstruction algorithms. In *SPIE's International Symposium on Optical Engineering and Photonics in Aerospace Sensing*, pages 248–259. International Society for Optics and Photonics, 1994.

[2] Ole Møller Nielsen and Markus Hegland. A scalable parallel 2d wavelet transform algorithm. Technical report, TR-CS-97-21, The Australian National University, 1997.

[3] Rade Kutil and Andreas Uhl. Hardware and software aspects for 3-d wavelet decomposition on shared memory mimd computers. In *Parallel Computation*, pages 347–356. Springer, 1999.

[4] Maria Lucka and T Sorevik. Parallel wavelet-based compression of two-dimensional data. *Proceedings of Algorithmy*, pages 1–10, 2000.

[5] Athanassios Skodras, Charilaos Christopoulos, and Touradj Ebrahimi. The JPEG 2000 still image compression standard. *Signal Processing Magazine, IEEE*, 18(5):36–58, 2001.

[6] NI Chadha, A Cuhadar, and Howard Card. Scalable parallel wavelet transforms for image processing. In *Electrical and Computer Engineering, 2002. IEEE CCECE 2002. Canadian Conference on*, volume 2, pages 851–856. IEEE, 2002.

[7] John Clyne. The multiresolution toolkit: Progressive access for regular gridded data. In *Proceedings of Visualization, Imaging, and Image Processing 2003*, pages 152–157, 2003.

[8] Joaquín Franco, Gregorio Bernabé, Juan Fernández, and Manuel E Acacio. A parallel implementation of the 2d wavelet transform using cuda. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 111–118. IEEE, 2009.

[9] Bharatkumar Sharma and Naga Vydyanathan. Parallel discrete wavelet transform using the open computing language: a performance and portability study. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[10] Wladimir J Van der Laan, Andrei C Jalba, and Jos BTM Roerdink. Accelerating wavelet lifting on graphics hardware using cuda. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):132–146, 2011.

[11] Gregorio Bernabé, Ginés D Guerrero, and Juan Fernández. Cuda and opencl implementations of 3d fast wavelet transform. In *Circuits and Systems (LASCAS), 2012 IEEE Third Latin American Symposium on*, pages 1–4. IEEE, 2012.

[12] Gregorio Bernabé, Javier Cuenca, and Domingo Giménez. Optimization techniques for 3d-fwt on systems with manycore gpus and multicore cpus. *Procedia Computer Science*, 18:319–328, 2013.

[13] Corporation Nvidia. Programming guide: Cuda toolkit documentation, 2015.