An Application-Oriented Approach for Accelerating Data-Parallel Computation with Graphics Processing Unit

S. Ponce J. Huang S. I. Park C. Khoury Y. Cao F. Quek W. Feng

Department of Computer Science, Virginia Tech

Abstract

This paper presents a novel parallelization and quantitative characterization of various optimization strategies for data-parallel computation on a graphics processing unit (GPU) using NVIDIA's new GPU programming framework, Compute Unified Device Architecture (CUDA). CUDA is an easy-to-use development framework that has drawn the attention of many different application areas looking for dramatic speed-ups in their code. However, the performance tradeoffs in CUDA are not yet fully understood, especially for data-parallel applications. Consequently, we study two fundamental mathematical operations that are common in many data-parallel applications: convolution and accumulation. Specifically, we profile and optimize the performance of these operations on a 128-core NVIDIA GPU. We then characterize the impact of these operations on a video-based motion-tracking algorithm called vector coherence mapping, which consists of a series of convolutions and dynamically weighted accumulations, and present a comparison of different implementations and their respective performance profiles.

1 *Introduction*

With the volume of scientific data doubling every year [5], application scientists require high-end computing operate on this data. In many cases, the data can be broken into independent pieces so that it may be processed in a data-parallel manner, e.g., a single compute node with multiple processors (or accelerators) or a cluster supercomputer with tens of thousands of processors [2].

Here we focus on a data-parallel computing unit known as the GPGPU, short for general-purpose graphics processing unit, where each GPGPU consists of as many as 128 highly parallel streaming cores. The GPGPU has evolved into an extremely powerful computational resource, as exemplified by the NVIDIA GeForce 8800 GTX, which can deliver over 300 GFLOPS in performance while a 3.0-GHz Intel Core2 Duo delivers a "mere" 50 GFLOPS [3, 13].

In light of the emergence of the GPGPU as an alternative platform for accelerating data parallel or computationally intensive tasks, graphics card vendors have now coupled their programmable GPUs with high-level languages and environments such as Brook+ (an AMD-enhanced implementation of Brook, the popular open-source C-level language and com-

piler) and CUDA (short for Compute Unified Device Architecture by NVIDIA [1]). Such higher-level languages have enabled the GPU to emerge as a more general-purpose, massively parallel, cost-effective solution over its previous-generation stream-computing counterpart.

For the purposes of this paper, we present a novel parallelization and quantitative characterization of various optimization strategies for data-parallel computation on a GPU using NVIDIA's CUDA [1]. The framework has a C-language based programing environment and its GPU architecture supports a controllable memory hierarchy, which makes massively parallel GPU programming much easier than previous shader-based GPU programming. We choose two fundamental data-parallel operations, convolution and accumulation, to explore performance optimization strategies using CUDA-supported GPUs.

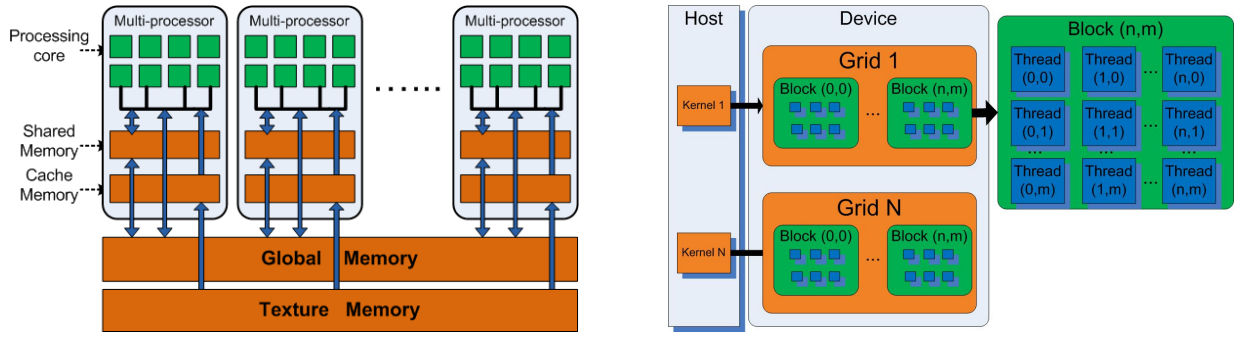
We first present two general optimization strategies: avoiding instructional branching and hiding high memory-access latency. Then, we quantitatively analyze how efficiently we can apply these two optimization strategies to the data-parallel operations of convolution and accumulation. We present an algorithmic performance evaluation with respect to different parallel implementations and algorithm parameters in order to demonstrate the efficacy of our proposed acceleration strategies for the two operations. We find that our branch-avoidance implementation improves the performance of convolution by 2.68 times and accumulation by 1.6 time. Our memory-latency hiding implementation achieves a performance gain of 17.4 times for convolution.

Using the proposed GPU implementations for convolution and accumulation, we then develop a CUDA-based parallel program for a motion vector tracking system. This video-based tracking system, called Vector Coherence Mapping (VCM), is a canonical data-parallel application built by the operations of convolution and accumulation. Our performance evaluation results show that the GPU implementation exhibits a performance gain of more than 22-fold over a state-of-art CPU implementation of VCM.

In summary, the paper makes the following three contributions.

- We rigorously characterize the effects of various CUDA-based GPU programming optimization strategies for data-parallel computation using two fundamental mathematical operations, convolution and accumulation, as examples.
- By exploring the implementation space of different algorithm parameters, we provide optimized GPU implementations for convolution and accumulation.
- We apply the GPU-based optimization strategies to a video processing application for massively parallel motion-vector tracking.

The reminder of the paper is organized as follows. In Section2, we describe the recent advances in GPU hardware and programming framework, discuss previous effort on application acceleration using CUDA framework, and the use of



(a) Geforce 8 series GPU architecture (b) Thread and block structure of CUDA

Figure 1: CUDA-supported GPU architecture and computation organization in CUDA.

GPUs to data parallel algorithms in computer vision applications. In Section3, we introduce two data parallel algorithms, convolution and accumulation, which are two of the fundamental mathematical operations in computer vision. We then discuss two general performance optimization strategies in Section4. In the heart of the paper, Section5, we provide a detailed analysis of different implementations for convolution and accumulation based on the proposed two general optimization strategies. Given such rigorous characterization of CUDA-based optimization, we also propose a standard GPU implementation of convolution and accumulation. In Section6, we use a motion vector tracking system as our example application to demonstrate our proposed GPU optimization strategies. We conclude this paper in Section7.

2 *Background*

2.1 *The NVIDIA CUDA Programming Framework*

Traditionally, general-purpose GPU programming was accomplished by using a shader-based framework. The shader-based framework has several disadvantages. This framework has a steep learning curve that requires in-depth knowledge of graphics programming. Algorithms have to be mapped into vertex transformations or pixel illuminations. Data has to be casted into texture maps and operated on like it is texture data. And because shader-based programming was originally intended for graphics processing, there is little control over data flow. Unlike a CPU program, a shader-based program cannot have random memory access for writing data. In addition, there are limitations on the number of branches and loops a program can have. All of these limitations hindered the use of the GPU for general-purpose computing.

NVIDIA recently released a new generation of graphics cards, the GeForce 8 series, that dramatically changed the general purpose GPU programming. As illustrated in Figure1 (a), the Geforce 8 GPU architecture is a collection of multiprocessors, each of which has 8 SIMD (Single Instruction, Multiple Data) processing cores. The SIMD processor architecture allows each processor in a multiprocessor to run the same instruction on different data, making it ideal for data-parallel computing. Each multiprocessor also has 16K of shared memory and 8K of cache memory that is shared between its processing cores. Shared memory allows for read and write access with little latency. There is also global memory and texture memory that is shared by all the multiprocessors.

NVIDIA released CUDA to assist developers in general-purpose computing. The purpose of CUDA is to provide a general programming framework for GPU programming. The syntax is intuitive for users who are familiar with the C language. In order to organize a high volume of threads running in parallel on the GPU, CUDA organizes threads into logical blocks. Each block is mapped onto a multiprocessor in the GPU. All the threads in one block can be synchronized together and communicate with each other. Because there is a limited number of threads that a block can contain, these blocks are then organized into grids allowing for a larger number of threads to run concurrently. Threads in different blocks cannot be synchronized, nor can they communicate even if they are in the same grid. All the threads in the same grid run the same GPU code. This block organization allows the algorithm to scale with future generations of the GPU.

CUDA has several advantages over the shader-based model. Because CUDA is an extension of C, there is no longer a need to understand shader-based graphics APIs. This allows for a shallow learning curve for most of C/C++ programmers. The CUDA programming framework also supports the use of memory pointers, which enables random memory-read and write-access ability. Therefore, GPU programming has arguably become as easy as general CPU programming. In addition, the CUDA framework provides a controllable memory hierarchy which allows the program to access the cache (shared memory) between GPU processing cores and GPU global memory.

2.2 *Data-Parallel Algorithms and GPU Acceleration*

Data-parallel algorithms [21] are a set of algorithms in which the same set of instructions is applied to all the elements in a data set. In other words, all these algorithms run in a SIMD (Single Instruction, Multiple Data) fashion, where many parallel processors are concurrently operating on multiple data elements, rather than concurrently executing different lines of code. Such algorithms feature independence among parallel threads, with no or little communication in between. Examples of these algorithms include image processing algorithms, N-body simulation, parallel reduction, and scanning of large arrays.

Hillis et al. [9] provides a general description of data-parallel algorithms. They also described a general implementation of such algorithms on multi-core system with more than 10,000 processors. Their claim is to use $O(N)$ processors to solve problems of size N, we can get a time complexity of $O(\log N)$. Siege and Wang [20] surveyed some issues regarding how to design efficient data-parallel algorithms for large-scale parallel machines, such as optimal data distribution for a given problem-machine combination, minimization of I/O latency, and algorithm scalability.

The design of graphics processing pipeline is a typical hardware implementation of data-parallel processing framework. Therefore, GPUs are the ideal candidate for cost-efficient computation hardware for data-parallel algorithms [15]. Recently, He et al. [7] studied two basic data-parallel algorithms, gather and scatter, and reported a performance speedup of 2-7 times. The paper mainly focused on the optimization strategies on GPU memory performance. The authors further extended their research results to a database application which relies on a data-parallel operation, called Relational Joins [8]. Sengupta et al. [19] described the GPU implementation of a set of "scan" based data-parallel computational primitives and demonstrated

their results with general algorithms such as sorting and sparse matrix multiplication.

2.3 *GPU Computation in Computer Vision*

Computer vision algorithms rely on image processing methods, which operate on a large amount of two-dimensional (2D) pixel data. Most computer vision and image processing tasks perform the same computations on a number of pixels, which is a typical data-parallel operation. Thus, they can take advantage of SIMD architectures and be parallelized effectively. We found several applications of GPU technology for vision reported in the literature. De Neve et al. [12] demonstrated a pixelwise YCoCg-R color transform on a shader-based GPU program. Using gradient vector field computation, Canny edge extraction, and a geometric model of a hand, Ohmer et al. [14] realized real-time hand tracking. Ready et al. [18] constructed a tracking system for aerial image stabilization that can track up to 40 points in real-time. Their system computes the stabilization transform in the CPU and does the image rectification in the GPU. Therefore, it features extensive communication between the CPU and GPU, and consequently, suffers from steep overheads in CPU-GPU data transfer. Mizukami and Tadamura [11] implemented Horn and Schunck's [10] optical flow algorithm based on the CUDA programming framework. Making use of on-chip shared memory, they were able to get approximately 2.25 times speed up using a NVIDIA GeForce 8800GTX card over a 3.2-GHz CPU on Windows XP.

3 *Data Parallel Algorithms*

Data-parallel computation allows for massively parallel computation when the data is strongly independent. This computation follows the Single Instruction Multiple Data (SIMD) paradigm. On an architecture with many cores, each core executes the same set of instructions, but on different subsets of data. This type of computation relies on the fact that the result of each data element is independent of each other. In this section, we will examine two data-parallel operations, convolution and accumulation. The discussion over these two operations is by no means exhaustive, but will reflect most implementation issues related to data parallel algorithms.

3.1 *Convolution*

Convolution is a common signal and image processing operator. The general formulation of a 2D convolution in the spatial domain is defined as [4]:

$$w(x,y) \star f(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x-s, y-t), \text{for all } x \in [1,n], \text{and } y \in [1,m], \tag{1}$$

where $w(x,y)$ is a 2D filter kernel of size (a,b), $f(x,y)$ is a 2D image of size (n,m).

Methods for convolution in the frequency domain are faster when convolution kernel sizes are large. This is because the cost of a convolution in the spatial domain depends on the kernel size as well as the image size. The running time of

convolution in the frequency domain are dependent only on the image size. However, a fourier transform is required to transform the image into the frequency domain, and the overhead is too large to justify when kernel sizes are small. For our purposes, we deal with small kernel sizes, and therefore only convolution in the spatial domain will be examined.

A general algorithm to convolve a kernel w of size (a, b) and an image f of size (n, m) is described as follows:

Algorithm 1 *Convolution(w, f)*.

Input: 2D kernel w of size (a, b), 2D image f of size (n, m)
Output: Filtered image h of size (n, m)
1: **for all** (x, y) in f **do**
2: **for all** (s, t) in w **do**
3: $h[x, y]$ += $w[s, t] \times f[x - s, y - t]$
4: **end for**
5: **end for**

The running time of this convolution algorithm with the image size of (n, m) and the kernel size of (a, b) is $x \times y \times a \times b \times C$, where C is the cost of one multiplication, one addition, and one memory access.

3.2 *Accumulation*

Accumulation is another widely applied mathematical operations, which is commonly used in image processing and realistic rendering. We examine the accumulation of a series of images or matrices.

Accumulation of k 2D images $f_i, 1 \leq i \leq k$ of size (n, m) is defined as:

$$g(x, y) = \sum_{i=1}^{k} f_i(x, y), \text{for all } x \in [1, n], \text{and } y \in [1, m]. \tag{2}$$

A general algorithm for performing accumulation k images $f_i, 1 \leq i \leq k$ of the same size (n, m) would be:

Algorithm 2 *Accumulation(k, f_i)*.

Input: k 2D images $f_i, i \in [1, k]$ of the same size (n, m)
Output: Accumulated image g of size (n, m)
1: **for all** (x, y) in f **do**
2: **for all** i in $[1, k]$ **do**
3: $g[x, y]$ += $f_i[x, y]$
4: **end for**
5: **end for**

The running time of an accumulation of k images of size (n, m) is $k \times n \times m \times C$, where C is the cost of an addition.

Most applications are using a variation of the accumulation algorithm, weighted accumulation. Weighted accumulation is analogous to blending, with each accumulated value having a weight associated with it. Weighted accumulation is defined

as:

$$g(x,y) = \sum_{i=1}^{k} w_i \times f_i(x,y), \text{for all } x \in [1,n], \text{and } y \in [1,m]. \quad (3)$$

The running time of a weighted accumulation of k images of size (n,m) is $k \times (W + n \times m \times C)$, where W is the cost of computing or retrieving a single weight, and C is the cost of a multiplication with the weight and an addition.

4 *General Performance Optimization Strategies with CUDA*

In modern computer architecture, there are two types of execution stalls, instruction stall and data stall. Because the GPUs have relative high memory access latency, access to instruction memory and data memory can be the major performance bottleneck.

Most of the instruction stalls are coursed by branching instructions, where instruction cache has to be flashed and a cache miss occurs. To optimize the performance for instruction stall, the general strategy is how to design the parallel algorithm to avoiding instructional branching. Data stall is another significant performance killer. However, Geforce 8 series GPU architecture, shown in Figure 1(a), supports very high memory bandwidth. To minimize data stall on GPU, the general strategy is to take the advantage of controllable memory hierarchy to hide memory latency.

In this section, we will discuss these two general performance optimization strategies for CUDA supported GPUs in terms of how to organize different thread/block structures.

4.1 *Avoiding Instructional Branching*

Performance decreases when two different code paths exist. Divergent branching occurs when threads within a warp perform different instructions. This is known to decrease performance. A non-divergent branch occurs when there are two code paths, but all threads within a warp follow the same code path. We have found this to be a significant factor in performance. Every control statement creates a branch, which in includes conditional if-else statements and iterative while and for loops.

A strategy to minimize looping in a GPU kernel is to split the workload among threads and blocks so that each thread performs one iteration of the loop. This is ideal when each iteration inside of a loop does not depend on previous iterations. Also, the number of allowable blocks in a single dimension on a G80 GPU is 65535, which is relatively large. The cost of block replacement is much lower than the cost of a branching instruction, as shown by our results.

Simply reducing iterations and increasing blocks will not be sufficient to obtain performance gain. The same total number of branches occurs, because if the loops in one block are halved then the number of blocks is doubled. It is necessary to completely eliminate the $for - loop$ structure in code, and the number of blocks should be determined by the host at runtime. However, if for a given algorithm that the number of iterations is known at compile time, it is possible to tell the compiler to unroll the loop. The compiler simply repeats the code within a loop a specified number of times.

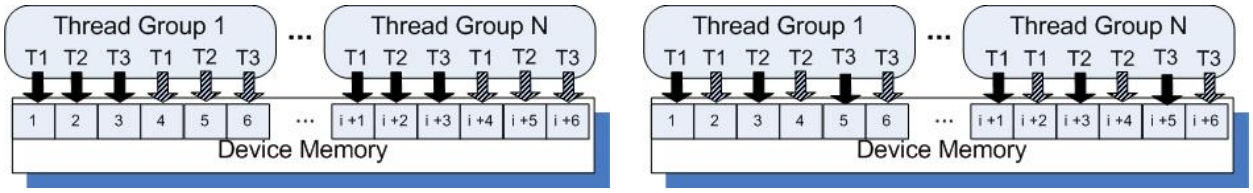


Figure 2: (a) Example of a coalesced memory access pattern (b) Example of a non-coalesced memory access pattern

4.2 *Hiding Memory Latency*

Data transfer between host memory to device memory The transfer of data between the host machine and the GPU device is limited by the bus. For most setups, this is limited by the PCI-Express x16 maximum theoretical bandwidth of 4GB/s. However, applications do not reach that speed, and typically reach a maximum of 2GB/s. This is slow compared to memory transfer speeds within the device, which has a theoretical maximum of 86GB/s. It is best to minimize the amount of data transferred, even if a part of an algorithm exhibits poor parallelism. It is a trade-off between running a poorly data-parallel algorithm on the GPU, or incurring a host-device memory transfer.

Global and texture memory access Global memory is a linear memory space and the access to global memory is not cached. In order to hide the high latency to access global memory, the access pattern for all threads should follow a coalesced fashion. Figure2 illustrates two different global memory access pattern with and without coalesced fashion. In Figure2 (a), a group of threads is performing a coalesced memory access, where the accesses to the global memory among the threads are consecutive. If the memory access is coalesced, all the reads from the group of threads are actually executed as one single read. In Figure2 (b), because the access among the threads are not consecutive, the access pattern is not coalesced. For example, if n memory requests can be synchronized with n threads together for a single coalesced access, we only hit the memory latency once, comparing with $n - 1$ times with non-coalesced access. According to NVIDIA's optimization tutorial [6], coalesced memory access is about ten times faster than non-coalesced access.

The texture memory is optimized for 2D spacial locality. Reading from texture memory is cached so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from the texture cache. If the access to device memory can not be organized for coalesced access, one should make use of the texture cache as much as possible by maximizing the locality of computation within one block.

Share memory usage As illustrated in Figure 1(a), shared memory is fast on-chip read/write memory, served as controllable L1 cache between device memory and GPU processors. Within a single block, all threads have access to the same section of shared memory. This allows for communication among threads within a thread block. Random access to this memory is fast with a latency of 4 clock cycles for an access, compared to a latency of 400-600 cycles for an access to global

memory. However, shared memory size is limited to 16kB per multiprocessor. If an algorithm is heavily data-intensive and a parallel operation requires a large amount of memory as input, then the limited amount of shared memory is not enough. Also, if a block requires more than half of the shared memory, then only one block may run on the multiprocessor at a time. A good data-parallel algorithm will split the data into pieces small enough so that multiple blocks can share a multiprocessor. As the number of concurrently running blocks increases, memory latency is less noticeable.

A limitation of shared memory is possibility of contention. Shared memory is split into 16 banks, and only one thread in a half warp can access a single bank at a time. If 6 threads are accessing a single bank, then the threads are serialized in order to resolve the conflict. Sometimes the thread manager broadcasts the value of an address if enough threads are accessing it. In this case, there is no serialization due to the word being broadcast. A strategy to improve performance is to have threads access a single 32-bit word at a time, with each word being assigned to one thread.

5 *Performance Optimization of Convolution and Accumulation Using CUDA*

Given the definition and pseudo-code of convolution and accumulation in Section3, we will exercise the general optimization strategies of CUDA-based GPU implementation on these two data parallel operations.

5.1 *Convolution*

In order to exam the two proposed optimization strategies on convolution, we design different GPU implementations and evaluate them with different algorithm parameters.

Implementation design When considering different implementations of convolution kernel, we explore the following three variables: the source image location, shared memory usage and thread workload. In terms of the source image location, we want to compare texture fetch versus global memory access. To exam the share memory performance, we create different implementations with or without using shared memory. When using shared memory, the GPU kernel first loads a block of source pixels into shared memory, and then execute convolution on them. Without shared memory usage, the kernel reads pixel data directly from global memory or texture memory. In the end, we also want to explore how different thread workloads can affect the performance of convolution. Please find our experimental kernels defined in Table1.

For the first four kernels defined in Table1, we organize the threads and blocks in a way that the total number of threads of all blocks is equal to the number of pixels in the source image. For example, in Figure3, the size of the image is 32×32 and we create 256 threads per block. Therefore, each block is organized into a 16×16 matrix of threads and covers one fourth of the image which contains 16×16 pixels. Each thread is then assigned to a pixel at the corresponding location.

We create the fifth kernel, where each thread computes multiple pixels in a $for - loop$ structure. Therefore there are fewer blocks than that in the first kernel.

Kernels	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5
Source image location	Texture memory		Global memory		Texture Memory
Shared memory Usage	No	Yes	No	Yes	No
Thread workload	Single pixel				Multiple pixels

Table 1: GPU convolution kernels

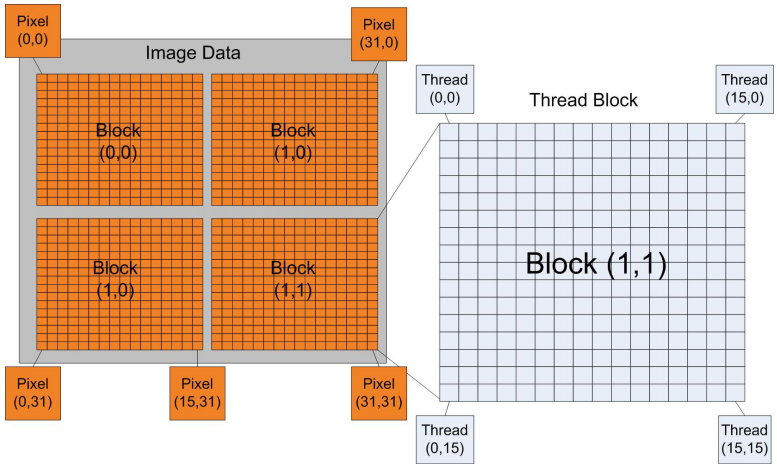


Figure 3: Thread organization illustration for convolution kernels 1-4

Due to the irregular data fetching caused by the image boundaries, reading from global memory cannot be coalesced. But in all the kernels, writing the result back into global memory is coalesced access. When shared memory is used , we eliminated bank conflicts as much as possible.

For all defined kernels, the the total amount of computation is $O(N_p \times N_c^2)$, where N_p is the number of pixels in the image and N_c is the convolution size. The difference between these kernels is how the computation is divided among the blocks and threads, thus there will be different degree of parallelism. Theoretically, given the number of threads per block $N_{threads}$, the computation workload for each thread is $O(N_{threads} \times N_c^2)$ for the first four kernels, and $O(N_{threads} \times N_c^2 \times N_{loops})$ for the last kernel, where N_{loops} equals the number of result pixels calculated by one thread.

Implementation evaluation We design and evaluate kernel 1 and kernel 5, in order to exam how instructional branching affects the performance. Each thread in Kernel 1 covers the computation for one pixel, so there is no $for - loop$ structure, therefore, no branching instruction. In kernel 5, each thread covers the computation of more than one pixels by using a $for - loop$ structure. The performance comparison of these two kernels is illustrated in Table 2. In these experiments, the image size is 512×512, the convolution size is 3×3, the number of thread per block is 384. The results show that the performance for no-branching implementation is 2 times faster than the one with branching instructions.

We compare different implementations with different memory access strategies with the design of the first four kernels, and show the performance evaluation in Figure4 and Table3. In these experiments, the image size is 512×512 and the

Kernels	GPU Time (ms)	Occupancy	Number of Branches
Kernel 1	196.384	0.5	0
Kernel 5	400.8	0.5	576

Table 2: GPU convolution performance with different thread workload. The image size is 512×512, and the convolution size is 3×3.

Threads per block	64	128	192	256	320	384
Kernel 1	285.664	269.568	262.272	131.68	236.832	196.384
Kernel 2	274.72	283.68	264.288	250.592	251.712	226.592
Kernel 3	1937.89	1888.61	1784.99	1172.54	1711.84	1500.99
Kernel 4	601.12	572.832	549.632	457.44	521.056	472.532

Table 3: GPU convolution performance with different numbers of threads per block. The image size is 512×512, and the convolution size is 3×3. The time is in milliseconds.

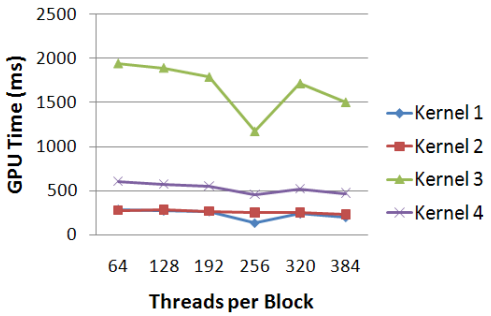


Figure 4: GPU convolution performance with different numbers of threads per block. The image size is 512×512, and the convolution size is 3×3.

convolution size is 3×3. As shown in Figure4, the performance of kernel 1 and kernel 2 is faster than kernel 3 and kernel 4. The speedup of texture memory access over global memory access is between 6.78 to 8.9 when not using shared memory, and 1.825 to 2.188 when using shared memory. In terms of shared memory usage, the speedup for using shared memory is 2.63 to 3.25 when accessing global memory in a non-coalesced fashion.

However, when accessing texture memory, it is not necessary to use shared memory as the cache, because the texture memory access is already cached with hardware control. That is the reason why, in our experiments for convolution size of 3×3, the performance is a little slower when using shared memory to cache texture memory access.

Based on all the analysis of our experiments above, for convolutions with an image size of or below 1024×1024, we recommend the implementation type of kernel 1 if the convolution size is 3×3.

5.2 *Accumulation*

Accumulation is more memory-intensive than it is computationally intensive. The major bottleneck of an accumulation operation is the memory access. The input data comes from global memory, which is characterized by its high-bandwidth but high-latency memory access cost. This latency can be hidden by increasing the number of concurrently running blocks.

More blocks keep the GPU busy while threads wait for requested memory.

We create an implementation where each thread accumulates all values for one or more pixels. So, if there are m images to accumulate, a thread sequentially sums m values. If the image size is small, the GPU will be underutilized because there are more threads able to be executed than there are pixels. In weighted accumulation, multiple accumulation operations can be performed, each with different weight parameters. Separate accumulation operations can be performed in parallel. With multiple accumulations, the GPU can be fully utilized, even with small images.

Since there may not be enough threads for the block to operate on the entire image, it may be split among multiple blocks. For example, if each block has t threads for an image of n pixels, then the number of blocks is n/t. This assumes that the number of pixels per image is greater than or equal to the number of threads in a block. In weighted accumulation, different weights might be applied to create multiple accumulations. So if m accumulations are to be performed, then $m \times n/t$ blocks must be created. A large number of blocks will keep the GPU busy, and will scale to future generations of GPUs.

Implementation design To examine the proposed optimization strategies on accumulation, we design experiments that explores the following variables that affect accumulation performance: pixels per image, images per accumulation, accumulations performed, and number of threads. In addition, two implementations are developed to test the effect of looping, and subsequently branching, on accumulation performance. The first implementation allows a single thread to operate on multiple pixels. As defined in Algorithm2, the outer loop controls which pixel the thread is operating on, and the inner loop controls which image is currently being accumulated. The second implementation eliminates the outer loop, and each thread only operates on a single pixel.

Figure 5 (a) shows how occupancy and blocks help to hide memory latency. As the number of blocks increase, the time per computation will decrease to a certain point for two reasons. The first reason is if there are less blocks than multiprocessors, the GPU will be underutilized. The second reason is when more blocks occupy a multiprocessor, there are more threads for the multiprocessor to execute. When there are more threads to execute, latency can be hidden by executing threads that are ready while others are waiting for global memory. There is a maximum limit to how many blocks can occupy a multiprocessor, so when there are massive numbers of blocks, they are queued until a multiprocessor can execute them. Memory intensive applications perform best when there are large numbers of high-occupancy blocks.

It is not always clear how to determine the optimal number of threads. For our implementation, it is determined empirically, as shown in Figure 5 (b). The optimal number of threads per block to use is 64, while 512 threads is the next best thread size. Thread sizes of 256 and 128 performed worse, with 256 threads having a high variance compared to other thread sizes. Having 32 threads per block proved to be the worst, being 33 percent slower than the optimal thread size.

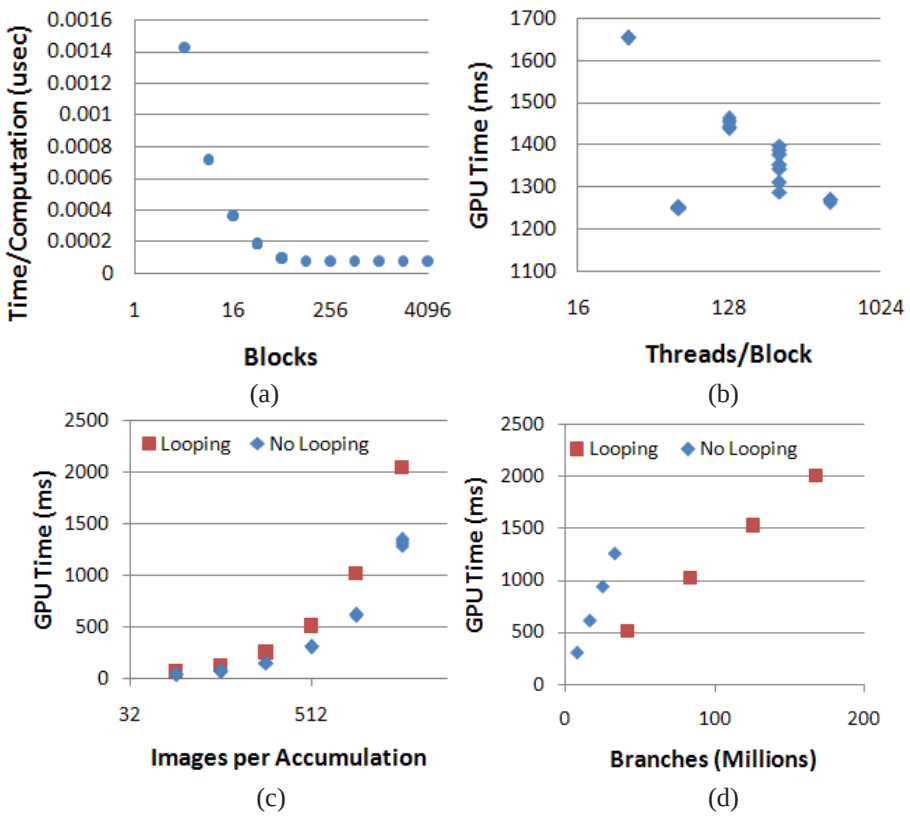


Figure 5: (a) Time per unit of computation vs. number of blocks (b) Total accumulation time vs. threads per block (c) Images per Accumulation vs. GPU Time (ms) (d) Branches vs. GPU Time, with varying Pixels per Image

Branching has a major impact on accumulation performance. Figure 5 (c) shows that the implementation with looping performs consistently worse on different input sizes. Figure 5 (d) shows the relationship between branching and performance. Looping on the GPU performs worse because before each loop iteration, there is a termination condition check. This termination check counts as a branch, which reduces the speed. Nested loops have an exponential effect on the number of branches. Any effort to replace loops entirely with increased blocks will increase performance. Between the two implementations of accumulation, the one without any loop consistently performed the best. Without looping, there was a 1.6x speedup.

6 *Case Study - VCM*

To facilitate a performance comparison between proposed approaches and demonstrate GPU optimization strategies for data parallel applications, we chose Vector Coherence Mapping (VCM) algorithm [16, 17] as our test application. VCM algorithm is for the computation of an optical flow field from a video image sequence. By applying spatial and temporal coherence constraints with fuzzy image processing operation, it tracks sets of interest points in the frame in parallel. A voting scheme is featured to enforce the constraints on vector fields. Figure 6 describes how VCM incorporates spatial coherence constraints with the correlation for tracking interest points in a flow field. Three detected points are shown at

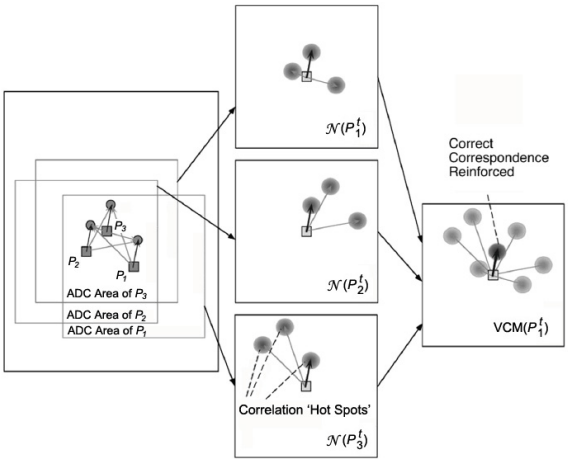


Figure 6: Algorithm illustration for Vector Coherence Mapping.

the top of the picture where the shaded squares represent the position of the three interest points at time t and the shaded circles represent the corresponding position of those points at $t + 1$. Correlation matching results for each point, which is labeled $N(p1')$, $N(p2')$, and $N(p3')$, provide three hotspots as shown at middle of picture. (The correlation matching result is called Normal Correlation Map (NCM).) By using weighted summation of this neighboring NCMs, we can obtain Vector Coherence Map of point p_i^t with minimizing the local variance of the vector field as shown at the bottom of figure. VCM algorithm is inherently parallel since it employs convolution and pixel-wise summation as it's dominant operations. Input data of the algorithm is given in the form of video sequences. Hence implementation of VCM on GPU allows us to experiment for parallel data processing on GPU with different strategies for each kind of operation.

6.1 *Algorithm Decomposition for GPU Acceleration*

Computational processing in VCM can be decomposed to three phases by their data dependencies on calculation and pattern of computation: 1) Interest Point (IP) extraction, 2) Normal Correlation Map (NCM) computation, and 3)Vector Coherence Map (VCM) computation. Each of these steps presents a different memory access and computational profile. The characteristics of each phase are illustrated in Table 4.

IP extraction applies convolution and image differencing operations to every pixel in two frames. Then sorting and selection is applied to the result values to set top n values as IP list. NCM computation performs convolution operation with IP list in image $I_t$and next image I_{t+1}, then generates 1 NCM for each IP. The entire NCM array should be maintained for VCM computation. VCM computation requires a massive number of accumulations. Summation of multiple NCMs is the major computational task to complete VCM computation. Scanning the result of VCM for finding the strongest motion vector is also performed in this phase. Each phase is processed for each frame in video stream, and the results are returned as a list of vectors, which means moving interest points across two successive frames.

Phase	Computation	Input	Output
IP Extraction	Convolution Image differencing IP sorting/selection	2 images	IP list
NCM	Correlation	2 images, IP list	1 NCM per IP
VCM	Accumulation Maximum detection	IP list, 1 NCM per IP	Resulting vector

Table 4: Computation phases in the VCM algorithm

6.2 *Interest Point Detection (Convolution)*

To ensure even distribution of interest points over the image, each video frame is segmented into 16 x 16 sub-windows and two interest points are detected per sub-window. Two methods are used to extract interest points on the frame. In the first approach, Sobel operator is applied to estimate the spatial local gradients. When the Sobel edge detector is calculating the weighted sum of a neighborhood pixels, it uses a pair of 3x3 convolution matrices for weight information, and each value from the neighborhood pixel is multiplied with its corresponding weight from the mask. The convolution is expensive if it is handled in sequential manner since it requires retrieval of all the neighborhood and all the element in the mask. However there is no data dependency to calculate each weighted neighbor pixel so that parallel computation is applicable to this process for efficiency. A series of sub-windows are processed by a set of CUDA blocks in parallel. In the second phase, image differencing is employed to estimate temporal changes for moving interest points. Results from two phases are incorporated into spatio-temporal gradient as fuzzy-and operation is applied to them. Once all of the pixels in the current image are processed, each pixel value is sorted to pick the highest two interest points. To be efficient, the sorting process is optimized as well; a thread picks up a value in sub-window array and compares it sequentially with other values within the array.

6.3 *NCM (Correlation)*

NCM computation is based on Absolute Difference Correlation. A 5 x 5 neighborhood around interest point P_i^t in frame I^t is correlated to 64 x 64 area in frame I_{t+1} which of center is positioned on P_i^t. The way of calculating correlation is exactly the same with convolution in that it is multiplying the pixel by the value from the corresponding matrix. The only difference is when multiply the pixel, the equivalent position value is used instead of opposite position in the matrix which is in convolution. Hence correlation is parallelized across the available threads, and correlation value of each pixel is generated by each thread. When the resulting NCM is passed for VCM computation, an optimized way of writing should be used since global memory access is costly.

6.4 *VCM (Accumulation)*

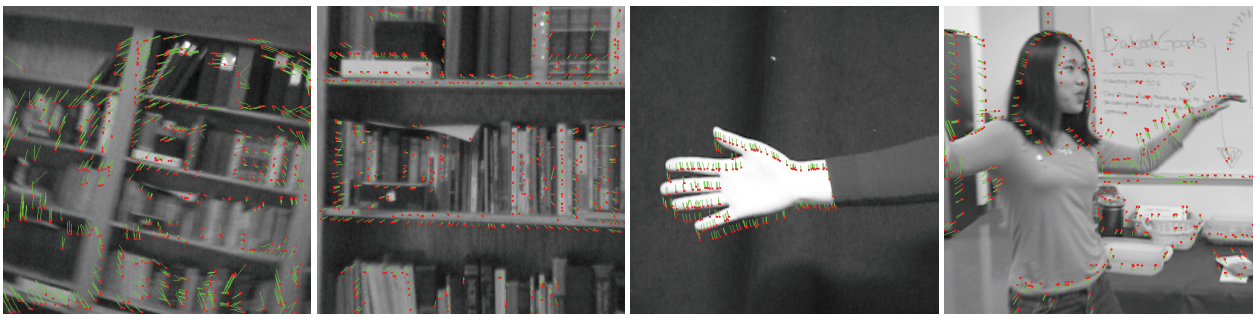
A VCM is computed by pixel-wise accumulation. To decide the contribution of the NCM of a point p_j^t on vector p_i^t, a weighting function based on the distance between two points are applied to the corresponding NCM in the form of pixel-wise multiplications. Each point is assigned to a block, which instantiates a 64 x 64 array for maintaining the VCM being accumulated. The summation process can be parallelized in different ways as we discussed above. By using this weighted summation of neighboring NCMs, VCM estimates the termination point of the new position of p_i^t in the next frame.

6.5 *Data Access and Pipeline*

While GPU kernels are being executed, the CPU is also capable of running. By making independent CPU and GPU operations run concurrently, the overall running time can be reduced. Among the CPU operations in our implementation, disk access consumes the most time. On one of our test computers, loading a single frame of video takes 0.0521 seconds. On the other hand, the kernel performing the VCM phase on the GPU is the most expensive kernel execution, leaving the CPU idle the longest time. Plus, this phase requires only the NCMs from the previous phase, not the video frames. Therefore, modifying a video frame during calculation will not affect the result. As a result, we choose to load the next video frame during the VCM phase instead of loading it sequentially after the VCM phase is complete. This leads to a decrease of the average overall running time per frame from 0.177 seconds to 0.125 seconds. It shows that taking advantage of the opportunities for concurrent execution can result in a significant speedup.

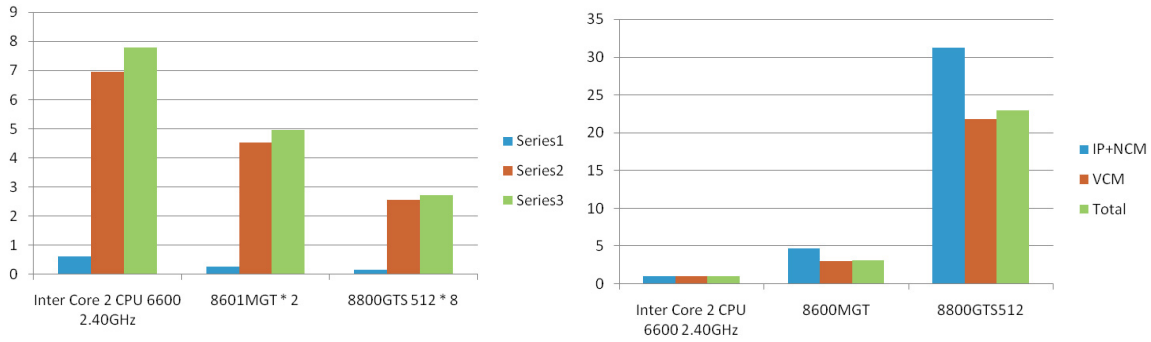
6.6 *Results*

In this section we present the results of video stream analysis with GPU implementation of VCM algorithm. All examples are real video sequences and represent different type of vector movement. The computed vectors are rendered on top of the respective image. Figure 7(a) and 7(b) show analysis of video streams with global dominants field. At figure 7(a), the camera is rotated on its optical axis. Even fair amount of motion is blurred, camera movement detected precisely. Figure 7(b) shows the results of zoom-out video stream. Again, the result shows VCM's ability to track the motion exactly though



(a) (b) (c) (d)

Figure 7: Results: (a) Camera rotating on its optical axis, (b) Camera zooming out, (c) Hand moving up, (d) Girl dancing with camera zooming in



(a) (b)

Figure 8: (a) Time Comparison between GPU and CPU operation: The 8600MGT time was multiplied by 2, and the 8800GTS-512 was multiplied by 8 for visibility (b) Speed Comparison between GPU and CPU operation: The speed of CPU was set at 1.0 for to be base comparison

image is grainy and movement is very subtle (Small vectors that converges at the center of scene). At figure 7(c), the sequence of hand moving is analyzed. The subject is dancing while camera is zooming in the subject in figure 7(d).

To evaluate effectiveness of our approach for utilizing GPUs as massive data-parallel processor, we used two different implementation of VCM algorithm; one supports utilizing GPU and the other is based on CPU. For the comparison, two different GPUs executed the GPU version code: a 8600MGT which is equipped on Apple MacBookPro, and a 8800GTS-512. The CPU version is executed on 2.4 GHz Inter Core 2 with Windows XP. The algorithm is implemented in the similar way as much as possible; both of them used the same data structure for IP, NCM, and VCM. The same number and size of sub-windows and the same size of IP list, NCM, and VCM were used. Graph 8(a) and 8(b) show experiment results with 2048 IPs. In the time graph, 8600MGT operation time had to be multiplied by 2 and 8800GTS-512 operation had to be multiplied 8 for visibility. The speed of CPU operation was used as the base comparison in the speed up graph. 8600MGT showed 3.15 times speed enhancement, and 8800GTS-512 showed 22.96 times performance enhancement.

7 *Conclusion*

Data parallel algorithms have a wide range of applications in computer graphics, computer vision, database and bioinformatics. Recently, with the advances in graphics computing hardware, people can rely on a cost-efficient parallel computing resource provided by the GPU. In this paper, we rigorously analyze two fundamental data parallel operations, convolution and accumulation, with respect to the CUDA enabled GPU implementation. We identify two general GPU optimization strategies, branch avoiding and memory latency hiding, and thoroughly explore different ways of parallel implementation and hundreds of different algorithm parameters of the two data parallel operations. Based on the experiment results, we recommend a standard CUDA-based GPU implementation for convolution and accumulation. To further demonstrate our proposed GPU optimization strategies, we adopt a data parallel computer vision algorithm as our case study application. The algorithm, called VCM, consists of two convolution operations and one weighted accumulation operation. Our GPU

implementation of VCM exhibits a performance gain of more than 22 times of speedup comparing with a state-of-art CPU implementation.

Acknowledgement

This research has been partially supported by NSF grants Embodied Communication: Vivid Interaction with History and Literature, IIS-0624701, Interacting with the Embodied Mind, CRI-0551610, and Embodiment Awareness, Mathematics Discourse and the Blind, NSF-IIS- 0451843.

References

[1] NVIDIA's Compute Unified Device Architecture. `http://developer.nvidia.com/object/cuda.html/`.

[2] Blaise Barney. Introduction to Parallel Computing. `https://computing.llnl.gov/tutorials/parallel_comp/`, 2007.

[3] Magnus Ekman, Fredrik Warg, and Jim Nilsson. An in-depth look at computer performance growth. *SIGARCH Comput. Archit. News*, 33(1):144–147, 2005.

[4] Rafael C. Gonzales and Richard E. Woods. *Digital Image Processing, 3rd Edition*. Prentice Hall, New Jersey, USA, 3rd edition, 2008.

[5] Jim Gray, David Liu, Maria Nieto-Santisteban, Alex Szalay, DeWitt, and Gerd Heber. Scientific data management in the coming decade. *Cyberinfrastructure Technology Watch*, 1(1), February 2005.

[6] Mark Harris. Cuda tutorial: Optimization, part i. `http://www.astrogpu.org/talks/NVIDIA/AstroGPU.4.Optimization.Harris.pdf`, Nov. 2007.

[7] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC07)*, Reno, NV, November 2007.

[8] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 08)*, Vancourver, Canada, June 2008.

[9] W. Daniel Hillis and Jr. Guy L. Steele. Data parallel algorithms. *Communication of the ACM*, 29(12):1170–1183, 1986.

[10] Berthold K.P. Horn and Brian G. Schunck. Determining optical flow. Technical report, Cambridge, MA, USA, 1980.

[11] Yoshiki Mizukami and Katsumi Tadamura. Optical flow computation on compute unified device architecture. In *ICIAP*, pages 179–184, 2007.

[12] Wesley De Neve, Dieter Van Rijsselbergen, Charles Hollemeersch, Jan De Cock, Stijn Notebaert, and Rik Van de Walle. Gpu-assisted decoding of video samples represented in the ycocg-r color space. In *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, pages 447–450, New York, NY, USA, 2005. ACM.

[13] NVIDIA Corporation, Santa Clara, California, USA. *NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, Version 1.1*, November 2007.

[14] J.F. Ohmer, F. Maire, and R. Brown. Real-time tracking with non-rigid geometric templates using the gpu. *Computer Graphics, Imaging and Visualisation, 2006 International Conference on*, pages 200–206, 26-28 July 2006.

[15] John Owens. Data-parallel algorithms and data structures. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, page 3, New York, NY, USA, 2007. ACM.

[16] F. Quek, Xin-Feng Ma, and R. Bryll. A parallel algorithm for dynamic gesture tracking. *Recognition, Analysis, and Tracking of Faces and Gestures in Real-Time Systems, 1999. Proceedings. International Workshop on*, pages 64–69, 1999.

[17] Francis K. H. Quek and Robert K. Bryll. Vector coherence mapping: A parallelizable approach to image flow computation. In *ACCV '98: Proceedings of the Third Asian Conference on Computer Vision-Volume II*, pages 591–598, London, UK, 1997. Springer-Verlag.

[18] J.M. Ready and C.N. Taylor. Gpu acceleration of real-time feature based algorithms. *Motion and Video Computing, 2007. WMVC '07. IEEE Workshop on*, pages 8–8, Feb. 2007.

[19] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for gpu computing. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[20] Howard Jay Siegel and Lee Wang. Data parallel algorithms. Technical Report TR-EE 94-38, december 1994.

[21] Gregory V. Wilson. A glossary of parallel computing terminology. *IEEE Parallel and Distributed Technology*, 1(1):52–67, 1993.