Applying Parallel Design Techniques to Template Matching with GPUs

Robert Finis Anderson, J. Steven Kirtzic, and Ovidiu Daescu

University of Texas at Dallas
Richardson, TX, USA
{rfa061000, jsk061000, daescu}@utdallas.edu

Abstract. Designing algorithms for data parallelism can create significant gains in performance on SIMD architectures. The performance of General Purpose GPU's can also benefit from careful analysis of memory usage and data flow due to their large throughput and system memory bottlenecks. In this paper we present an algorithm for template matching that is designed from the beginning for the GPU architecture and achieves greater than an order of magnitude speedup over traditional algorithms designed for the CPU and reimplemented on the GPU. This shows that it is not only desirable to adapt existing algorithms to run on GPUs, but also that future algorithms should be designed with their architecture in mind.

1 Introduction

The advent of massively multiprocessor GPUs has opened a floodgate of opportunities for parallel processing applications, ranging from cutting-edge gaming graphics to the efficient implementation of classic algorithms [1]. In this paper we refer to the desktop machine containing the GPU as the "host".

Figure 1 depicts the structure of the NVIDIA GeForce 8800 series as an example of a typical GPGPU (General Purpose GPU) device. The GeForce 8800 contains 16 multiprocessors, each containing 8 semi-independent cores for a total of 128 processing units. Each of the 128 processors can run as many as 96 threads concurrently, for a maximum of 12,288 threads executing in parallel. The computing model is SIMD (single instruction multiple data), and the memory model is a NUMA (non-uniform memory access) with a semi-shared address space. This stands in contrast to a modern desktop or server PC's CPU, which is typically either SISD (single instruction single data) or MIMD (multi-instruction multiple data), in the case of a multi-processor or multi-core machine. Additionally, from the perspective of the programmer, all memory is explicitly shared (in multi-threading environments) or explicitly separate (in multi-processing environments) on a desktop machine.

These differences in processor architectures lead to different programming models, with different optimal algorithm designs. For an example of an algorithm design under similar architectural constraints, see [2]. Likewise, for a good introduction to the differences in algorithm analysis for various architectures,

which must take into account not only running time, but also the amount of idle processing power and the amount of extra work done in a parallel setting over the best serial algorithms, see [3].

In addition to these considerations, the GPGPU has one more unique constraint: the connection bandwidth between the CPU and the GPU is quite limited compared to the bandwidth of the GPU's internal memory [4, 5]. In fact, given that the GPU cannot directly access the host's main memory, hard drives, or peripherals, and modern hosts can contain multiple interconnected GPU units, dealing with the GPU can be thought of as distributed computation on a small local network with the host acting as a control node.

In this paper we present a GPU-based algorithm design for image template matching, which is a building block for many high-level Computer Vision applications, such as face and object detection [6, 7], texture synthesis [8], image compression [9, 10], and video compression [11, 10]. Algorithms of this type are often infeasibly slow in raw form [12], and there has been much research into methods for accelerating template matching for various applications.

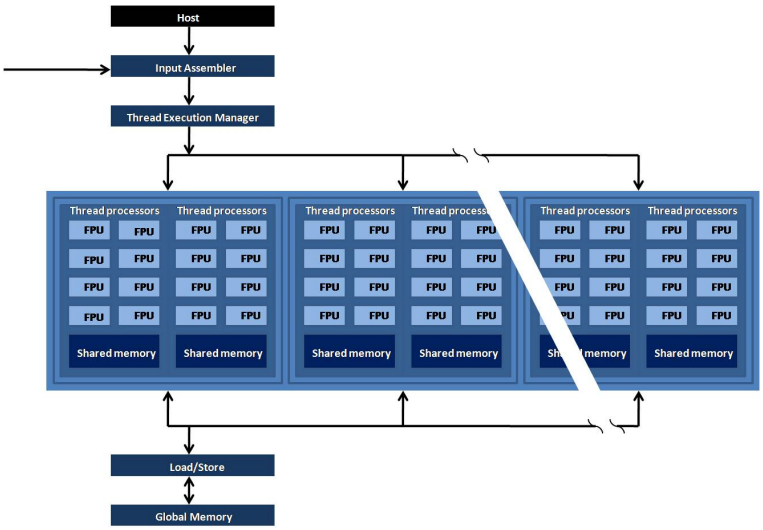

Fig. 1: NVIDIA GeForce 8800 Architecture

To date, there have been several attempts at adapting sequential algorithms to the data-parallel GPU architecture [13–16] rather than designing with data-parallelism in mind. In contrast, we designed an algorithm for GPGPU execution from the ground up, while analyzing the unique steps taken in the design process.

1.1 Template Matching Background

Some template matching acceleration methods ignore image information deemed irrelevant or unnecessary to reduce run time, or make use of statistical analysis to produce a likely answer, but are unable to guarantee finding the best match

according to the chosen error measure e.g. [17–19]. A second set of methods which has emerged recently makes use of bounds on the error measure to achieve acceleration without sacrificing accuracy, although the choice in error measures is somewhat more limited [12, 20, 21]. Our proposed algorithm falls into this second set.

Throughout this paper we make use of the l_1 norm-based distance measure (i.e. the sum of absolute differences) between the template and the image subwindow. We denote the l_1 norm of a vector x by $|x|$.

Let vector $x \in \Re^n$ represent the template we are matching. This vector is formed by concatenating the rows of the template image together into one long sequence. Let I represent the image we are searching, which is larger in all dimensions than the template image. We consider each template-sized subwindow y_i in I a potential match. The subwindows often overlap, and each of them contains n pixels. Each of these subwindows is converted into a vector using the same process as for x. For convenience we define $Y = \{y_1, y_2, \ldots y_m\}$ to be the set of all potential match vectors. In practice, m (the number of potential matches) is slightly less than the number of pixels in I.

The error for the i^{th} candidate (or sub-window) is: $E_i = |x - y_i|$. Given x and I, a template matching algorithm attempts to find the y_i which minimizes E_i. In accelerating template matching, we place bounds on the value of E_i, which we denote as $l_i \leq E_i \leq u_i$. We define those bounds using the triangle and Cauchy-Schwarz inequalities: $|y_i| - |x| \leq |y_i - x| \leq |y_i| + |x|$ Note that if we define an orthogonal set of masking vectors m_j, described in Fig. 2, we can define a tightening series of bounds on E_i by taking the major diagonal of the outer product of m_j with x and y_i to get x^j and y_i^j, where j is the index of the masking vector m. This is analogous to the "image strips" of [12]. Using these values we define a recursive relation on the series of bounds on E_i in Fig. 3.

2 Case Study: Full Search and On-Card Memory

We first consider the case of the Full Search Method of template matching, otherwise known as a brute force method. We have selected as that feature set the pixel values of x and y_i. For our purposes, we define E_i as the distance between the total pixel values of x and y_i. The traditional Full Search Method calculates

$$
\begin{array}{l|l}
m_0 & 0\ 0\ 0 \cdots 0\ 0\ 0 \cdots 0\ 0\ 0 \cdots 0\ 0\ 0 \cdots \\
m_1 & 1\ 1\ 1 \cdots 0\ 0\ 0 \cdots 0\ 0\ 0 \cdots 0\ 0\ 0 \cdots \\
m_2 & 0\ 0\ 0 \cdots 1\ 1\ 1 \cdots 0\ 0\ 0 \cdots 0\ 0\ 0 \cdots \\
m_3 & 0\ 0\ 0 \cdots 0\ 0\ 0 \cdots 1\ 1\ 1 \cdots 0\ 0\ 0 \cdots \\
\vdots &
\end{array}
$$

Fig. 2: The set of masking vectors m_j. The length of the sections of 1s and 0s is typically some constant fraction of image width * image height.

$$
\begin{aligned}
\text{diff}_0 &= |x^0 - y_i^0| \\
s_x^0 &= x^0 \\
s_y^0 &= y_i^0 \\
u_i^0 &= \text{diff}_0 + |x - s_x^0| + |y_i - s_y^0| \\
l_i^0 &= \text{diff}_0 + |x - s_x^0| + |y_i - s_y^0| \\
\text{diff}_j &= \text{diff}_{j-1} + |x^j - y_i^j| \\
s_x^j &= s_x^{j-1} + x^j \\
s_y^j &= s_y^{j-1} + y_i^j \\
u_i^j &= \text{diff}_j + |x - s_x^j| + |y_i - s_y^j| \\
l_i^j &= \text{diff}_j + |x - s_x^j| - |y_i - s_y^j|
\end{aligned}
$$

Fig. 3: Definition of the progressive bounds on E_i.

E_i for all $y_i \in Y$, and returns $y_{opt} = \arg\min_{y_i} E_i$. The algorithm is straightforward: as each E_i is calcuated, the algorithm compares it to a global minimum, updating as necessary. The first step in adapting an existing serial algorithm for implementation on a GPU is to analyze the algorithm and determine which parts (if any) can benefit from parallelization. Our GPU adaptation is very similar to the original, with the exception that after computing E_i at all locations simulatenously, the algorithm uses the 'reduce' subroutine [2, 16], commonly used in data parallel environments, to find a minimum or maximum. Given that m is the number of subwindows, and the template x contains n pixels, this approach runs in $O(mn)$ time, which comes to $\approx 4 * 10^{10}$ operations. GPU implementation of similar methods has been explored in [16]. The straightforward GPU implementation should run in $O(\frac{mn}{p} + \log m)$ time, where p is the number of processors, assuming that $1 \ll n$. This bound comes from mn work being done on p processors, and the reduction step which takes $\log m$ time. We present the actual results in Table 1. Compilation of the CPU code was performed by MS Visual C++ 2008 with all optimzations turned on, while the GPU code was compiled by NVIDIA's nvcc and optimized by open64 [22]. One can see that the ratio of runtimes of the CPU to naive GPU implementation (or "speedup", S) is only 7.14. Given the number of processing units p is 128, this is clearly not a cost optimal solution, as it yields an efficiency of .056 (from $E = \frac{S}{p}$). The majority of this is due to communication overhead, as main memory on the GPU is uncached. Experimentation confirms that the instruction throughput is only .034.

Most GPGPU architectures include a limited, local, user controlled cache. This local cache (which is called "shared memory") is typically too small to hold an entire image (in our case it is 16KB in size). Therefore the image must be loaded a portion at a time, and the threads sharing a given piece of memory synchronized. The groups of threads which can access a given piece of shared

memory are organized into "blocks". Threads within a block can typically use shared memory to communicate and synchronize with one another, but are unable to do so (directly) with threads outside of that block. Therefore, the input data should be broken up according to thread blocks when possible. In the case of template matching this is relatively easy, given that inputs y_i though y_{i+n} are the only information required to compute E_i through E_{i+n}. However, the values y_i and y_{i+1} overlap considerably, leading to a certain amount of replication. The results of this approach appear in Table 1 as "GPU Shared". While it represents a vast improvement, the instruction throughput (processor utilization) is still only around .5 due to synchronization, bank conflict, and redundant loading issues.

A fast, cached, read-only memory called "texture memory" is also available on most GPUs, which in practice operates at nearly the speed of the shared memory. This memory is effectively a cached version of the the GPU's main memory, which becomes read-only to prevent cache inconsistency. Using this memory eliminates the expensive synchronization step and its associated processor idle time. Using the texture memory to hold the template and the image, we see a speedup of $S = 212.23$. [1] Furthermore, the instruction throughput of this approach is .966, and given that this method has only a factor of $\log m$ excess computation over the serial algorithm, this means that theoretical efficiency is near 1. This also gives us our theoretical run time of $O(\frac{mn}{p} + \log m)$. This

	Run Time	Copy Time
CPU	23290	N/A
GPU	3042	217.7
GPU Shared	200.68	217.7
GPU Text.	107.38	2.361

Table 1: Run time in ms for Full Search Method template matching on a 512x512 image and a 64x64 template. Times are in ms.

texture method is fast when compared to Full Search Methods on the CPU, but performs a great deal of excess computation when compared to the best serial methods (i.e. accelerated methods), giving it a low efficiency $E = \frac{S}{p} = \frac{T_s}{pT_p}$, where T_s is serial execution time, and T_p is parallel execution time. In other words, it is not strictly necessary to compute E_i at all locations. Our algorithm attempts to address this fact, while maintaining efficient parallel execution.

The third column in Table 1 represents the amount of time required to copy the image data from the host to the GPU under these various approaches. As can be seen, the copy time of this step cannot be ignored. We further explore

[1] Noting again that $p = 128$, this would appear to be super-linear, especially considering that the clock speed of the GPU is considerably slower than that of the CPU. This is likely due to CPU cache-miss issues.

this issue in Table 2, where we compare the memory allocation and copy times for varying sizes of data. We conclude from this that it may be beneficial to perform some tasks serially on the host if they can reduce the amount of data that must be transferred to the GPU.

size	malloc	copy	malloc 2D	copy 2D
$4 * 10^3$	0.067567	0.005253	0.116700	0.014929
$4 * 10^5$	0.118616	0.291486	0.122187	0.296680
$4 * 10^6$	0.141160	2.576290	0.180513	2.713126
$4 * 10^7$	0.241793	23.344471	0.629537	24.801236

Table 2: Average results over 1000 trials of basic CUDA memory operations. "malloc" and "malloc 2D" refer to allocating an array and a byte aligned 2 dimensional array on the GPU, respectively. "copy" and "copy 2D" refer to copying data from the CPU's global memory to the GPU's global memory into the respective data structures. The first column refers to the amount of data used for that experiment, in bytes. All times are in ms.

3 GPU Acceleration Method

In designing the algorithm in Figure 4, we wanted to off-load as much of the computation that could be conducted in parallel onto the GPU as possible, while still minimizing the amount of memory transfer that had to be done. In addition, we wanted to minimize the total work done by the algorithm, to reduce the level of excess computation as compared to the best serial algorithms. Lastly, but with equal importance, we needed to use data parallel design methodologies in the algorithm.

The unique points of our algorithm when compared to the Full Search Method are a) the combination of the upper bounds of [21] with the very fast bounding methods of [12], and more importantly b) the division of steps between the CPU and GPU such that the CPU deals with the largest amount of memory, and the largest number of subwindows, while also doing as little real computation as possible, leaving the GPU to do extensive computation on only a minimal number of subwindows. The second point has the combined effect of minimizing memory transfer and excess computation.

Essentially, the algorithm begins by performing an initial scan of the data on the CPU, performing approximately 5 operations per subwindow to find initial upper and lower bounds on the match value of each location in the image using the base case of Fig. 3, as explained in Sectin 1.1. The image-strips (or masking vectors) were chosen in particular because they reduce the amount of excess computation over other bounding methods used in template matching, i.e. [21, 20, 17]. Every time the bounds of y_i are updated, the computed values can be reused directly for computing E_i. Reduction of excess computation is especially important in GPU programming, as it is replicated over each processor.

```
ParallelTemplateMatch(x, Y)
 1  InitBounds(Y, x)
 2  E_guess, y_best ← FindBestInitMatch(Y, x)
 3  Y ← Prune(Y, E_guess)
    ▷ From here onwards, the code is executed on the GPU by many
    ▷ threads in parallel.
 4  while |Y| > 1
 5        do
              ▷ Tighten the bounds on the remaining members of Y.
 6            i ← ThreadID
 7            UpdateBounds(y_i, x)
 8            if l_i < E_guess
 9               then
10                     if u_i < E_guess
11                        then l_i, u_i ← ComputeE(y_i, x)
12                             E_guess ← u_i
13                             y_best ← y_i
14               else  break
15  if E_i < E_guess
16     then E_guess ← E_i, y_best ← y_i
17  return y_best, E_guess
```

Fig. 4: The main method of our GPU based template matching algorithm.

The next step is a single run of the "Prune" method (see Figure 5) on the CPU before beginning the run of the algorithm on the GPU. The Prune step reduces execution time because it drastically reduces the number of locations that the GPU must consider (and therefore the amount of data transfer from host to GPU), often by 99% or more. Yet this step does only a very small fraction of the overall work of the algorithm (on the order of a single comparison operation per y_i). Experimentation has shown, however, that as image noise levels increase, fewer candidates are pruned, resulting in more calculations to be done, which requires the remaining calculations to be done on the GPU versus the CPU.

```
FindBestInitMatch(Y, x)
1  l_min, y_min
2  for y_i ∈ Y
3        do
4           if l_i < l_min
5              then l_min ← l_i
6                   y_min ← y_i
7  l_min, u_min ← ComputeE(y_min, x)
8  return l_min, y_min

Prune(Y, E_guess)
1  for y_i ∈ Y
2        do
3           if l_i > E_guess
4              then Y ← {Y − y_i}
5  return Y
```

Fig. 5: The relevant subroutines called by our main method.

Some of these initial steps could benefit from parallel execution, except that in our experiments the cost of transferring the full image meta-data from host to GPU memory more than cancels the benefits. These steps could, however, be implemented to run on a multicore CPU and one should expect to see a significant increase in speed. The pruning method is examined in more depth in Figure 5. All steps after this point take place on the GPU.

We chose to transfer to GPU at this point because the workload increases dramatically here, as the algorithm begins comparing pixel values directly to tighten the bounds on the individual y_i. The pixel values of the y_i are held in texture memory as opposed to shared memory, as are those of the template, since they are not modified during the run of the algorithm. This allows for a great increase in access and copy speeds. Furthermore, very little data is actually shared between concurrent threads at run time. This, combined with the very

limited size of the shared memory, led to our decision to only use it to store pointers to the candidates. The upper and lower bounds of the candidates are held in global memory initially, but since we have chosen a one-to-one candidate-to-thread mapping, each thread copies the bounds to local memory (registers) and performs their calculations there, avoiding costly global memory access. With the CUDA architecture, threads are organized into blocks that can be of one, two or three dimensions in geometry. These blocks are then organized into grids that can likewise be one, two, or three dimesions. Our grids of thread blocks are two-dimensional grids consisting of three-dimensional thread blocks. Experimentally we did not notice any significant difference in performance due to differences in grid and thread block geometries. The sizes of our grids and blocks were determined based upon the size of the input data. Although branching is typically avoided in SIMD programming, we stop those threads whose candidates are no longer possible matches (that is, $l_i > E_{guess}$). These threads wait at a synchronization barrier, allowing the multiprocessor to allocate more time to the threads that still contain potential matches. Each thread then compares its current distance value against a global minimum to allow for a degree of synchronization between multiprocessors.

The combination of these steps to reduce the memory footprint, memory copy time, and execution workload on the GPU result in our algorithm's accelerated performance. This design is scalable and not hardware specific, and can be ported to any CUDA GPU with similar results.

4 Results

Our experimental design consisted of averaging the results of running our algorithm over a number of trials with a variety of images of different sizes and resolutions. We first tested with a few standard images ("pentagon" at 512x512, "airport" at 512x512, and "man" at 1024x1024), and then considered a few images captured on a modern digital camera. We extracted a template from each and tested with noise levels ranging from noiseless to very noisy ($\sigma = 70$).

We then ran the Full Search Method (using textures as described above) for the same number of trials on the same GPU using the same input.

Our experimentation yielded the following performance results: When comparing the performance of our algorithm to the Full Search Method on small images (512 x 512) at zero to low noise levels, our algorithm has better performance than the Full Search Method. However, as the amount of noise increases to extreme levels, our algorithm begins to slow down, while the Full Search Method remains unchanged. This is due to the fact that at high noise levels, the Prune step executed on the CPU eliminates fewer candidates and effectively becomes excess computation or overhead instead of contributing efficiently to returning a result. The results for these experiments run with the pentagon and airport images are shown in Figure 6, where we report on the speedup factor compared to increasing noise levels.

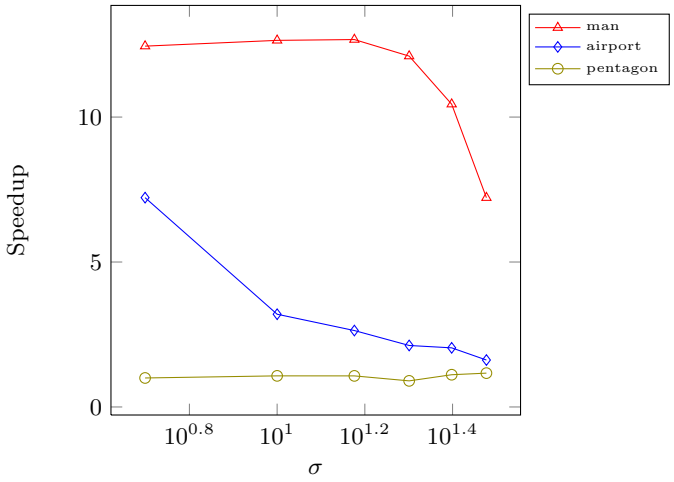

Fig. 6: Ratio of speedup versus noise level σ of our algorithm for different images of different sizes.

When comparing our algorithm's performance to that of the Full Search Method on medium to large images one can see the tremendous performance increase of our algorithm. With an image size of 1024x1024 and a template size of 128x128, our algorithm experiences a 12 times performance increase over the Full Search Method. The comparison of the performance increase for the 1024x1024 man image to the performance increase of the smaller 512x512 images can also be seen in Figure 6. Furthermore, with a noiseless image size of 2306x1535 and a template size of 304x280, our algorithm performed 7 times faster, and nearly 39 times faster with a noiseless 3072x2304 image and a template size of 584x782. The results for the runnning times on these large images are summarized in Table 3.

Image	Noise	Parallel	Full Search	Improvement
second	0	3979.879	27930.175	7.018
rob ref	0	6123.839	237215.515	38.736

Table 3: The images "second" and "rob ref" were taken with a modern digital camera, and are of size 2306x1535 and 3072x2304 respectively. These larger images allow for comparatively large improvements in run time. The run times for the Parallel and Full Search algorithm implementations are expressed in milliseconds.

4.1 Analysis

The worst case run time of the algorithm is actually no better than the naive algorithm described in Section 2. In practice, however, the expected run time of the algorithm is significantly lower than this. This is not uncommon in accelerated template matching techniques, GPU or host based [12, 20, 21].

Along similar lines, the fact that our algorithm does not make use of a great deal of the GPU during the final stages of its runtime to avoid excess comptuation means that the instruction throughput is actually quite low (around .05). This has positive and negative consequences. The obvious negative is that much of the GPU is idle, and current GPUs do not allow multiple host threads to use the GPU simultaneously. The positive consequence is that it means the algorithm is very efficient, and since next generation GPU architectures do allow multiple host threads to use the GPU simultaneously [23, 24], our algorithm will leave more of the GPU open to other threads. This would be advantageous in machine vision settings where template matching is used as a low level algorithm since it would "leave room" on the GPU for higher level processes.

5 Conclusions and Future Work

We have shown here that while adapting existing algorithms to run on GPUs can provide considerable increases in performance, an algorithm that is designed specifically to run on a GPU can have a nearly 39 times performance increase over algorithms that are simply adapted to run on GPUs. We have shown that in addition to considerations of data parallel algorithm design and analysis, one must also carefully consider the unique memory structure and transfer costs of GPUs to fully harness their power. That power is increasing, with CPU and GPU manufacturers preparing to release next generation GPU architectures, which will include features such as C++ support, error correcting memory, double precision support, and a chip-wide high-speed communication [23, 24].

The work done here could very well be extended to multimedia database search, as our algorithm's ability to eliminate many candidates before calling the GPU would allow searching a very large database without overwhelming the GPU's limited memory. Additionally, using a clever memory copy algorithm, one could adapt this algorithm to search extremely large images, such as those generated by astronomical surveys, by loading only image regions representing likely matches onto the GPU.

References

1. Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., Phillips, J.: GPU computing. Proceedings of the IEEE **96**(5) (2008) 879–899
2. Hillis, W.D., Steele, J.G.L.: Data parallel algorithms. Commun. ACM **29**(12) (1986) 1170–1183
3. Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to Parallel Computing. 2nd edn. Addison-Wesley Longman Publishing Co., Inc. (2002)

4. NVIDIA Corp.: NVIDIA CUDA programming guide v2.3.1 (August 2009)
5. AMD Inc.: ATI stream computing user guide rev1.4.0a (April 2009)
6. Jin, Z., Lou, Z., Yang, J., Sun, Q.: Face detection using template matching and skin-color information. Neurocomput. **70**(4-6) (2007) 794–800
7. Brunelli, R., Poggio, T.: Face recognition: features versus templates. Pattern Analysis and Machine Intelligence, IEEE Transactions on **15**(10) (1993) 1042–1052
8. Efros, A.A., Freeman, W.T.: Image quilting for texture synthesis and transfer. In: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, ACM (2001) 341–346
9. Luczak, T., Szpankowski, W.: A suboptimal lossy data compression based on approximate pattern matching. Information Theory, IEEE Transactions on **43**(5) (1997) 1439–1451
10. Rodrigues, N., da Silva, E., de Carvalho, M., de Faria, S., da Silva, V.: On dictionary adaptation for recurrent pattern image coding. Image Processing, IEEE Transactions on **17**(9) (2008) 1640–1653
11. Li, R., Zeng, B., Liou, M.: A new three-step search algorithm for block motion estimation. Circuits and Systems for Video Technology, IEEE Transactions on **4**(4) (1994) 438–442
12. Tombari, F., Mattoccia, S., Stefano, L.D.: Full-Search-Equivalent pattern matching with incremental dissimilarity approximations. Pattern Analysis and Machine Intelligence, IEEE Transactions on **31**(1) (2009) 129–141
13. Abate, A., Nappi, M., Ricciardi, S., Sabatino, G.: GPU accelerated 3D face registration / recognition. In: Advances in Biometrics. (2007) 938–947
14. Huang, J., Ponce, S.P., Park, S.I., Cao, Y., Quek, F.: GPU-accelerated computation for robust motion tracking using the CUDA framework. In: Visual Information Engineering, 2008. VIE 2008. 5th International Conference on. (2008) 437–442
15. Stefano, L.D., Mattoccia, S., Tombari, F.: Speeding-up NCC-based template matching using parallel multimedia instructions. In: Computer Architecture for Machine Perception, 2005. CAMP 2005. Proceedings. Seventh International Workshop on. (2005) 193–197
16. Massachusetts Institute of Technology: IAP09 CUDA@MIT 6.963 (2009)
17. Goshtasby, A., Gage, S.H., Bartholic, J.F.: A Two-Stage cross correlation approach to template matching. Pattern Analysis and Machine Intelligence, IEEE Transactions on **PAMI-6**(3) (1984) 374–378
18. Rosenfeld, A., Vanderburg, G.: Coarse-Fine template matching. Systems, Man and Cybernetics, IEEE Transactions on **7**(2) (1977) 104–107
19. Pele, O., Werman, M.: Robust Real-Time pattern matching using bayesian sequential hypothesis testing. Pattern Analysis and Machine Intelligence, IEEE Transactions on **30**(8) (2008) 1427–1443
20. Hel-Or, Y.: Real-time pattern matching using projection kernels. Pattern Analysis and Machine Intelligence, IEEE Transactions on **27**(9) (2005) 1430–1445
21. Schweitzer, H., Anderson, R.F., Deng, R.A.: A near optimal Acceptance-Rejection algorithm for exact Cross-Correlation search. In: Proceedings of the IEEE International Conference on Computer Vision, Kyoto, Japan (2009) Poster Session.
22. Murphy, M.: NVIDIA's experience with open64. In: Open64 Workshop at Intl. Symposium on Code Generation and Optimization (CGO), Boston, Massachusetts, United States (April 2008)
23. NVIDIA Corp.: NVIDIAs next generation CUDA compute architecture: Fermi. (September 2009)

24. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: a many-core x86 architecture for visual computing. ACM Trans. Graph. **27**(3) (2008) 1–15