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 to be parallel from the beginning can lead to substantial performance increases. In this paper we present an algorithm for template matching that is targeted 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 a multi-core architecture in mind. Its speedup factor is nearly linear in p, and its overall cost is not far from serial implementation.

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 2007 NVIDIA released one of the first GPU architectures for general purpose computing. This architecture, known as CUDA (Compute Unified Device Architecture), is a SIMD (single instruction multiple data) machine using non-uniform memory access patterns (NUMA) with a shared address space multiprocessor architecture. Figure 1 depicts the structure of the NVIDIA GeForce 8800 series as an example. The GeForce 8800 contains 16 multiprocessors, each containing 8 semi-independent cores for a total of 128 processing units. Each of the 16 multiprocessors typically handles much more than 8 threads simultaneously.

Template matching is a building block for many high level Computer Vision applications, such as face and object detection [2, 3], texture synthesis [4], image compression [5, 6], and video compression [7, 6]. Its run time is often infeasibly slow in raw form [8] and there has been much research into methods for accelerating template matching for various applications.

Previous work with vision algorithms and template matching on the GPU has typically involved adapting sequential algorithms to the data-parallel GPU architecture [9–12]. We have taken the approach of designing a template matching algorithm that was intended for data-parallel execution from the beginning to show the advantages of GPU specific design.

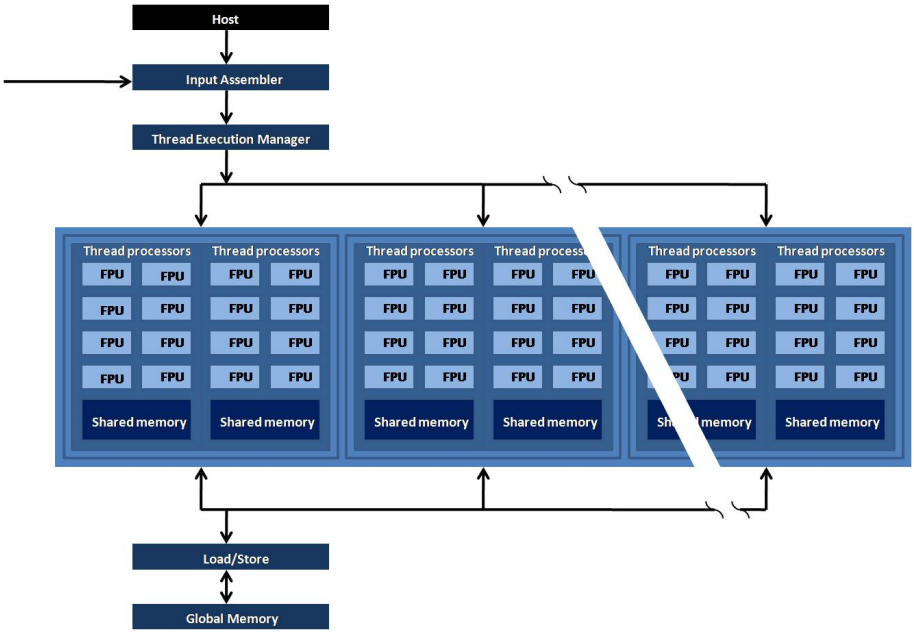

Fig. 1. NVIDIA GeForce 8800 Architecture

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. [13–15]. A second set which has emerged recently makes use of bounds on the error measure to achieve acceleration without sacrificing accuracy, though the choice in error measures is somewhat more limited [8, 16, 17]. Our proposed algorithm falls into that second set.

Throughout this paper we make use of the l_1 norm-based distance measure (i.e. the sum of absolute differences) between template and 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 all contain 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 $l_i \leq E_i \leq u_i$.

2 Full Search Method

We first consider the case of the Full Search Method of template matching, otherwise known as a brute force method. The traditional Full Search Method calculates E_i for all $y_i \in Y$, and returns $y_{opt} = \arg\min_{y_i} E_i$. Figure 2 represents this algorithm, as well as its adaptation to the GPU. Given that m is the num-

```
FullSearchCPU(Y, x)
1  E_opt, y_opt
2  for y_i ∈ Y
3       do E_i ← ComputeE(y_i, x)
4          if E_i < E_opt
5             then
6                  E_opt ← E_i
7                  y_opt ← y_i
8  return E_opt, y_opt
```

```
FullSearchGPU(Y, x)
1  E_opt, y_opt
2  i ← ThreadID
3       do E_i ← ComputeE(y_i, x)
4          E_i ← reduce(E)
5          if ThreadID = 0
6          E_opt ← E_i
7          y_opt ← y_i
8  return E_opt, y_opt
```

Fig. 2. Full search methods on CPU and GPU

ber 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 [12]. The straightforward GPU implementation should run in $O(\frac{mn}{p} + \log m)$ time, where p is the number of processors, assuming that $1 \ll n$. Table 1 represents the results of implementing both versions in various forms. The ratio of run times of the naive full search implementation on GPU to serial implementation (or "speedup", S) is only 7.14. Given that the number of processing units p is 128, this is clearly not a cost optimal implementation, as it yields an efficiency of .056 (from $E = \frac{S}{p}$). The majority of this is due to communication overhead, since main memory on the GPU is uncached, and experimentation confirms that instruction throughput (the average number of instructions executed per core per clock cycle) is only .034.

In the version making use of texture memory (GPU Text.) $S = 212.23$. 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. However, the instruction throughput of this approach is .966, and given that this method has almost

zero excess computation over the serial algorithm, this means that theoretical efficiency is near 1. This also gives us our theoretical run time of $O(\frac{mn}{p} + \log m)$. The texture method is fast and efficient when compared to full search methods

	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 template matching on a 512x512 image and a 64x64 template. *GPU* is the basic form of the full search algorithm as shown in fig 2. *GPU Shared* makes use of shared memory to cache the template, and *GPU Text.* stores all image and template information in the on card texture memory.

on the CPU, but performs a great deal of excess computation when compared to the best serial methods (i.e. accelerated methods). Our algorithm attempts to address that fact. Additionally, we make use of methods specific to parallel architectures to further improve run time.

3 Design Considerations

Designing an algorithm from the beginning to be implemented on a current parallel processing architecture requires special consideration due to issues of overhead and the importance of careful design of memory access. Also, the CUDA architecture has certain drawbacks due to its current design that greatly influence how CUDA programmers design their code. For example, each of the 16 multiprocessors contains 16kB of very fast, purely local, user-managed cache shared between its 8 cores. Its size is considerably limited, and it suffers from "bank-conflicts", wherein multiple requests to neighboring addresses result in serial access. Bank conflicts can be avoided if all requests are for the same address or if sequential threads request sequential addresses. This was the primary limiting factor in the performance of the "GPU Shared" approach in Table 1. A fast, automatically cached "texture memory" also exists, but it is read-only. Since so many threads may be active at once, threads are organized into "blocks" of 512. Threads within a block can communicate with one another, and synchronize with one another, but are unable to do so (directly) with threads outside of that block. Furthermore, transfer of data between cores in the same multiprocessor is fast, but between multiprocessors it requires the use of slower global memory. Therefore, one must carefully consider how the problem is broken down into blocks.

Finally, and perhaps most importantly, due to the considerable cost in time of transferring data from host memory to device memory one has to decide what data to load from main memory to device memory as well as when and how to

minimize these transfers as much as possible. The cost of allocating memory on the card and copying data from the host to the card is explored in Table 2.

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 in the respective data structures. The first column refers to the amount of data used for that experiment, in bytes. All times are in ms.

4 GPU Acceleration Method

In designing the algorithm in Figure 3, 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 levels of excess computation as compared to the best serial algorithms. Consequently, lines 1 through 3 of the pseudocode are conducted on the CPU, while lines 5 through 13 are conducted on the GPU, as they are suited to the strengths of the different architectures. The unique points of this algorithm are a) the combination of the upper bounds of [17] with the very fast bounding methods of [8], 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 few 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 around 5 operations per subwindow to find initial upper and lower bounds on the match value of each location in the image. This is analogous to the initialization step used in [8], and we make use of their "image strips" to provide the initial bounding. The concept of using upper bounds in template matching was first proposed in [17]. The upper and lower bounds are initialized by setting $l_i = ||x| - |y_i||$, and setting $u_i = ||x| + |y_i||$. These bounds are updated according to the procedures in [8, 17], shown in brief in Figure 5. The image-strips were chosen in particular because they reduce the amount of

```
ParallelTemplateMatch(x, Y)
    ▷ We first need to initialize the lower and ▷ upper bounds for each y_i ∈ Y.
 1  InitBounds(Y, x)
    ▷ Using those bounds we make a guess at the best match value.
 2  E_guess, y_best ← FindBestInitMatch(Y, x)
    ▷ We remove everything in Y whose bounds prohibit it from
    ▷ being a best match.
 3  Y ← Prune(Y, E_guess)
    ▷ At this point Y is typically very small.
    ▷ 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. 3. The main method of our GPU based template matching algorithm.

```
InitBounds(Y, x)
1  for y_i ∈ Y
2        do u_i ← ||x| + |y_i||
3           l_i ← ||x| − |y_i||

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

ComputeE(y_i, x)
1  E_i ← |∑_{a,b}(x(a, b) − y_i(a, b))|
2  return E_i
```

Fig. 4. The relevant subroutines called by our main method.

excess computation over other bounding methods used in template matching. 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 an important point in parallel algorithm scalability.

The next step is a single run of the "Prune" method 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 (often by 99% or more), while doing only a very small fraction of the overall work of the algorithm (on the order of a single comparison operation per y_i).

Some of these initial steps could benefit from parallel execution, except that the cost of memory transfer to the GPU outweighs the benefits. These steps could, however, be implemented to run on a multicore CPU and see important increases in speed. These steps are examined more in depth in Figure 4. 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 are those of the template, since they are not modified during the run. This allows for a great increase in access and copy speeds. 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 works on them there, avoiding costly global memory access. 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 point, 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 not hardware specific, and can be ported to any CUDA GPU with similar results.

5 Results

After many experiments with data storage, transfer, and manipulation, we have developed a very fast and efficient implementation of our algorithm, which takes full advantage of the CUDA architecture on our GPU. Our experimental design consisted of averaging the results of running our algorithm a number of trials with a variety of images of different sizes and resolutions. We first tested with a few standard images (lenna at 512x512, as well as pentagon, airport, 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 ranging from noiseless to very noisy ($\sigma = 70$). We then ran the Full Search Method (using

```
UpdateBounds(y_i, x)
1  if l_i ≠ u_i
2     then
3          diff ← |y_{i_a} − x_a|
4          l_i ← diff +||y_i − y_{i_a}| − |x − x_a||
5          u_i ← diff +||y_i − y_{i_a}| + |x − x_a||
```

Fig. 5. The procedure for tightening the bounds in our algorithm. What happens in UpdateBounds is dependent on error measure. This example uses the L_1 norm. One way to visualize a is to imagine masking off the lower $\frac{n-1}{n}$ of y_i and x, and calling the remaining portions b. Each time we call this function, we would unmask another $\frac{1}{n}$ of y_i and add it to a.

textures, as described above) for the same number of trials on the same GPU using the same input. We compare and report the average run time for both methods in milliseconds.

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 nearly the same performance as 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 extensively to the answer. However, 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. Furthermore, with an image size of 2306x1535 and a template size of 304x280, our algorithm performed 7 times faster, and 38 times faster with a 3072x2304 image and a template size of 584x782. Figure 6 and Table 3 summarize these results.

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.

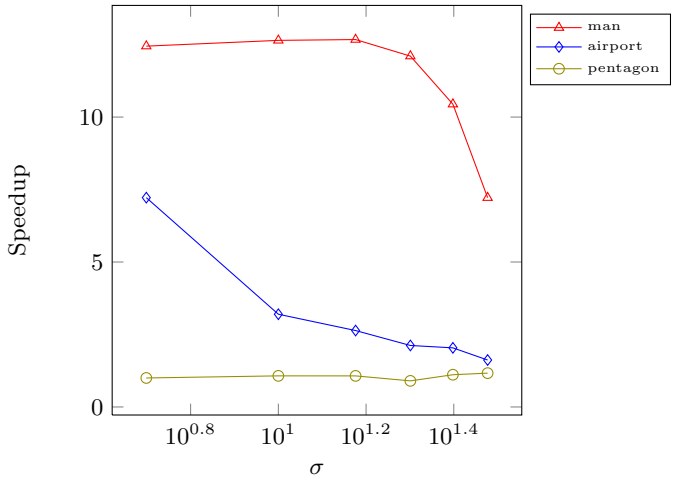

Fig. 6. Ratio of run time of Full Search on the GPU to the run time of our algorithm.

6 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 40 times performance increase over algorithms that are simply adapted to run on GPUs. Therefore, a paradigm that involves designing algorithms from the onset to run on GPU architectures must be adopted to take full advantage of all that this technology has to offer. Our design is extendable to other GPU architectures where it would offer similar improvements.

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. Finally, NVIDIA has announced a new GPU architecture, codenamed "Fermi", which will include C++ support, error correcting memory, double precision support, and a chip-wide L2 cache [18]. This will allow even greater performance gains due to the flexibility of the memory model.

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. Jin, Z., Lou, Z., Yang, J., Sun, Q.: Face detection using template matching and skin-color information. Neurocomput. **70**(4-6) (2007) 794–800

3. Brunelli, R., Poggio, T.: Face recognition: features versus templates. Pattern Analysis and Machine Intelligence, IEEE Transactions on **15**(10) (1993) 1042–1052
4. 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
5. Luczak, T., Szpankowski, W.: A suboptimal lossy data compression based on approximate pattern matching. Information Theory, IEEE Transactions on **43**(5) (1997) 1439–1451
6. 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
7. 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
8. 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
9. Abate, A., Nappi, M., Ricciardi, S., Sabatino, G.: GPU accelerated 3D face registration / recognition. In: Advances in Biometrics. (2007) 938–947
10. 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
11. 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
12. : IAP09 CUDA@MIT 6.963 (2009)
13. 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
14. Rosenfeld, A., Vanderburg, G.: Coarse-Fine template matching. Systems, Man and Cybernetics, IEEE Transactions on **7**(2) (1977) 104–107
15. 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
16. Hel-Or, Y.: Real-time pattern matching using projection kernels. Pattern Analysis and Machine Intelligence, IEEE Transactions on **27**(9) (2005) 1430–1445
17. Schweitzer, H., Anderson, R.F., Deng, R.A.: A near optimal Acceptance-Rejection algorithm for exact Cross-Correlation search. In: Proceedings of the International Conference on Computer Vision, Kyoto, Japan (2009) Poster Session.
18. : NVIDIAs next generation CUDA compute architecture: Fermi. (September 2009)