# Real-Time Dense DAISY on GPU[1]

Ioannis Panousis, Matthew Brown

Department of Computer Science
University of Bath
Bath, United Kingdom
{ip223,m.brown}@bath.ac.uk

## Abstract

In the field of computer vision, the dense extraction of reliable image features for every pixel has proven very computationally expensive. High level applications that need feature matches across images have to compromise with sparse correspondence, perhaps a few hundred keypoints per frame, at a high cost of robustness and accuracy. This project makes a coupling of recent appearances in vision and GPU programming in order to speed up this vital operation of dense correspondence and make it reach real-time speeds. The first is a fast and reliable descriptor algorithm called DAISY, and the second is the recent standard by the Khronos Group called OpenCL, for generic processor and platform programming. The DAISY algorithm is found highly suited to speed enhancements on massively parallel hardware and this is realised by implementing it in OpenCL. In this project we analyse the nature and key features of the DAISY algorithm to decompose it into a parallel design. We then proceed to implement it on the GPU. We achieve dense DAISY extraction with 188 FPS with 320x240 resolutions and 24 FPS for 1024x768 on modern hardware.

## 1 Introduction

A problem that has been at the forefront of vision systems is image correspondence. Correspondence means finding matches between similar features across images. An example is shown in Figure 1, which poses the high-level application of object recognition, one of the main uses of image correspondence. *Dense correspondence* on the other hand means performing this for all the pixels from one image to another, or a great portion of those.

Current popular algorithms for solving the problem of correspondence are forced to downgrade their quality of performance by reducing the granularity in their correspondence. This is called a sparse computation, where only a subset of the total number of pixels are utilised. This is commonly done to reduce the computational cost of the operation, namely in extracting the image features and performing the matching between two images.

However, many applications nowadays have a naturally dense requirement, or others can greatly benefit from the richer information such as dense optical flow or reconstruction which is often seen as a post-product of sparse correspondences [16] [10]. Some naturally dense applications include dense 3D reconstruction, structure-from-motion, object categorisation and more [8] [15] [2]. These require state-of-the-art quality of results and are being denied real-time capability due to the complexity of the underlying dense feature

[1] Technical report in progress, to be published in University of Bath, UK. 2012.


Figure 1: *A feature correspondence example using a template image. These images were generated by DAISY descriptor matching, with the extraction done on GPU.*

extraction.

In attempt to contribute to this area, we wish to enable *highly real-time dense feature extraction* for high resolution cameras. Thus we first ask ourselves what feature to use in order to satisfy the high quality requirements of today's vision front. Related work on feature extraction and matching is quite broad. Some methods use intensity gradients and color to perform cross-correlation [5] [11], others use variants for robustness by incorporating multi-resolution wavelets and log-polar [1] [18]. Another set of methods, however, use local descriptors as features to find correspondences. Descriptors are vectors that contain much more information about a pixel as well as its surrounding local region. The revolutionary descriptor SIFT has proven its natural robustness in many areas, including image stitching [9] [19] [3] and bag-of-visual-words scene classification. A more recent descriptor, called DAISY [12] [13], provides a comparable case in quality and in speed of computation. The DAISY descriptor accelerates from SIFT, by consequence of design, by using Gaussian convolution operations where SIFT has to build histograms of of gradient norms, which makes optimising much easier when using the Gaussian property of separability.

The immediate implication of DAISY's speed is that *dense extraction* can be achieved, for every pixel in an image. In turn this can lead to accelerating a wide range of higher level vision applications including dense 3D reconstruction, object tracking in 3D, dense scene flow. The common trade-off in those systems is quality and/or quantity for speed. In contrast, DAISY can provide both quality and quantity, in number of features per image, with significantly faster computation.

There are a few techniques to counter the lower reliability resulting from this. One is to detect worthwhile features, called *keypoints*, in an image where pixels are labelled as reliable points to extract features from, such that they satisfy cross-image repeatability[2]. This indicates they contain enough variation in their local region that they can be con-

[2]Repeatability is the property that "good" keypoints possess which means they can be detected in more than one appearance of the object they belong to, i.e. a new frame and under new conditions.

fidently discriminated from other pixels. In this way a few points can be easily matched across images and, if required, a fully dense or at least finer correspondence to be estimated from these few match seeds. This will achieve a result of the same type as a dense correspondence but with less complexity of computation.

What makes DAISY fit well with our requirements is that it achieves comparable high quality results to SIFT by using the same concept of local image gradients at different smoothing scales. In contrast, the descriptor design and the extraction method used favour faster computation through a natural mapping to parallel computation.

However, as the 8-core C++ version runs at just 13 FPS for 320x240 images, it requires one more push to reach the next level of speed performance and the OpenCL language can make this happen.

# 2 GPGPU

General-purpose programming on graphics cards has been around for many years already. Developers have used it for data parallel tasks such as video encoding, compression, conversion, filtering, or less relative such as data modelling and analysis, GPUs have provided acceleration with the less obvious languages like the OpenGL Shader Language (GLSL) and other lower level shader languages before that. These are languages that have not been architectured to serve general-purpose applications and this has effectively hindered the development of this area of programming.

The language that has surfaced now in many area of research that require heavy computation is CUDA, for Nvidia GPUs, and has made GPGPU very popular. However, with the appearance of OpenCL it is possible to exploit any mainstream CPU or GPU device with a major OS.

## 2.1 OpenCL - Open Computing Language

Recently, with the first release in 2008, a new framework has been architectured and implemented to support the incredibly wide range of currently available CPU and GPU devices, along with the dominant operating systems of today. This is the Open Computing Language (OpenCL) standard [7], by the Khronos Group.

This is the truly generic form of a framework that will allow largely parallel applications to be accelerated regardless of their purpose, hardware and software they are running on. This is because virtually any modern commercial device and mainstream operating system conforms to the standard. Among graphics card developers, AMD support OpenCL 1.1 and partially 1.2 features already, along with an extensive set of developer tools and guides for OpenCL optimisation. Nvidia have implemented upto OpenCL 1.1 and also have profiling support and guides for OpenCL programming. The OpenCL standard is under continued development by the Khronos Group, with many library developers integrating OpenCL support as an 'easy' way to accelerate computations.

# 3 Massively Parallelising DAISY

The DAISY descriptor local structure can be visually shown as on the left side of Figure 4. It was developed by Engin et al., and presented within [12] and [13], with the objective of a highly discriminative feature that can be used in wide and multi baseline matching. Its design therefore aims to be robust to high parallax due to perspective change, depth and illumination. This is achieved by the multi-scale smoothed data extracted in each feature's local region, and the fact that the descriptor uses gradients rather than raw intensities.

## 3.1 Decomposition

The extraction algorithm requires the following stages.

| A | B | C | D |
|---|---|---|---|
| Denoise | Gradients | Gaussians | DAISY Transposition |

The first step is to remove high frequency noise in the image and a narrow 2D Gaussian is used. Step B is to extract multi-orientation local image gradients for every pixel in the image.

$$\frac{dI^{+}}{do} \qquad ((I_o * G_1) * G_2) * G_3$$

Equation 1: *Gradients* Equation 2: *Gaussians*

After steps A and B, the data has dimensions *Gradients x Height x Width* and has to undergo a set of consecutive Gaussian filters to produce multi-scale sets of the image gradients. Equation 2 indicates the filter series, where $*$ denotes convolution. The use of Gaussians throughout the data generation stage ensure immediate parallel mapping to multi-core as well as many-hundred-core GPUs. This is part of the DAISY design that makes it susceptible to fast computation.

The computational complexity of computing multi-scale gradients is significantly reduced by Gaussian properties of separability and of consecutive Gaussians. Separability of a 2D filter into two 1D filters means a cost reduction of $\frac{W^2}{2W}$ in general. Consecutive Gaussians reduce complexity as the end-smoothing to sigma $S_3$ can be achieved by using a more narrow Gaussian of sigma $S_{2'} < S_3$ on the data previously smoothed to sigma $S_2$. This benefits DAISY computation especially as there is a large data volume to be processed, which is the multi-orientation gradients. This way DAISY achieves very rich descriptors at a very low computational cost.

The algorithm, following a parallel decomposition, is mapped onto the GPU model with the components and flow shown in Figure 2. Checks were done at the end of each component to determine if any task parallelism is possible, to overlap the work within a component. For instance, X and Y separable convolutions cannot be overlapped as their input/output requires them to be serialised. Additionally, the multiple consecutive Gaussian operations have inputs/outputs that are chain-connected, as can be drawn from

Equation 2, and prevent work overlap.

| Dimensions | G * H * W | *Uncoalesced* | |
|---|---|---|---|
| Size | 8 * 1024 * 768 | Bandwidth (GB/sec) | Time (msec) |
| SmoothingsNo | 3 | 6 | **46.875** |
| Separable | 2 | *Fully Coalesced* | |
| Read+Write | 2 | Bandwidth (GB/sec) | Time (msec) |
| Transfer (MB) | 288 | 96 | **2.929** |

Table 1: *Impact of uncoalesced multi-scale Gaussians with current data volume. The bandwidth shown is relevant to commodity GPUs.*

GPUs have immense raw computing power, the bottlenecks in performance are introduced by memory transactions. So, GPU memory possesses a vital capability of combining multiple memory accesses such that they will take the time of a single transaction only. This feature is known as *coalescence* and it is the unique strength that must be exploited for an efficient implementation. We show in Table 1 how important coalescence is within our design.

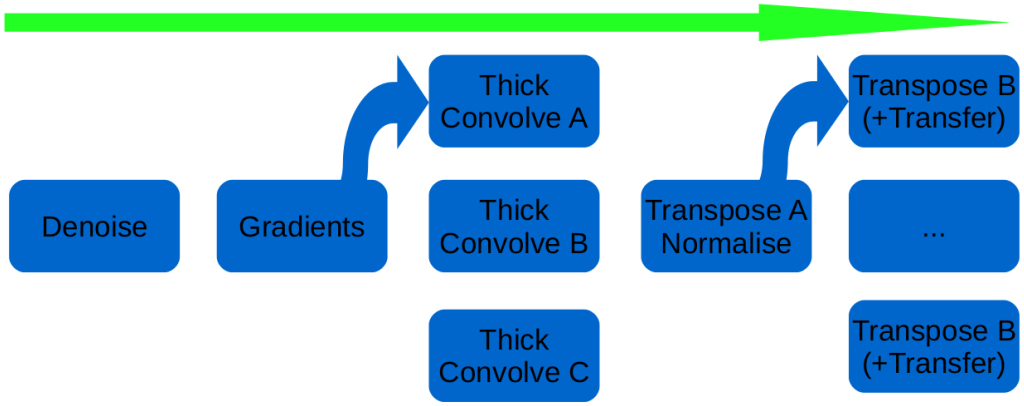


Figure 2: *Post-decomposition tasks and flow that is suitably fit to the GPU model.*

Table 1 shows that: 1) the default number of gradient orientations is equal to 8, 2) the thick convolution (i.e. of dimensions *GxHxW*) is performed the same number of times as the *SmoothingsNo*, 3) each of those convolutions are separable and therefore the full volume of the data is accessed twice, 4) each convolution consists of a read and a write of the data. With this the total onboard data transfer sums up to 288MB for a 1024 * 768 image. When at this degree of transfer, we can see that the impact of coalescence is huge, especially when in pursuit of a real-time implementation.

The memory access patterns that are used within the convolutions of this implementation fully comply with coalescence requirements on GPU memory that has width 256-bit. Therefore we exploit the full potential bandwidth, by using a pattern as shown in Figure 3. This visualises how a workgroup, which is a group of workers or threads on the GPU, will collaboratively read and write its collective input and output to memory. This means the workers must read consecutive memory addresses and write consecutively too. This is ensured in all vertical and horizontal convolutions.

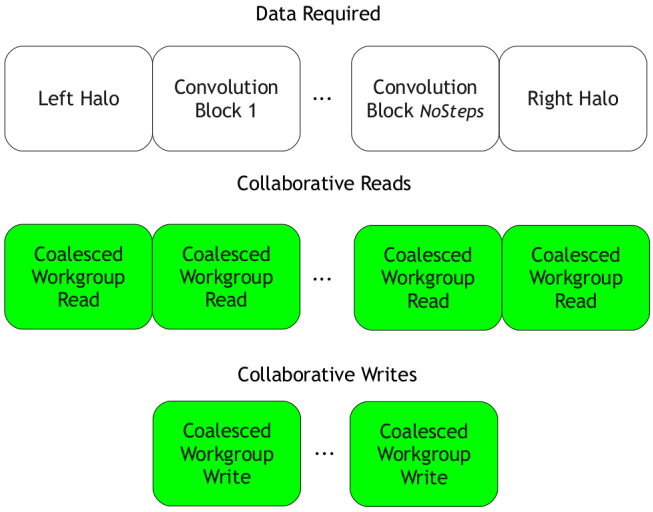


Figure 3: *Coalesced read and write transactions performed by the whole workgroup during the horizontal convolution phases. NoSteps shows that each workgroup may be assigned more than one output pixel.*

# DAISY Transposition

This part of the decomposition, of mapping the DAISY Transposition on the GPU, was vital to overall performance. The importance of coalescence has already been highlighted, and it becomes *SmoothingsNo (= 3)* times more crucial during this final operation. This is essentially due to the current data volume of *SmoothingsNo x GradientsNo x Height x Width*.

The structure of the descriptor, however, makes the design for coalescence very difficult. Think first on the final descriptor structure, as in Equation 3, where N corresponds to the number of positions shown in Figure 4, called *petals*. The left half of Figure 4 uses colour to indicate which smoothing scale the data will come from. The right half represents the memory order, as in Equation 3, of those data.

$$D = \{P_1 = \{g_1 ... g_{GradientsNo}\} ... P_N = \{g_1 ... g_{GradientsNo}\}\}$$

Equation 3: *Final DAISY structure in memory. Each $P_i$, which comes from a particular smoothing scale, consists of $GradientsNo$ gradients.*

There are two problems to solve in attempting to transpose the data from *SmoothingsNo x GradientsNo x Height x Width* into the *Height x Width x* $\{P_0 ... P_N\}$ dimensions of the descriptor. The problems relate to data locality. The first is that the final descriptors need the gradient orientations of a pixel at Y,X to be consecutive, as $g_1 ... g_{GradientsNo}$ shows. The second issue is that the circularity of DAISY means that a set of gradients, or a petal, say $P_i$ and $P_{i+1}$ need to be written consecutively in memory as well. However, each petal has an offset in *Height x Width* space which poses a problem of *read* coalescence.

In the interest of efficient memory access patterns, we decompose this complex operation

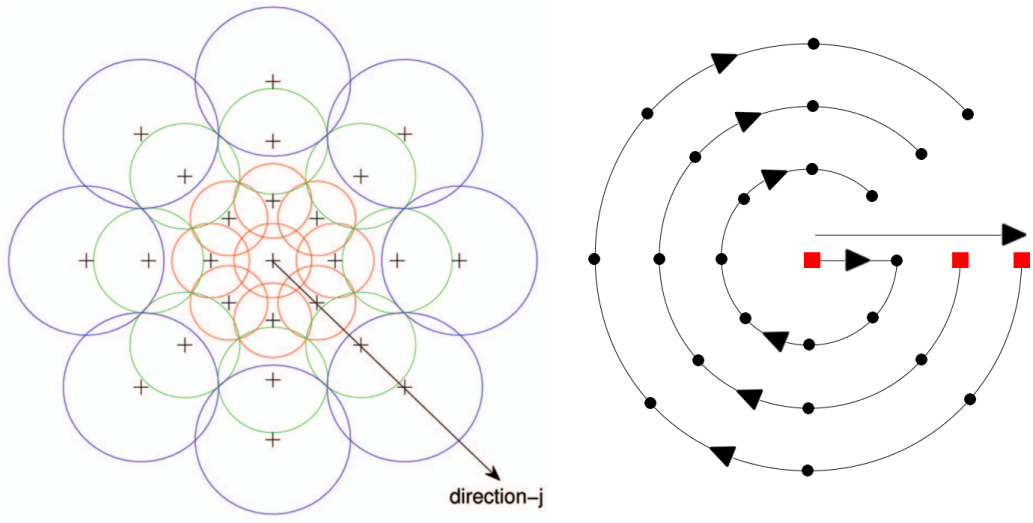


Figure 4: Left: *DAISY design for extraction of local pixel information.* Right: *Memory order for descriptor data, starting from central red square and outwards. It is critical to design memory access patterns to fit well with this order.*

into two lesser evils. They are called **Transpose A** and **Transpose B**, as in the task flow of Figure 2. **Transpose A** will ensure the data has gradient locality, such that the dimensions become *SmoothingsNo x Height x Width x GradientsNo*. This transposition is implemented very efficiently and in full coalescence, so it manipulates the large data volume with minimal effect on performance. This operation on the GPU is also exploited for the fact that gradient data for a pixel are stored in intermediate local memory. This allows for an L2 normalisation on the gradients, required for the DAISY algorithm, to be performed at this point, with little impact on performance.

**Transpose B** still holds the complexity of having to pair up petals around circular local regions in order to write them consecutively in the output. This requires the GPU workers to know locations to find consecutive petals in the source data array and the destination descriptor and petal number to write to. Due to the delicate pre-processing requires to generate these offsets, this cheap operation was done on the CPU. Then, lists of input petal offsets and their destinations were given to GPU workers. This way they would know exactly what data to manipulate without the high redundancy of pre-processing, which is best done on CPU in this case.

The visual interpretation of this operation is shown on the left half of Figure 7. The right half shows an even better re-design of the same operation, that was implemented later. With the first design, as shown in the figure, pairable petals within the red square grid would be used to target a destination descriptor shown as the single red square. The red grid was the data that was read in coalescence. The CPU lists generated were sets of those pairable petals inside that grid. The pairs could then be written consecutively in a target descriptor. The problem with this design, however, is that the pairable petals written were not a high enough percentage of total petals written (i.e. single + pairs of petals) and this harmed performance. It will be shown how the optimised design solves this.

## 3.2 Inter-operation performance

At the end of the zeroth optimisation cycle, the inter-operation performance is analysed. This serves to spot bottlenecks and will guide further improvements in design. Figure 5 shows that the complex transpose of Transpose B takes longest even when coalescence is utilised. It also shows that the transfer takes almost as long. Transfer of the results from GPU to RAM may be important if an application is interested on post-processing on CPU, although optimally operations would be continued on GPU.

On account of these results, optimisations are done for both expensive operations.

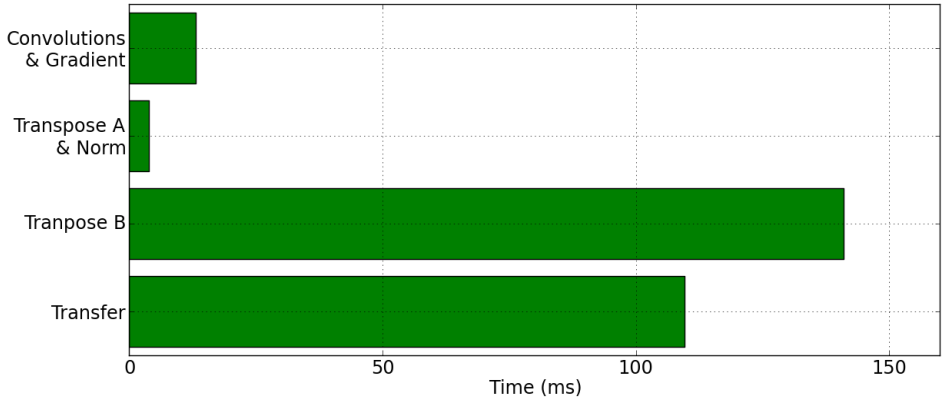


Figure 5: *The distribution of computation among the operations, at the end of the zeroth optimisation cycle.*

## 3.3 Optimisation 1: Compute/Transfer overlap

The first inter-operation analysis showed the transfer to RAM to have a substantial role in performance.

The best solutions to this were found to be; pinned memory buffers in RAM and compute/transfer overlap. Pinned memory buffers are memory blocks that are non-pageable and are more efficient to access that paged memory. These can be allocated on either the host RAM or the GPU device memory. In our case we are interested in writing back to RAM. The bandwidth difference between pageable and non-pageable memory in the tested hardware was as in Table 2.

| **Memory** | **Speed** |
|---|---|
| Pinned | 4803 MB/sec |
| Non-pinned | 2978 MB/sec |

Table 2: *Buffer copy speed to pinned and non-pinned memory.*

Vitally, GPUs also have the capability to perform host-device transfers while doing computations at the same time. OpenCL also has been designed from the start to allow developers to utilise this in many ways. In this case, we use asynchronous calls to

OpenCL transfer commands, while having previously enqueued computations. This requires chained synchronisation events that will ensure the correct order of execution of computations and transfers. The OpenCL design is flexible to allow this. With the computation of Transpose B and the transfer of the output data being done in blocks, it is possible to concurrently transfer while computing another block. Inherently, this speed-up of this optimisation scales well with increasing image size. Since, as the number of blocks increases so will the percentage of overlapped operations.

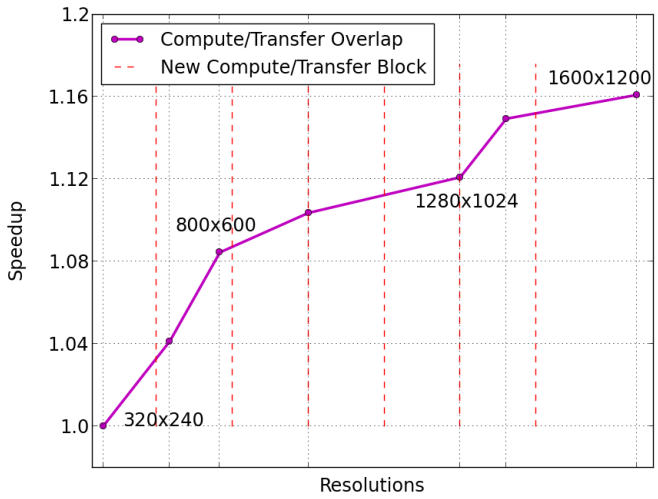


Figure 6: *Speed-up of overlapping the computation, on GPU, and transfer, to RAM, compared to no overlap. As expected, impact increases with the number of blocks.*

In reality, an application is better interested in post-processing DAISY onboard the GPU rather than returning the data to the host RAM. However, it can be understood that a GPU implementation may not be as straight-forward. So, when the user requires transfer back to RAM, the design is made efficient for that purpose.

## 3.4 Optimisation 2: Transpose B v2.0

Optimising the second transposition is higher priority than the transfers, since it will affect both cases of keeping or not keeping the output data on GPU. However, the task complexity made optimising less clear. After some further analysis, it is discovered that the initial design is not efficient enough in its memory transactions.

Of the data written to memory, only 41% are coalesced and the remainder costs the implementation dearly. Instead of coalescing such little output data, it is proven better to read 4 times as much input data and to achieve close to full coalescence in the output. This is calculated as in Table 3 before proceeding to a re-implementation, and proves itself true when tested on the GPU.

| | v1.0 | | v2.0 | |
|---|---|---|---|---|
| READ / WRITE | R | W | R | W |
| Units of data | 1 | 8.3 | 4.3 | 8.3 |
| Coalescence degree | 1 | 0.41 | 1 | 0.96 |
| Cost | 21.24 | | 12.92 | |

Table 3: *The versions differ in how much data is read and written. Reading more data in pays off if the writes are to be more coalesced.*

This formula used to derive the memory transaction cost was devised as part of this work. The cost is given in Equation 4. A unit of data of 1 is the full source data volume. Coalescence degree ranges in [0,1] and 1 indicates full coalescence.

$$Cost = \frac{U_R}{CD_R} + \frac{U_W}{CD_W}$$

Equation 4: *Coalescence cost, where U is units of data and CD is coalescence degree.*

The criteria for measuring the coalescence degree are by how much the kernel design conforms to byte alignment and number of aligned accesses. This can indeed by evaluated prior to the actual implementation of a kernel. The most recent graphics cards manage to coalesce non-aligned accesses that fall within a certain memory bank. The old kernel design would still be inefficient because it is not an issue of alignment, but an issue that there are *more* distinct write transactions for each worker. The new design reduces those transactions and combines them to be coalesced.

The way that this is done is by partitioning the work over the petals in the descriptor structure. Each descriptor has 25 petals, as seen in Figure 4. Hence, a dedicated kernel is launched on the GPU for each pair of petals in the descriptor. Since only 24 petals can be paired, 12 kernels are launched. One kernel, for instance, will be run to read and write the north and north-east petals as in the right half of Figure 7. Read coalescence can still be ensured for a workgroup by storing a range of petals in local memory. In the same figure as before, this is shown by having petals $A_i$ and $B_i$ where $i = 1..8$, though they need not be 8. Write coalescence is then ensured because the kernel has been launched to specifically write pairs of petals $A_iB_i$. So 24 out of 25 petals written will be coalesced, thus the 0.96 (24 / 25) coalescence degree in Table 3. The 0.04 stands for the one remainder petal.

The potential speed-up from implementing the new design, as calculated from the costs above, could be $\frac{Cost_{V1.0}}{Cost_{V2.0}} = 1.64$. The actual speed-up gained with the new design measures considerably higher for this particular operation. The overall speed-up of the DAISY extraction will depend on the previous operations as well.

| Image Size | v1.0 (msec) | v2.0 (msec) | Speed-up | Overall |
|---|---|---|---|---|
| 320x240 | 5.4 | 2.1 | 2.53 | 1.56 |
| 1024x768 | 41.7 | 18.7 | 2.22 | 1.43 |
| 2048x1536 | 170.0 | 67.3 | 2.52 | 1.59 |

Table 4: *Optimised Transpose B speed-up. Overall speed-up takes into account previous operations, transfers are omitted.*

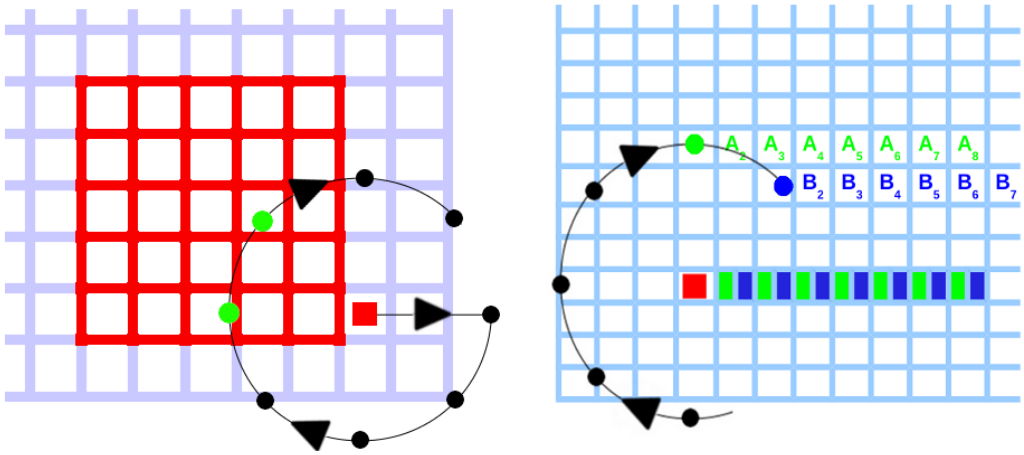


Figure 7: Left: *Pattern used by initial design. A square block is imported but not all petals are pairable.* Right: *Optimised design. Two rows of petals are imported and all will form pairs to be written along a row of descriptors (along row of the red square).*

The higher speed-up could be explained by the simpler and better design of the kernel. Partitioning the work into more kernels served to use less local memory per workgroup, leading to more concurrent workgroups. Also, instructions per worker are now fewer, which could have led to better work granularity and therefore more efficient use of GPU resources.

# 4 Benchmark with SIFT

For validation of the implementation, we used C versions of the same functions and thresholded the OpenCL and C numerical difference. In addition we compared the GPU version with the baseline CPU DAISY by running nearest neighbour descriptor matching between stereo pairs of images. For these tests we used the publically available Middlebury data. Last, GPU DAISY was validated against the ground truth for those stereo pairs.

The C-equivalent comparison fell within 10e-4 accuracy. The GPU-CPU comparison of best match disparity showed both versions to agree on the same matches for an average of 94% of the pixels in the stereo pairs, and the remainder were pixel differences of 1. The GPU and ground truth comparison gave error of 2-3%, which is a single pixel for the ground truth used. CPU DAISY performed similarly with the ground truth.

What was done next was in the interest of briefly demonstrating that DAISY can produce just as high quality results as the SIFT descriptor. SIFT is the most comparable descriptor to DAISY given the rich local information it exploits and therefore the testing of this work is best posed in this way.

As an example real case scenario, we integrated GPU DAISY into a sophisticated dense optical flow algorithm [8] that was developed for SIFT. The flow is found using Belief

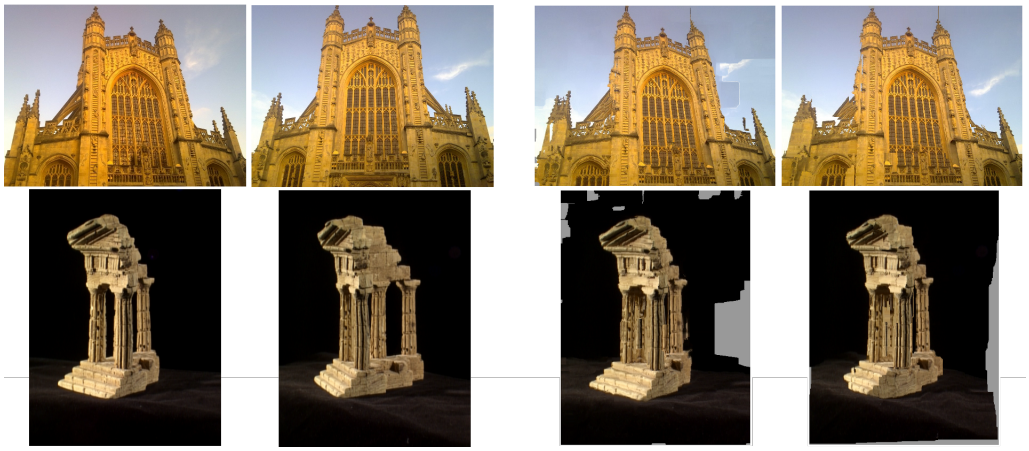

Figure 8: *Solving correspondence using Belief Propagation, with GPU DAISY outputs in the third column and SIFT outputs in the last. The dense correspondences are used to re-animate the reference frame (far left) using only the data of the target frame (centre left).*

Propagation. This testing was done for 10 pairs of images. The results were not statistically analysed as there was no ground truth, but the technique of warping the target frame to form the reference frame was used. There were no outstanding cases of mismatch among DAISY and SIFT and both delivered close to plausible images such as the example in Figure 8.

At this point it is shown that, at least for BP matching, DAISY works just as well as SIFT. Theory says it should work better due to the increased local texture that it utilises. Further quality testing that supports this can be found in [17] [21] [20], where DAISY is used in favour over SIFT for object recognition and dense stereo matching. Our work here is justified by the state-of-the-art quality of DAISY and so it was made densely real-time in order to contribute rich features to higher level applications with significantly less computation.

As a result of the GPU optimisation, the implementation described in section 3 runs highly real-time for standard camera resolutions. The hardware used was an NVIDIA GTX660 with 960 cores at 980MHz and 2GB of 192-bit onboard memory. Figure 9 shows what GPU DAISY is capable of. It can be used to aid high level applications that have regular or even fast frame rates while still reserving a large portion of the processing time per second for matching, recognition, motion segmentation etc.. Such a fast extraction of rich features can be vital for a wide range of applications.

Some may desire to transfer the results to RAM. For regular or high frame rate QVGA (320x240) video, the output descriptors could be returned to RAM and still leave a lot of room for further computation on CPU. Low frame rate VGA could also be returned. However, VGA and higher are best post-processed on the GPU. GPU programming is becoming rapidly more common and OpenCL is assisting this movement through a generic yet flexible framework.

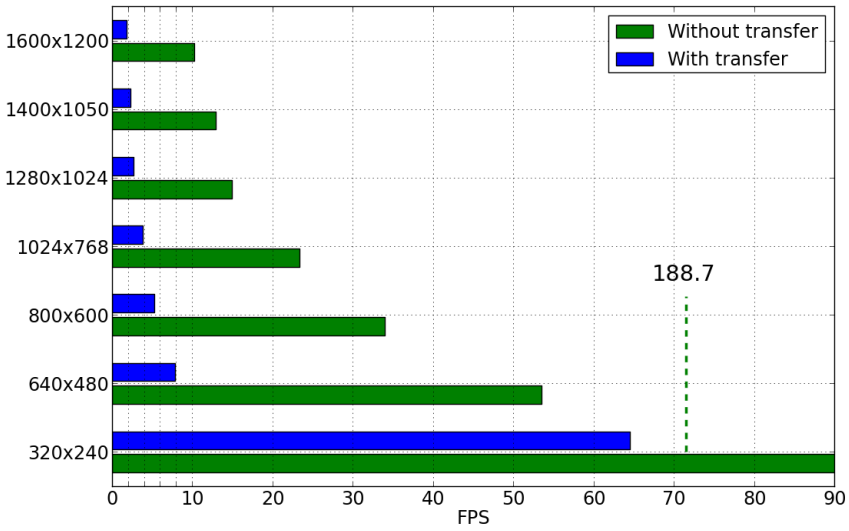


Figure 9: *Showing frame rate distribution for the standard camera resolution range.*

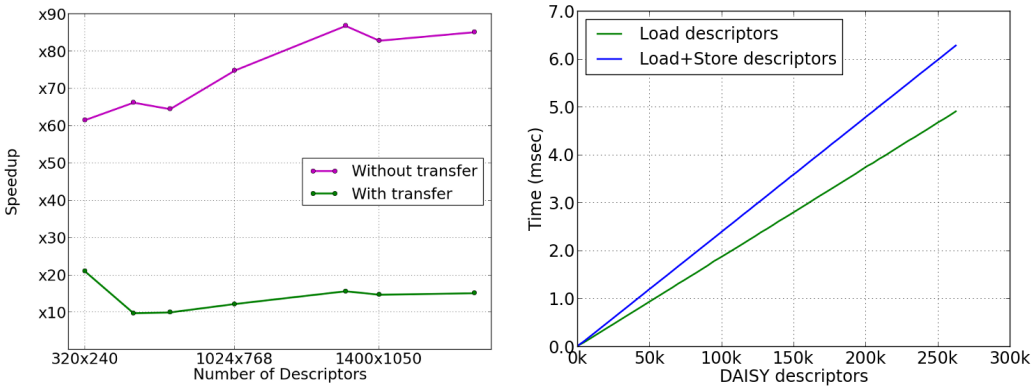


Figure 10: Left: *Speed-up of GPU DAISY over baseline CPU version on 1 core running at 2.53GHz.* Right: *Time it takes to move DAISY descriptors on the GPU, from global to local memory (load) and to write back to global (store).*

| Times (msec) | | |
|---|---|---|
| | 320x240 | 512x512 |
| SiftGPU no RAM | 91.9 | 345.6 |
| GPU DAISY to RAM | 15.5 | |
| GPU DAISY | 5.3 | 14.6 |

Table 5: *Dense descriptor extraction times on GPU with a CUDA version for SIFT and OpenCL for DAISY.*

Benchmarking speed was done, or attempted, against two GPU implementations of SIFT. Dense SIFT by [14] using CUDA is very slow and incomparable. The SiftGPU work of Wu [4] is faster. However, it has a very slow extraction time per descriptor and does not scale well densely. SiftGPU, also, could not be made comparable to our speeds after tuning CUDA block sizes and using the recommended optimal configuration for speed, or by subtracting the transfer time of SIFT descriptors from GPU to RAM. The best times achieved on the same hardware with CUDA enabled are in Table 5.

The slow SIFT time per descriptor might trigger questions on the cost of DAISY descriptor manipulation onboard the GPU. To clarify on this, post-processing a large number of DAISY descriptors on the GPU is not a problem. Measurements were taken and shown in the right half of Figure 10. A test kernel was written to import the descriptors into local memory, in one scenario, and, secondly, to write them back to global memory. These are mentioned as load and store operations. The times increase linearly with such a small slope that loading and storing over 250k DAISY descriptors takes less than 7ms. To store a descriptor would not be a common operation yet an application may want to normalise it in a custom way.

# 5 Conclusions

The driving aims of this work were to bring quality and quantity of **dense feature extraction** *closer to each other*, *closer to real-time computation* and *closer to a wide range of modern platforms*.

High quality features are brought by the DAISY descriptor and real-time computation is enabled by the GPU power, while OpenCL is a cross-platform and cross-device standard. Portability testing is still to be done, so far the code runs on two different NVIDIA cards running Linux. More extensive testing need be carried out to claim the implementation as portable, but the OpenCL code uses only standard built-in functions and no device or implementation-specific extensions. This will better ensure portability.

An important less tangible product of this work is a good understanding of programming in a GPU context. A few important steps to GPU development are to understand the architecture, prioritise the vital and less vital resources of the GPU model in terms of the application in question, and to proceed with a thorough algorithm decomposition tailored to the massively parallel GPU. This report brief ommits the extensive and constructive analysis detailed within the project dissertation. However, this valuable insight is taken forward and will aid in further GPU development.

The work has many potential extensions, some of which are; downsampling the data to greatly reduce volume size and boost speed performance, employing compression of the data through discretisation as in [17], real-time dense optical flow on the GPU for high

resolution video, etc.. We choose to progress in a new direction by attempting spatially and temporally dense real-time object recognition [6].

# References


[1] R. Anderson, N. Kingsbury, and J. Fauqueur. Coarse-level object recognition using interlevel products of complex wavelets. In *Image Processing, 2005. ICIP 2005. IEEE International Conference on*, volume 1, pages I–745. IEEE, 2005.

[2] G. Brostow, J. Shotton, J. Fauqueur, and R. Cipolla. Segmentation and recognition using structure from motion point clouds. *Computer Vision–ECCV 2008*, pages 44–57, 2008.

[3] M. Brown and D.G. Lowe. Automatic panoramic image stitching using invariant features. *International Journal of Computer Vision*, 74(1):59–73, 2007.

[4] Wu Changchang. Sift on gpu (siftgpu), December 2012.

[5] M. Goesele, B. Curless, and S.M. Seitz. Multi-view stereo revisited. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 2, pages 2402–2409. IEEE, 2006.

[6] Panousis Ioannis. Github - ipanousis, December 2012.

[7] Group Khronos. Opencl - the open standard for parallel programming of heterogeneous systems, December 2012.

[8] C. Liu, J. Yuen, A. Torralba, J. Sivic, and W. Freeman. Sift flow: Dense correspondence across different scenes. *Computer Vision–ECCV 2008*, pages 28–42, 2008.

[9] D.G. Lowe. Object recognition from local scale-invariant features. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[10] R.A. Newcombe and A.J. Davison. Live dense reconstruction with a single moving camera. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 1498–1505. IEEE, 2010.

[11] J. Pilet, C. Strecha, and P. Fua. Making background subtraction robust to sudden illumination changes. *Computer Vision–ECCV 2008*, pages 567–580, 2008.

[12] E. Tola, V. Lepetit, and P. Fua. A fast local descriptor for dense matching. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[13] E. Tola, V. Lepetit, and P. Fua. Daisy: An efficient dense descriptor applied to wide-baseline stereo. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(5):815–830, 2010.

[14] K. E. A. van de Sande, T. Gevers, and C. G. M. Snoek. Evaluating color descriptors for object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1582–1596, 2010.

[15] M. Vergauwen and L. Van Gool. Web-based 3d reconstruction service. *Machine vision and applications*, 17(6):411–426, 2006.

[16] A. Wedel, C. Rabe, T. Vaudrey, T. Brox, U. Franke, and D. Cremers. Efficient dense scene flow from sparse or dense stereo data. *Computer Vision–ECCV 2008*, pages 739–751, 2008.

[17] S. Winder, G. Hua, and M. Brown. Picking the best daisy. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 178–185. IEEE, 2009.

[18] G. Wolberg and S. Zokai. Robust image registration using log-polar transform. In *Image Processing, 2000. Proceedings. 2000 International Conference on*, volume 1, pages 493–496. IEEE, 2000.

[19] Jun Yang, Yu-Gang Jiang, Alexander G. Hauptmann, and Chong-Wah Ngo. Evaluating bag-of-visual-words representations in scene classification. In *Proceedings of the international workshop on Workshop on multimedia information retrieval*, MIR '07, pages 197–206, New York, NY, USA, 2007. ACM.

[20] G. Zhao, L. Chen, and G. Chen. A speeded-up local descriptor for dense stereo matching. In *Image Processing (ICIP), 2009 16th IEEE International Conference on*, pages 2101–2104. IEEE, 2009.

[21] C. Zhu, C.E. Bichot, and L. Chen. Visual object recognition using daisy descriptor. In *Multimedia and Expo (ICME), 2011 IEEE International Conference on*, pages 1–6. IEEE, 2011.