UCL DEPARTMENT OF
COMPUTER SCIENCE

Research Note

RN/14/02

Genetically Improved CUDA kernels for StereoCamera

20 February 2014

W. B. Langdon and M. Harman

Abstract

Genetic Programming (GP) may dramatically increase the performance of software written by domain experts. GP and autotuning are used to optimise and refactor legacy GPGPU C code for modern parallel graphics hardware and software. Speed ups of more than six times on recent nVidia GPU cards are reported compared to the original kernel on the same hardware.[1]

Keywords: GI, GP, gismoe, SBSE, software optimisation, nVidia, GPU, GPGPU, Tesla, GeForce GTX 580, evolutionary programming, software engineering

[1]To be published in part in EuroGP 2014 as "Genetically Improved CUDA C++ Software" [Langdon and Harman, 2014]. Technical Report RN/14/02 includes text, figures, etc., which were omitted from the LNCS version, partly to document the evolved kernels and the released code `ftp.cs.ucl.ac.uk` file `genetic/gp-code/StereoCamera_1_1.tar.gz` and `StereoCamera_v1_1c.zip`, which replaces `StereoCamera v1.0b` for CUDA 5.0 and later.

1 Introduction

Genetic Programming (GP) [Poli *et al.*, 2008] is increasingly being used in Software Engineering [Harman *et al.*, 2013]. We are using GP to make software more adaptable [Harman *et al.*, 2012] and are particularly interested in GP to generate code for bug fixing and for improving existing code. With increasing use of embedded and mobile devices there is a growing need to cheaply generate software which meets multiple interacting performance constraints, such as memory limits, energy consumption and real-time response [Tiwari *et al.*, 1994; White *et al.*, 2011]. Similarly there is increasing use of parallelism both in conventional computing but also in mobile applications. At present the epitome of parallelism are dedicated multi-core machines based on gaming graphics cards (GPUs). Although originally devised for the consumer market, they are increasingly being used for general purpose computing on GPUs (GPGPU) [Owens *et al.*, 2008] with several of today's fastest peta flop super computers being based on GPUs. However, although support tools are improving, programming parallel computers continues to be a challenge and simply leaving code generation to parallel compilers is often insufficient. Instead experts, e.g. [Merrill *et al.*, 2012], have advocated writing highly parametrised parallel code which can then be automatically tuned. Unfortunately this throws the load back on to the coder [Langdon, 2011]. Here we demonstrate that genetic programming can work with an auto-tuner to adapt human written code to new circumstances and different hardware. In total we consider six types of hardware of differing ages, architectures and very different performance (Table 1). GP can give more than a six fold performance increase relative to the original system on the same hardware (Table 4).

The next sections briefly gives the background to the GISMOE project and the StereoCamera CUDA code. This is followed by descriptions of the stereo images (page 5), and the code tuning process (pages 5–12).

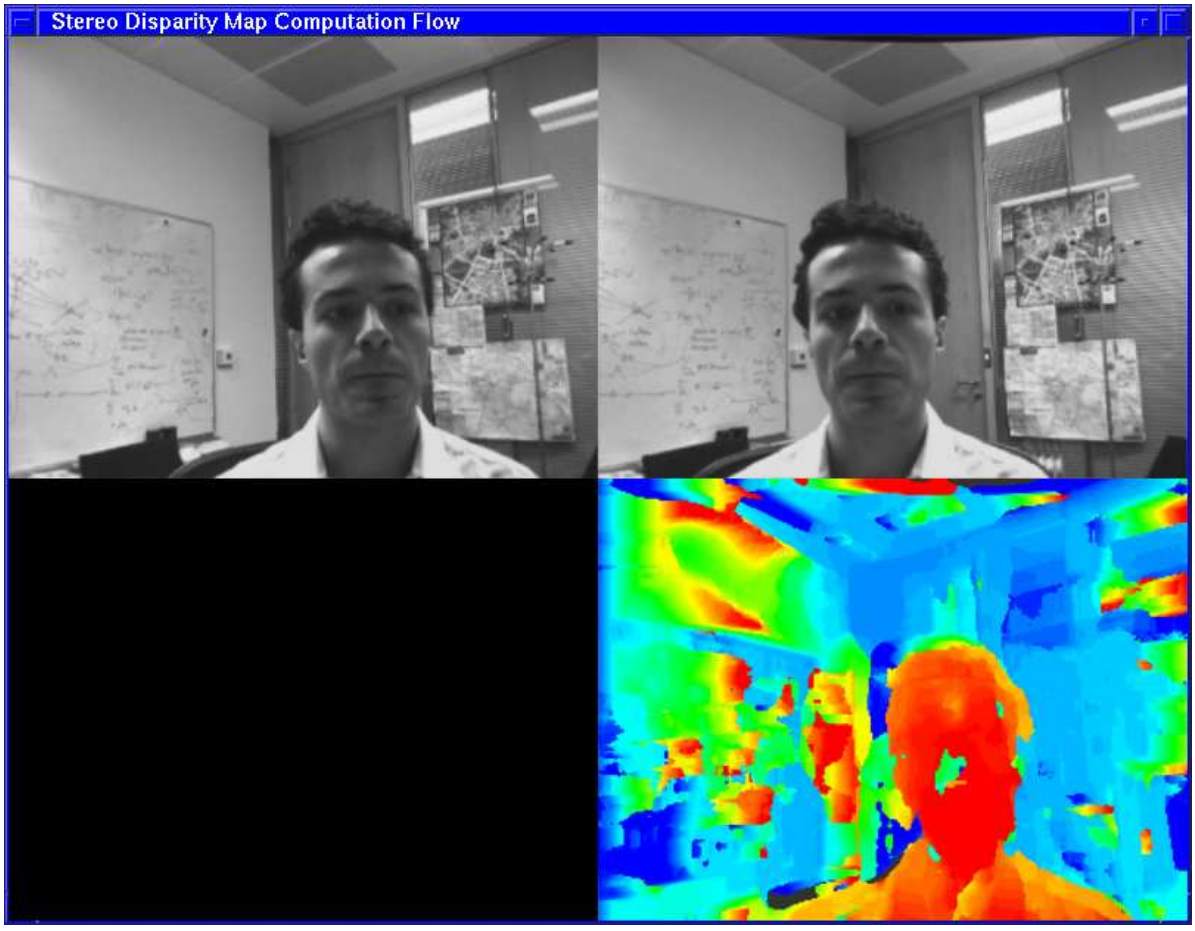

Figure 1: Top: left and right stereo images. Bottom: Discrepancy between images, which can be used to infer distance to camera.

Table 1: GPU Hardware. Year each was announced by nVidia in column 2. Third column is CUDA compute capability level. Each GPU chip contains a number of identical and more or less independent multiprocessors (column 4). Each MP contains a number of stream processors (cores, column 5) whose speed is given in column 7. Measured data rate (ECC on) between the GPU and its on board memory in last column.

Name	Announced	Capability	MP ×	cores	Clock	Caches		Bandwidth
					GHz	L1	L2	GB/s
Quadro NVS 290	2007	1.1	2 ×	8 = 16	0.92	none		4
GeForce GTX 295	2009	1.3	30 ×	8 = 240	1.24	none		92
Tesla T10	2009	1.3	30 ×	8 = 240	1.30	none		72
Tesla C2050	2010	2.0	14 ×	32 = 448	1.15	16/48KB	0.75 MB	101
GeForce GTX 580	2010	2.0	16 ×	32 = 512	1.54	16/48KB	0.75 MB	161
Tesla K20c	2012	3.5	13 ×	192 = 2496	0.71	16/32/48KB	1.25 MB	140

The changes made specifically for the K20c Tesla are described in Section 11 (page 14) whilst the Appendix (pages 20–23) holds the complete CUDA source code for the new stereoKernel tuned for compute level 3.5 devices, i.e. the K20c Tesla. The code is also available in `StereoCamera_v1_1c.zip`.

2 Background

In order to document the new genetically improved release of Stereo Camera we have extended [Langdon and Harman, 2014]. The UCL GISMOE project has taken a number of freely available non-trivial programs and shown they can be improved. In some cases this improvement can be substantial for a particular purpose [Langdon and Harman,] whilst the tailored code retained its general functionality [Langdon, 2013]. Stereo Camera has like wise been substantially improved on images like those it has been tailored for but the new version still gives improvement on other examples.

Our approach uses genetic programming (GP) [Koza, 1992; Poli *et al.*, 2008]. GP is increasingly being used in Software Engineering. The GISMOE project uses GP to make software more adaptable [Harman *et al.*, 2012] [Harman *et al.*,] [Jia *et al.*, 2013]. We are particularly interested in using GP to generate code [Langdon and Harman, 2010; Archanjo and Von Zuben, 2012]. There has recently been an explosion of interesting work on using GP for bug fixing [Le Goues *et al.*, 2012] and for improving existing code [Sitthi-amorn *et al.*, 2011] [White *et al.*, 2011] [Orlov and Sipper, 2011] [Langdon and Harman,] [Petke *et al.*, 2013] [Petke *et al.*, 2014] [Cotillon *et al.*, 2012] [Cody-Kenny and Barrett, 2013].

3 Source Code: StereoCamera

The StereoCamera system was written by nVidia's stereo image processing expert Joe Stam [Stam, 2008] to demonstrate their 2007 hardware and CUDA. It was the first to show GPUs could give real time performance ($>$ 30 frames per second) on stereo image processing. StereoCamera V1.0b is available from SourceForge [2] but, despite Moore's Law [Moore, 1965], and except for my bugfix, [3] it has not been updated since 2008. In the six years since it was written, nVidia GPUs have been through three major hardware architectures whilst their CUDA software has been through five major releases.

StereoCamera contains three GPU kernels plus associated host code, however we shall concentrate upon one, stereoKernel. For each pixel in the left image, GPU code stereoKernel reports the number of pixels the right image has to be shifted to get maximal local alignment (see Figure 2). [Stam, 2008] notes that the parallel processing power of the GPU allows the local discrepancy between the left and right images to be calculated using the sum of *squares* of the difference (SSD) between corresponding pixels and this

[2] http://sourceforge.net/projects/openvidia/files/CUDA Stereo Camera/

[3] My bug report: http://sourceforge.net/p/openvidia/discussion/342805/thread/34958dd9/

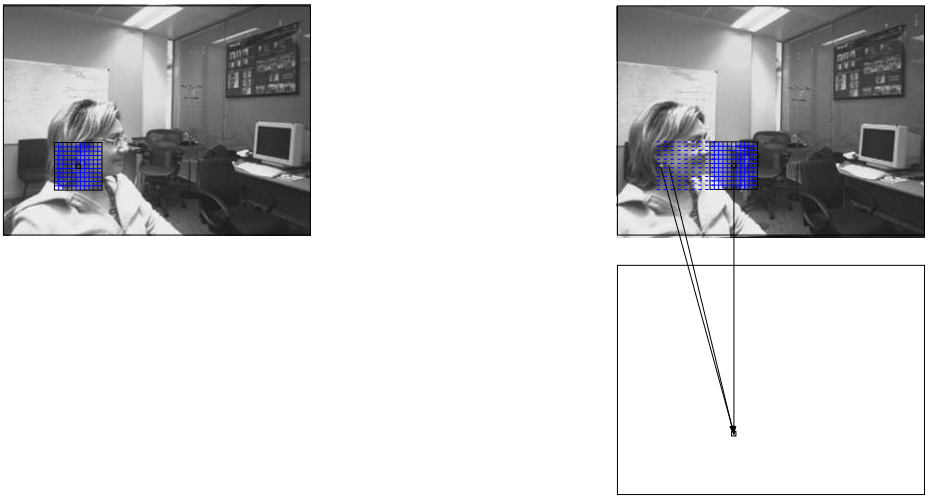

Figure 2: Schematic of stereo disparity calculation. Top: left and right stereo images. Bottom: output. Not to scale. For each pixel stereoKernel calculates the sum of squared differences (SSD) between 11×11 regions centred on the pixel in the left image and the same pixel in the right hand image. This is the SSD for zero disparity. The right hand 11×11 region is moved one place to the left and new SSD is calculated (SSD for 1 pixel of disparity). This is repeated 50 times. Each time a smaller SSD is found, it is saved. Although the output pixel (bottom) may be updated many times, its final value is the distance moved by the 11×11 region which gives the smallest SSD. I.e. the distance between left and right images which gives the maximum similarity between them (across an 11×11 region). This all has to be done for every pixel. Real time performance is obtained by parallel processing and reducing repeated calculations.

sum is taken over the relatively large 11×11 area. It does this by minimising the sum of squares of the difference (SSD) between the left and right images in a 11×11 area around each pixel. Once SSD has been calculated, the grid in the right hand image is displaced one pixel to the left and the calculation is repeated. Although the code is written to allow arbitrary displacements, in practice the right hand grid is move a pixel at a time. SSD is calculated for 0 to 50 displacements and the one with the smallest SSD is reported for each pixel in the left hand image. In principle each pixel's value can be calculated independently but each is surrounded by a "halo" of five others in each direction.

Even on a parallel computer, considerable savings can be made by reducing the total number of calculations by sharing intermediate calculations [Stam, 2008, Fig. 3]. Each SSD calculation (for a given discrepancy between left and right images) involves summing 11 columns (each of 11 squared discrepancy values). By saving the column sums in shared memory adjacent computational threads can calculate just their own column and then read the remaining ten column values calculated by their neighbouring threads.

After one row of pixel SSDs have been calculated, when calculating the SSD of the pixels immediately above, ten of the eleven rows of SSD values are identical. Given sufficient storage, the row values could be saved and then 10 of them could be reused requiring only one row of new square differences to be calculated. However fast storage was scare on GPUs and instead Stam compromised by saving the total SSD (rather than the per row totals). The SSD for the pixel above is then the total SSD plus the contribution for the new row *minus* the contribution from the lowest row (which is no longer included in the 11×11 area). Stam took care that the code avoids rounding errors. The more rows which share their partial results, the more efficient is the calculation but then there is less scope for performing calculations in parallel. To avoid re-reading data it is desirable that all the image data for both left and right images (including halos and discrepancy offsets) should fit within the GPU's texture caches. The macro `ROWSperTHREAD` (40) determines how many rows are calculated together in series. The macro `BLOCK_W` (64) determines how the image is partitioned horizontally (see Figures 3 and 4). To fit the GPU architecture `BLOCK_W` will often be a multiple of 32. In practise all these factors interact in non-obvious (and sometimes undocumented) hardware dependent ways.

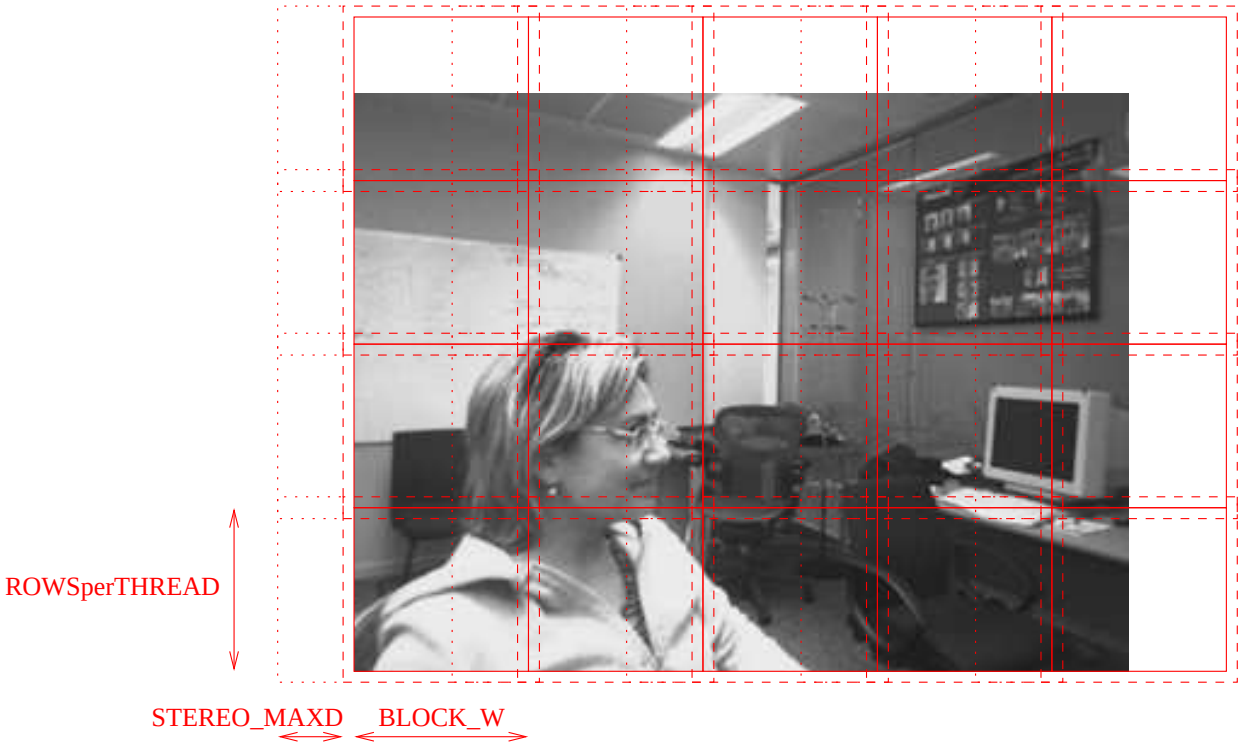

Figure 3: The left and right images (solid rectangle) are split into `BLOCK_W`×`ROWSperTHREAD` tiles. The dashed lines indicate the extra pixels outside the tile which must be read to calculate values for pixels in the tile. The right hand image is progressively offset by between zero and `STEREO_MAXD` pixels (50, dotted lines).

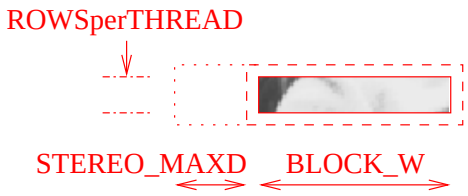

Figure 4: Part of right hand stereo image pair processed by a single CUDA thread block. The area covered in the right image is eventually shifted `STEREO_MAXD` (50) pixels to the left. For most GPUs, the optimal shape is greatly reduced vertically (`ROWSperTHREAD` reduced from 40 to 5) but width (`BLOCK_W`) is unchanged.

4 Example Stereo Pairs from Microsoft's I2I Database

Microsoft have made available for image processing research thousands of images. Microsoft's I2I database contains 3010 stereo images. Figure 2 (top) is a typical example. Many of these are in the form of movies taken in an office environment. Figure 1 shows the first pair from a typical example.

We downloaded i2idatabase.zip[4] (1.3GB) and extracted all the stereo image pairs and converted them to grey scale. Almost images all are 320×240 pixels. We took (up to) the first 200 pairs for training leaving 2810 for validation. Notice we are asking the GP to create a new version of the CUDA stereoKernel GPU code which is tuned to pairs of images of this type. As we shall see (in Section 10) the improved GPU code is indeed tuned to 320×240 images but still works well on the other I2I stereo pairs.

5 Host Code and Baseline Kernel Code

The supplied C++ code is designed to read stereo images from either stereo webcams or pairs of files and using OpenGL, to display both the pair of input images and the calculated discrepancy between them on the user's monitor. (see Figure 1). This was adapted to both compare answers generated by the original code with those given by the tuned GP modified code and to time execution of the modified GPU kernel code. These data are logged to a file and the image display is disabled.

The original kernel code is in a separately compiled file to ensure it is not affected by GP specified compiler options (particularly -Xptxas -dlcm, Table 2). For each pixel it generates a value in the range $0.0, 1.0, 2.0 \ldots 50.0$ being the minimum discrepancy between the left and right images. If a match between the left and right images cannot be found (i.e. SSD ≥ 500000) then it returns -1.0.

6 Pre- and Post- Evolution Tuning and Post Evolution Minimisation of Code Changes

In initial genetic programming runs, it became apparent that there are two parameters which have a large impact on run time but whose default settings are not suitable for the GPUs now available. Since there are few such parameters and they each have a small number of sensible values, it is feasible to run StereoCamera on all reasonable combinations and simply choose the best for each GPU. Hence the revised strategy is to tune `ROWSperTHREAD` and `BLOCK_W` before running the GP. (`DPER`, Section 7.2, is not initially enabled.) Figure 5 shows the effect of tuning `ROWSperTHREAD` and `BLOCK_W` for the GTX 295. As with [Le Goues *et al.*, 2012] and our GISMOE approach [Langdon and Harman,], after GP has run the best GP individual from the last generation is cleaned up by a simple one-at-a-time hill climbing algorithm. [Langdon and Harman,] (Section 6) and finally `ROWSperTHREAD`, `BLOCK_W` and `DPER` are tuned again. (Often no further changes were needed.)

For each combination of parameters, the kernel is compiled and run. By recompiling rather than using run time argument passing, the nVidia nvcc C++ compiler is given the best chance of optimising the code (e.g. loop unrolling) for these parameters and the particular GPU.

`BLOCK_W` values were based on sizes of thread blocks used by nVidia in the examples supplied with CUDA 5.0. (They were 8, 32, 64, 128, 192, 256, 384 and 512.) All small `ROWSperTHREAD` values or values which divide into the image height (240) were tested. (I.e., 1, … 18, 20, 21, 24, 26, 30, 34, 40, 48, 60, 80, 120 and 240.) Except for the NVS 290, which has only two multiprocessors, autotuning reduced `ROWSperTHREAD` from 40 to 5 before the GP was run. In many cases this gave a big speed up (see Figure 4 and middle and last columns of Table 4).

The best GP individual in the last generation is minimised by starting at its beginning and progressively removing each individual mutation and comparing the performance of the new kernel with the evolved one. For simplicity this is done on the last training stereo image pair. Unless the new kernel is worse the

[4] http://research.microsoft.com/en-us/um/people/antcrim/data_i2i/i2idatabase.zip

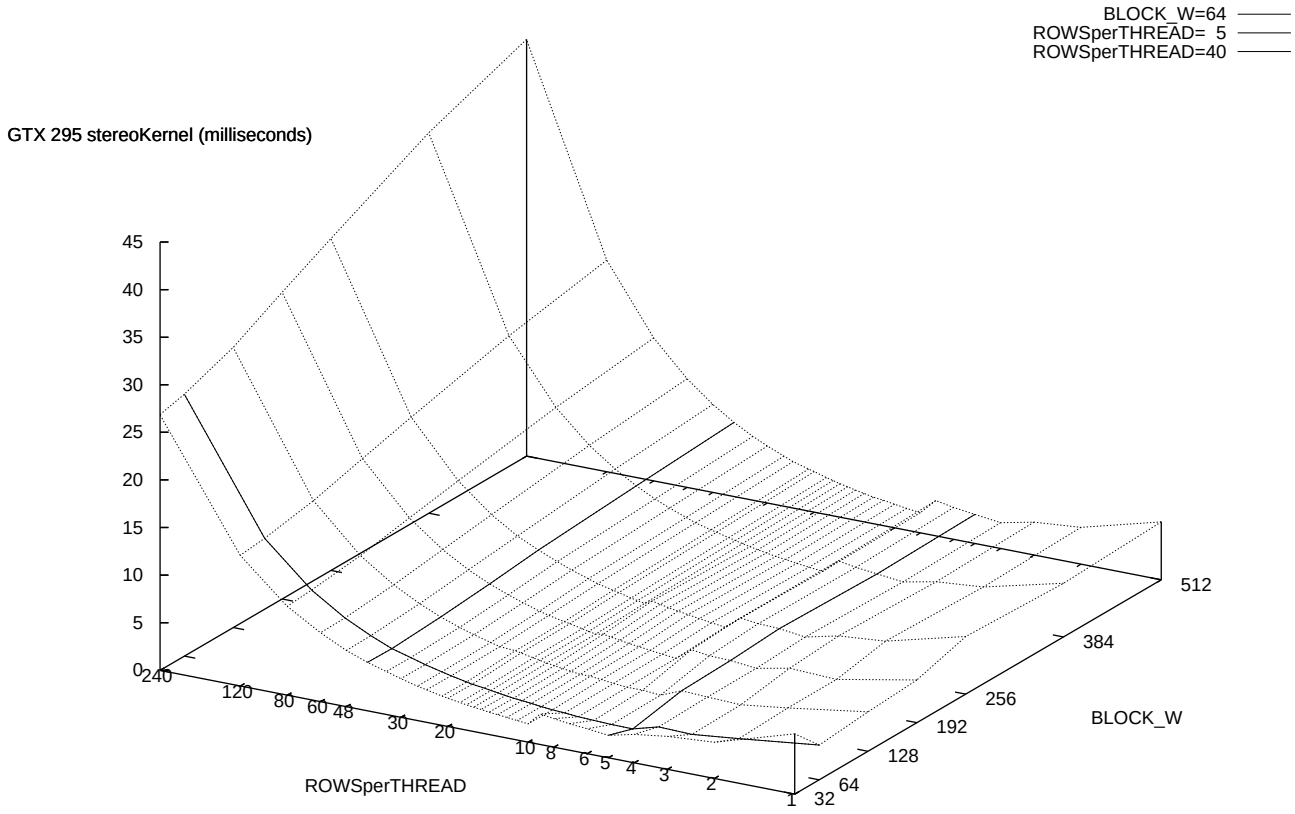

Figure 5: Effect of changing work done per thread (`ROWSperTHREAD`) and block size (`BLOCK_W`) and CUDA kernel speed before it is optimised by GP. stereoKernel is fastest at 5,64 (default is 40,64).

mutation is excluded permanently. To encourage removal of mutations with little impact, those that make less than 1% difference to the kernel timing are also removed.

In the after evolution tuning, if GP had enabled `DPER` (Section 7.2) then as well as tuning `BLOCK_W` and `ROWSperTHREAD` the autotuner tried values 1..4 for `DPER`. (In the two cases, GTX 580 and K20c, where GP enabled `DPER` its default value, 2, was optimal, and hence unchanged by the post evolution autotuner.)

7 Alternative Implementations

7.1 Avoiding Reusing Threads: XHALO

As mentioned in Section 3 each row of pixels is extended by five pixels at both ends. The original code reused the first ten threads of each block to calculate these ten halo values. Much of the kernel code is duplicated to deal with the horizontal halo. GPUs use SIMD parallel architectures, which means many identical operations can be run in parallel but if the code branches in different directions part of the hardware becomes idle. (This is known as thread divergence.) Thus diverting ten threads to deal with the halo causes all the remaining threads in the warp (32 threads) to become idle. Option XHALO allows GP to use ten additional threads which are dedicated to the halo. Thus each thread only deals with one pixel. In practise the net effect of XHALO is to disable the duplicated code so that instead of each block processing vertical stripes of 64 pixels, each block only writes stripes 54 pixels wide.

7.2 Parallel of Discrepancy offsets: DPER

The original code (Section 3) steps through sequentially 51 displacements of the right image with respect to the left. Modern GPUs allow many more threads and often it is best to use more threads as it allows greater parallelism and may improve throughput by increasing the overlap between computation and I/O. Instead

Table 2: Evolvable configuration macros and constants

Name	Default	Options	Purpose
Cache preference	None	None, Shared, L1, Equal	L1 v. shared memory
-Xptxas -dlcm		' ', ca, cg, cs, cv	nvcc cache options
OUT_TYPE	`float`	`float, int, short int, unsigned char`	C type of output
STORE_disparityPixel	GLOBAL	GLOBAL, SHARED, LOCAL	
STORE_disparityMinSSD	GLOBAL	GLOBAL, SHARED, LOCAL	
DPER	disabled		Section 7.2
XHALO	disabled		Section 7.1
__mul24(a,b)	__mul24	__mul24, `*`	fast 24-bit multiply
GPtexturereadmode	`Normalized Float`	`NormalizedFloat`, `ElementType`, no Textures	Section 8.1.4
texturefilterMode	`Linear`	`Linear`, `Point`	
textureaddressMode		`Clamp`, `Mirror`, `Wrap`	
texturenormalized		0, 1	

of stepping sequentially one at a time through the `for` loop controlling the displacement, the DPER option allows SSD values for multiple (e.g. 2, 3 or 4) displacements to be calculated in parallel. So instead of increasing the `for` loop control variable by one, it is incremented by the same amount (e.g. 2, 3 or 4). As well as increasing the number of threads, the amount of shared memory needed is also increased by the same factor. Nevertheless only one (the smallest) SSD value need be compared with the current smallest, so potentially saving some I/O. Although the volume of calculations is little changed, there are also potential saving since each DPER block uses almost the same data.

8 Parameters Accessible to Evolution

The GISMOE GP system [Langdon and Harman,] was extended to allow not only code changes but also changes to C macro `#defines`. The GP puts the evolved values in a C `#include .h` file, which is complied along with the GP modified kernel code and the associated (fixed) host source code.

Table 2 shows the twelve configuration parameters. Every GP individual chromosome starts with these 12 which are then followed by zero or more changes to the code.

8.1 Fixed Configuration Parameters

8.1.1 OUT_TYPE

The return value should be in the range -1 to 50 (Section 5). Originally this is coded as a `float`. OUT_TYPE gives GP the option of trying other types. Notice we do not use the fact that the smaller data types take less time to transfer between GPU and host, since the data will probably be used on the GPU. (I.e. all fitness times, Section 9.5.2, are on the GPU.)

8.1.2 STORE_disparityPixel and STORE_disparityMinSSD

disparityPixel and disparityMinSSD are major arrays in the kernel. Stam coded them to lie in the GPU's slow off chip global memory. These configuration options give evolution the possibility of trying to place them in either shared memory or in local memory. Where the compiler can resolve local array indexes, e.g. as a result of unrolling loops, it can use fast registers in place of local memory.

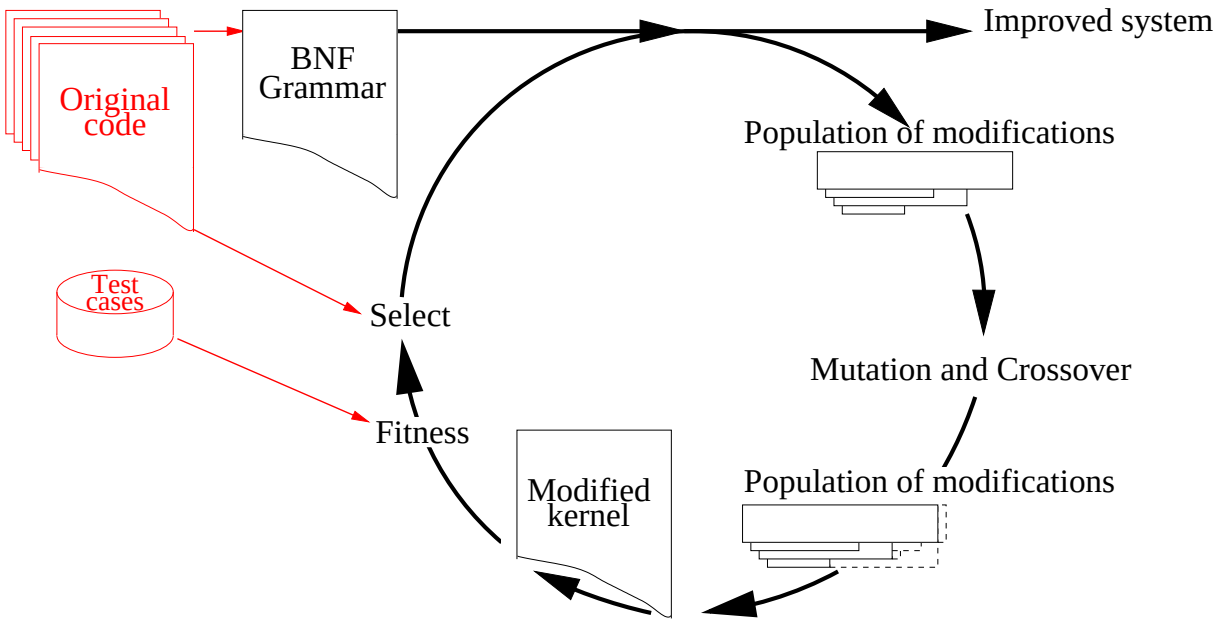

Figure 6: Genetic Improvement of stereoKernel

8.1.3 __mul24

For addressing purposes, older GPU's included a fast 24 bit multiply instruction, which is heavily used in the original code. It appears that in the newer GPUs __mul24 may actually be slower than ordinary (32 bit) integer multiply. Hence we give GP the option of replacing __mul24.

8.1.4 Textures

CUDA textures are intimately linked with the GPU's hardware and provide a wide range of data manipulation facilities (normalisation, default values, control of boundary effects and interpolation) which the original code does not need but is obliged to use. The left and right image textures are principally used because they provide caching (which was not otherwise available on early generation GPUs.) We allowed the GP to investigate other texture options. Including not using textures. Some combinations are illegal but the host code gives sensible defaults in these cases.

Whilst we investigated the use of textures in the code examples supplied with CUDA 5.0, we decided to allow GP access to all the various options for setting up textures and indeed to avoid textures and access the image data directly. There is a $\frac{1}{2}$ pixel discrepancy between direct access (which treats the images as 2D arrays) and textures where reference point is the centre of the pixel. This leads to small differences between direct access and the original code. Whilst such slight differences make little difference to the outputs appearance even so they are penalised by the fitness function (Section 9.5).

9 Evolvable Code

Following the standard GISMOE approach [Langdon and Harman,], cf. Figure 6, a grammar describing the legal changes to the kernel source code was automatically created from the human written source code. Due to the way Stam wrote his kernel (with all variables declared at the start) no mutation moves variables out of scope. Thus almost all GP created kernels compile, link and run. The only exception being two cases where GP created legal source code which provoked bugs in the nvcc 5.0 compiler. It is believed these bugs have been fixed in 5.5. Although legal, in both cases, even if the kernels did compile, they would have had low fitness. (To allow the GP run to continue, they are given a very poor fitness. This ensures they will not be selected to be parents of the next generation.)

Table 3: Genetic programming parameters for improving stereoKernel

Representation:	Fixed list of 12 parameter values (Table 2) followed by variable list of replacements, deletions and insertions into BNF grammar
Fitness:	Run on a randomly chosen 320×240 monochrome stereo image pair. Compare answer & run time with original code and time its execution. See Sections 9.5 and 9.6.
Population:	Panmictic, non-elitist, generational. 100 members. New randomly chosen training sample each generation.
Parameters:	Initial population of random single mutants heavily weighted towards the kernel header and shared variables. 50% truncation selection. 50% crossover (uniform for fixed part, 2pt for variable). 50% mutation 25% mutation random change to fixed part. 25% add code mutation (one of: delete, replace, insert, each equally likely). No size limit. Stop after 50 generations.

As with [Langdon and Harman,], the source code, including XHALO and DPER (Sections 7.1 and 7.2), is automatically translated line by line into a BNF grammar (see Figure 7). Notice the grammar is not generic, it represents only one program, stereoKernel, and variants of it. The grammar contains 424 rules, 277 represent fixed lines of C++ source code. There are 55 variable lines, 27 IF and 10 of each of the three parts of C `for` loops. (The kernel grammar does not contain any WHILE or ELSE rules.) In addition to these standard types we introduce five CUDA specific types:

pragma allows GP to control the nvcc compiler's loop unrolling. pragma rules are automatically inserted before each `for` loop but rely on GP to enable and set their values. Using the type constraints GP can either: remove it, set it to `#pragma unroll`, or set it to `#pragma unroll` n (where n is 1 to 11).

optvolatile CUDA allows shared data types to be marked as `volatile` which influences the compiler's optimisation. As required by the CUDA compiler, the grammar automatically ensures all shared variables are either flagged as `volatile` or none are.

The remaining three CUDA types apply to the kernel's header.

optconst Each of kernel's scalar inputs can be separately marked as `const`.

optrestrict All of the kernel's array arguments can be marked with `__restrict__`. This potentially helps the compiler to optimise the code. On the newest GPUs (SM 3.5) optrestrict allows the compiler to access read only arrays via a read only cache. Since both only apply if all arrays are marked `__restrict__`, the grammar ensures they all are or none are.

launchbounds is again a CUDA specific aid to code optimisation. By default the compiler must generate code that can be run with any numbers of threads. Since GP knows how many threads will be used, specifying it via `__launch_bounds__` gives the compiler the potential of optimising the code. `__launch_bounds__` takes an optional second argument which refers to the number of blocks that are active per MP. How it is used is again convoluted, but the grammar allows GP to omit it, or set it to 1, 2, 3, 4 or 5.

9.1 Initial Population

Each member of the initial population is unique. They are each created by selecting at random one of the 12 configuration constants (Table 2) and setting it at random to one of its non-default values. As the population is created it becomes harder to find unique mutations and so random code changes are included as well as the configuration change. Table 3 summarises the GP parameters.

RN/14/02

```
<KStereo.cuh_52>   ::= "__attribute__((global)) " <launchbounds_KStereo.cuh_52>
                       " void KERNEL(\n"
#kernel
<launchbounds_KStereo.cuh_52> ::= ""
<launchbounds_K0>  ::= "\n" "#ifdef DPER\n" "__launch_bounds__(BLOCK_W*dperblock)\n"
                       "#else\n" "__launch_bounds__(BLOCK_W)\n"   "#endif /*DPER*/\n"
                    ...
<launchbounds_K5>  ::= "\n" "#ifdef DPER\n" "__launch_bounds__(BLOCK_W*dperblock,5)\n"
                       "#else\n" "__launch_bounds__(BLOCK_W,5)\n" "#endif /*DPER*/\n"
<optrestrict_KStereo.cuh_52> ::= " __restrict__ "
#kernelarg
<KStereo.cuh_53>   ::= "OUTYPE *" <optrestrict_KStereo.cuh_52> "disparityPixel,\n"
<KStereo.cuh_54>   ::= <optconst_KStereo.cuh_54> "size_t out_Pitch,\n"
<optconst_KStereo.cuh_54> ::= "const "
<KStereo.cuh_55>   ::= "#ifdef GLOBAL_disparityMinSSD\n"
<KStereo.cuh_56>   ::= "int *" <optrestrict_KStereo.cuh_52> "disparityMinSSD,\n"
<KStereo.cuh_57>   ::= "#if OUT_TYPE != float_ && OUT_TYPE != int_\n"
<KStereo.cuh_58>   ::= <optconst_KStereo.cuh_58> "size_t out_pitch,\n"
<optconst_KStereo.cuh_58> ::= "const "
<KStereo.cuh_59>   ::= "#endif\n"
<KStereo.cuh_60>   ::= "#endif /*GLOBAL_disparityMinSSD*/\n"
                   ...
<KStereo.cuh_72>   ::= ")\n"
                   ...
<KStereo.cuh_141>  ::= " if" <IF_KStereo.cuh_141> "  extra_read_val = BLOCK_W+threadIdx.x;\n"
#"if
<IF_KStereo.cuh_141> ::= "(threadIdx.x < (2*RADIUS_H))"
                   ...
<KStereo.cuh_158>  ::= <pragma_KStereo.cuh_158> "for(" <for1_KStereo.cuh_158> ";" "OK()&&"
                       <for2_KStereo.cuh_158> ";" <for3_KStereo.cuh_158> ") \n"
#for
<pragma_KStereo.cuh_158> ::= ""
#pragma
<pragma_K0>        ::= "#pragma unroll \n"
<pragma_K1>        ::= "#pragma unroll 1\n"
                   ...
<pragma_K11>       ::= "#pragma unroll 11\n"
<for1_KStereo.cuh_158> ::= "i = 0"
<for2_KStereo.cuh_158> ::= "i<ROWSperTHREAD && Y+i < height"
<for3_KStereo.cuh_158> ::= "i++"
<KStereo.cuh_159>  ::= "{\n"
<KStereo.cuh_160>  ::= "" <_KStereo.cuh_160> "\n"
#other
<_KStereo.cuh_160> ::= "init_disparityPixel(X,Y,i);"
<KStereo.cuh_161>  ::= "" <_KStereo.cuh_161> "\n"
<_KStereo.cuh_161> ::= "init_disparityMinSSD(X,Y,i);"
<KStereo.cuh_162>  ::= "}\n"
```

Figure 7: Fragments of BNF grammar used by GP. Most rules are fixed but rules starting with <_, <IF_, <for1_, <pragma_, etc. can be manipulated using rules of the same type to produce variants of stereo Kernel. Lines beginning with # are comments.

9.2 Weights

Since most lines of kernel code are always used we do not use the Obins part of the GISMOE framework, instead each line of code is equally likely to be modified. However, only as part of creating a diverse initial population, the small number of rules in the kernel header (i.e. launchbounds, optrestrict, optconst and optvolatile) are 1000 times more likely to be changed than the other grammar rules. (Forcing each member of the GP population to be unique is only done in the initial population.) In future, it might be worthwhile ensuring GP does not waste effort changing CUDA code which can have no effect by setting the weights of lines excluded by conditional compilation to zero.

9.3 Mutation

Half of mutations are made to the configuration parameters (Table 2). In which case one of the 12 is chosen uniformly at random and its current value is replaced by another of its possible values again chosen uniformly at random. For the code, we use the three GISMOE mutations: delete a line of code, replace a line and insert a line [Langdon and Harman,]. The additional lines of code are not random but are copied from stereoKernel itself. This is like [Le Goues *et al.*, 2012] except we use the grammar.

9.4 Crossover

As in the GISMOE frame work [Langdon and Harman,], crossover creates a new GP individual from two different members of the better half (Section 9.6) of the current population. The child inherits each of the 12 fixed parameters (Table 2) at random from either parent (uniform crossover [Syswerda, 1989]). Whereas in [Langdon and Harman,] we used append crossover, which deliberately increases the size of the offspring, here, on the variable length part of the genome, we use an analogue of Koza's tree GP crossover [Koza, 1992]. Two crossover points are chosen uniformly at random. The part between the 2 crossover points of the first parent is replaced by the mutations between the two crossover points of the second parent to give a single child. On average, this gives no net change in length.

9.5 Fitness

To avoid over fitting and to keep run times manageable, each generation one of the two hundred training images pairs is chosen [Langdon, 2010]. Each GP modified kernel in the population is tested on that image pair.

9.5.1 CUDA memcheck and Loop Overruns

Normally each GP modified kernel is run twice. The first time it is run with CUDA memcheck and with loop over run checks enabled. If no problems are reported by CUDA memcheck and the kernel terminates normally (i.e. without exceeding the limit on loop iterations) it is run a second time without these debug aids. Both memcheck and counting loop iterations impose high overheads which make timing information unusable. Only in the second run are the timing and error information used as part of fitness. If the GP kernel fails in either run, it is given such a large penalty, that it will not be a parent for the next generation.

When loop timeouts are enabled, the GP grammar ensures that each time a C++ `for` loop iterates a per thread global counter is incremented. If the counter exceeds the limit, the loop is aborted and the kernel quickly terminates. If any thread reaches its limit, the whole kernel is treated as if it had timed out. The limit is set to $100\times$ the maximum reasonable value for a correctly operating good kernel.

9.5.2 Timing

Each of the Multiprocessors (MPs) within the GPU chip has its own independent clock. On some GPUs `cudaDeviceReset()` also resets all the clocks, this is not the case with the C2050. To get a robust timing scheme, which applies to all GPUs, each kernel block records both its own start and end times and

Table 4: Mean speed across all 2516 I2I 320×240 stereo image pairs. ± is standard deviation. Times in microseconds. In all cases tuning leaves `BLOCK_W` as 64. Tuning NVS 290 *increases* `ROWSperTHREAD` from 40 to 120, otherwise pretuning reduces it to 5. Post GP tuning leaves `ROWSperTHREAD` as 5, except C2050 (14) and GTX 580 (15).

GPU name	Original	Pretuned	Ratio	GP	Speedup
Quadro NVS 290	27402±116	26019±152			1.053±0.01
GeForce GTX 295	5448± 14	1518± 4			3.589±0.01
Tesla T10	5256± 12	1436± 3	3.661±0.01	1359±38	3.861±0.11
Tesla C2050	4632± 25	3017± 15	1.535±0.01	1130± 5	4.099±0.02
GeForce GTX 580	3077± 21	1650± 6	1.865±0.01	722±29	4.248±0.17
Tesla K20c	4362± 21	1839± 18	2.373±0.03	638± 1	6.837±0.04

the MP unit it is running on. (This cannot be modified by the GP.) After the kernel has finished, for each MP, the end time of the last block to use it and the start time of the first block to use it are subtracted to give the accurate duration of usage for each MP. (Note to take care of overflow `unsigned int` arithmetic is used.) Whilst we do not compare values taken from clocks on different MPs, it turns out to be safe to assume that the total duration of the kernel is the longest time taken by any of the MPs used. (As a sanity check this GPU kernel time is compared to the, less accurate, duration measured on the host CPU.) The total duration taken by the GP kernel (expressed as GPU clock tics divided by 1000) is the first component of its fitness.

9.5.3 Error

For each pixel in the left image the value returned by the GP modified kernel is compared with that given by the un-modified kernel. If they are different a per pixel penalty is added to the total error which becomes the second part of the GP individual's fitness.

If the unmodified kernel did not return a value (i.e. it was -1.0, cf. Section 5) the value returned by the GP kernel is also ignored. Otherwise, if the GP failed to set a value for a pixel, it gets a penalty of 200. If the GP value is infinite or otherwise outside the range of expected values (0..50) it attracts a penalty of 100. Otherwise the per pixel penalty is the absolute difference between the original value and the GP's value.

For efficiency, previously [Langdon and Harman, 2010] we batched up many GP generated kernels into one file to be compiled in one go. For simplicity, since we are using a more advanced version of nVidia's nvcc compiler, and GP individuals in the same population may need different compiler options, we did not attempt this. Typically it takes about 3.3 seconds to compile each GP generated kernel. Whereas to run the resulting StereoCamera program (twice see Section 9.5) takes about 2.0 seconds,

9.6 Selection

As with the GISMOE framework [Langdon and Harman,] at the end of each generation we compare each mutant with the original kernel's performance on the same test case and only allow it to be a parent if it does well. In detail, it must be both faster and be, on average, not more than 6.0 per pixel different from the original code's answer. However mostly the evolved code passes both tests. At the end of each generation the population is sorted first by their error and then by their speed. The top 50% are selected to be parents of the next generation. Each selected parent creates one child by mutation (Section 9.3) and another by crossover with another selected parent (Section 9.4). The complete GP parameters are summarised in Table 3.

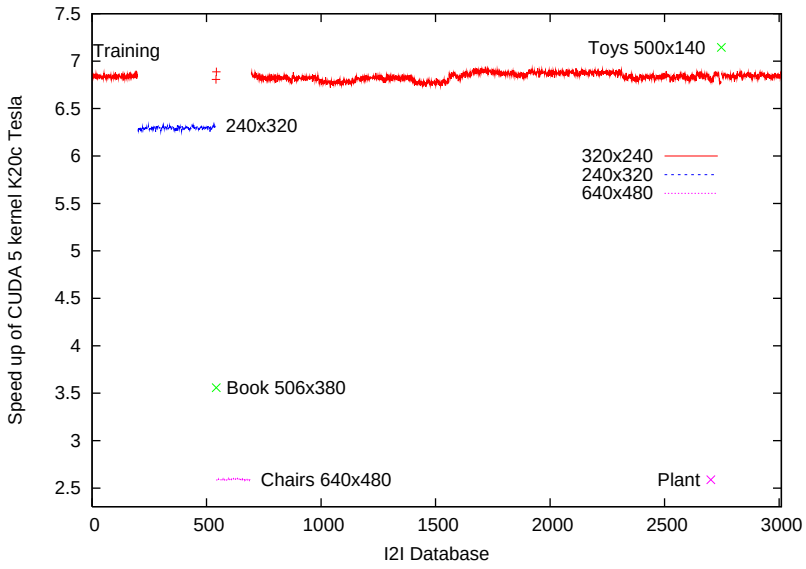

Figure 8: Performance of GP improved K20c Tesla kernel on all 3010 stereo pairs in Microsoft's I2I database relative to original kernel on the same image pair on the same GPU. Fifty of first 200 pairs used in training. The evolved kernel is always much better, especially on images of the same size and shape as it was trained on.

10 Results

Table 4 gives the speed up for six types of GPUs. By reducing `ROWSperTHREAD` from the original 40 to 5, pretuning (Section 6) itself gave considerable speed ups (columns 4-5 in Table 4). However for NVS 290, tuning `ROWSperTHREAD` increased it from 40 to 120 but only gave a modest improvement (last columns in Table 4). In all cases the original value of `BLOCK_W` (64) was optimal.

Unfortunately with CUDA 5.0 memcheck (Section 9.5.1), it proved impossible to keep the NVS 290 and GTX 295 operational for a complete GP run. Despite hardware monitoring, the problem remained non-reproducible. It is thought with more recent hardware, memcheck is able to catch and prevent problems caused by incorrect array indexes but on the NVS 290 and GTX 295 GPUs (with nVidia driver 310.40) incorrect program operation eventually lead to hardware lock up. This is at odds with our earlier successful use of GP on the GTX 295, where we had explicitly caught out-of-range indexes [Langdon and Harman, 2010]. We had expected that a more modern version of memcheck would provide this facility at lower overhead. However both memcheck and catching indefinite loops (Section 9.5.1) interfere with timing to such and extent that during fitness testing it is necessary to run correctly operating kernels a second time without either memcheck or loop checking to get accurate timings. Hence it might have been better to provide our own array bounds index checking. In Table 4 the "GP" columns for the NVS 290 and GTX 295 rows are blank and the last column refers to the speed up achieved by tuning `ROWSperTHREAD` and `BLOCK_W`.

With the four more modern GPUs, the best individual from the last generation (50) was minimised to remove unneeded mutations which contributed little to its overall performance and retuned (Section 6). This resulted in reductions in length: T10 31→14, C2050 17→10, GTX580 26→13 and K20c 29→10. The speeds of the re-tuned kernels are given in Table 4 under heading "GP". In each case this gave a significant speed up (last column of Table 4) compared to both the original kernel and the original kernel with the best `ROWSperTHREAD` setting. The speedup of the improved K20c kernel on all of the I2I stereo images is given in Figure 8. The speed up for the other five GPUs varies in a similar way to the K20c. Finally, notice typically there is very little difference in performance across the images of the same size and shape as the training data (see $\pm$ columns in Table 4).

Table 5: Numbers of most popular of each of the evolvable configuration macros and constants (Table 2) in the last breeding population.

Fixed mutation	Tesla T10		Tesla C2050		GTX 580		Tesla K20c	
Cache	None	62	L1	52	L1	66	None	48
-Xptxas -dlcm	ca	84	not used	50	cg	42	not used	32
OUT_TYPE	`float`	100	`float`	74	`float`	76	`float`	48
STORE_Pixel	LOCAL	100	LOCAL	100	LOCAL	76	GLOBAL	70
STORE_MinSSD	SHARED	100	SHARED	100	SHARED	56	SHARED	76
DPER	disabled	100	disabled	100	used	100	used	100
XHALO	disabled	100	used	100	used	100	used	100
__mul24(a,b)	__mul24	100	*	100	*	70	__mul24	98
GPtexturereadmode	Normalized	100	Normalized	100	Normalized	100	Normalized	100
texturefilterMode	Linear	100	Linear	100	Linear	100	Linear	100
texturenormalized	default	82	default	80	default	72	default	72
textureaddressMode	Wrap	40	Clamp	66	Mirror	42	Mirror	48

```
DPER=1 STORE_disparityMinSSD=SHARED XHALO=1 STORE_disparityPixel=SHARED
<pragma_KStereo.cuh_359><pragma_K3> <_KStereo.cuh_161>+<_KStereo.cuh_224>
<_KStereo.cuh_348> <optvolatile_KStereo.cuh_86> <pragma_KStereo.cuh_262><pragma_K11>
<IF_KStereo.cuh_326><IF_KStereo.cuh_154>
```

Figure 9: Best GP individual in generation 50 of K20c Tesla run after minimising, Section 6, removed less useful components. (Auto-tuning made no further improvements.)

10.1 GP better than Random Search

In the case of the K20c Tesla, the GP was run again for the same number of evaluations, the same population size, the same number of generations *but* with random selection of parents. The best in the whole run of 50 generations of random search is exceeded by the best in the third and subsequent GP generations.

11 Evolved Tesla K20c CUDA Code

For brevity we describe in detail only one of the evolved CUDA stereo kernels. The best of generation 50 individual changes 6 of the 12 fixed configuration parameters (Table 2) and includes 23 grammar rule changes. After removing less useful components (Section 6) four configuration parameters were changed and there were six code changes. See Figures 9 and 10. The complete code is given in the appendix (pages 20–23).

*int * __restrict__ disparityMinSSD,* //Global disparityMinSSD not kernel argument
volatile `extern __attribute__((shared)) int col_ssd[];`
volatile `int* const reduce_ssd = &col_ssd[(64 )*2 -64];`
`#pragma unroll 11`
if(X < width `&&` *Y < height)* replaced by `if(dblockIdx==0)`
__syncthreads();
`#pragma unroll 3`

Figure 10: Evolved changes to K20c Tesla StereoKernel. (Produced by GP grammar changes in Figure 9). Highlighted `code` is inserted. Code in *italics* is removed. For brevity, except for the kernel's arguments, disparityPixel and disparityMinSSD changes from global to shared memory are omitted. The appendix, pages 20–23, gives the complete source code.

DPER is enabled and the new kernel calculates two disparity values in parallel, Section 7.2. disparityPixel and disparityMinSSD are stored in shared memory, Section 8.1.2 and XHALO is enabled, Section 7.1.

The final code changes, Figure 10, are:

- disable volatile, Section 9.
- insert `#pragma unroll 11` before the `for` loop that steps through the `ROWSperTHREAD` - 1 other rows (Section 3).
- insert `#pragma unroll 3` before the `for` loop that writes each of the `ROWSperTHREAD` rows of disparityPixel from shared to global memory. Its not clear why evolution chose to ask the nvcc compiler to unroll this loop (which is always executed 5 times) only 3 times. But then when nvcc decides to do loop unrolling is obscure anyway.
- Mutation `<_KStereo.cuh_161>+<_KStereo.cuh_224>` causes line 224 to be inserted before line 161. Line 224 potentially updates local variable `ssd`, however `ssd` is not used before the code which initialises it. It is possible that compiler spots that the mutated code cannot affect anything outside the kernel and simply optimises it away. During minimisation removing this mutation gave a kernel whose run time was exactly on the removal threshold.
- Mutation `<IF_KStereo.cuh_326><IF_KStereo.cuh_154>` replaces `X < width && Y < height` by `dblockIdx==0`. This replace a complicated expression by a simpler (and so presumably faster) expression, which itself has no effect on the logic since both are always true. In fact, given the way `if(dblockIdx==0)` is nested inside another `if`, the compiler may optimise it away entirely. I.e. GP has found a way of improving the GPU kernel by removing a redundant expression.

  The original purposed of `if(X < width && Y < height)` was to guard against reading outside array bounds when calculating SSD. However the array index is also guarded by `i < blockDim.x`
- delete `__syncthreads()` on line 348. `__syncthreads()` forces all threads to stop and wait until all reach it. Line 348 is at the end of code which may update (with the smaller of two disparities values) shared variables disparityPixel and disparityMinSSD. In effect GP has discovered it is safe to let other threads proceed since they will not use the same shared variables before meeting other `__syncthreads()`. elsewhere in the code. Removing synchronisation calls potentially allows greater overlapping of computation and I/O leading to an overall saving.

12 Discussion

In some cases modern hardware readily gives on line access to other important non-functional properties (such as power or current consumption, temperature and actual clock speeds) of software as it runs. Potentially these might also be optimised by GP. ([White *et al.*, 2008] showed it can be possible to use GP with a cycle-level power level simulator to optimise small programs for embedded systems.) Here we work with the real hardware, rather than simulators, however real power measurements are not readily available with all our GTX and Tesla cards.

Many computers, including GPUs, especially in mobile devices, now have variable power consumption. Thus reducing execution time can lead to a proportionate reduction in energy consumption and hence increase in battery life, since as soon as the computation is done the computer can revert to its low power idle hibernating state.

Another promising extension is the combined optimisation for multiple functional and non-functional properties [Colmenar *et al.*, 2011]. Initial experiments hinted that NSGA-II [Deb *et al.*, 2002; Langdon *et al.*, 2010] finds it hard to maintain a complete Pareto front when one objective is much easier than the others.

Thus a population may evolve to contain many fast programs which have lost important functionality while slower functional program are lost from the population.

The latest version of CUDA (5.5) includes additional tools (e.g. CUDA race check) which might be included as part of fitness testing.

The supplied kernel code contains several hundred lines of code. It may be that this only just contains enough variation for GP's cut-and-past operations (Section 9.3). With this in mind we had intended to allow GP to also use code taken from the copious examples supplied by nVidia with CUDA 5.0 (see [Petke *et al.*, 2014]) but in the end the only use made of these samples was in the tuning operations (Section 6) where `BLOCK_W` was tested as all the block size values used by nVidia's experts in these CUDA samples.

nVidia and other manufactures are continuing to increase the performance, economy and functionality of their parallel hardware. There are also other highly parallel (e.g. Intel) and low power chips with diverse architectures. These trends suggest the need for software to be ported [Langdon and Harman, 2010] to or adapt to new parallel architectures will continue to increase.

Although one of the great success for modular system design has been the ability to keep software running whilst the underlying hardware platforms have gone through several generations of upgrades. This has been achieved by freezing the software, even to the extent of preserving binaries for years. In practise this is not sufficient and software that is in use is under continual and very expensive maintenance. There is a universal need for software to adapt.

13 Conclusions

Up to Intel's Pentium, Moore's Law [Moore, 1965] had applied not only to the doubling of number of transistors but also to exponential rises in clock speeds. Since 2005 mainstream processor clock speeds have remained fairly much unchanged. However Moore's Law continues to apply to the exponential rise in the number of available logic circuits. This has driven the continuing rise of parallel multi-core computing. In mainstream computing, GPU computing continues to lead in terms of price v. performance. However GPGPU computing (and parallel computing in general) is still held back by the difficulty of high-performance parallel programming [Langdon, 2011; Merrill *et al.*, 2012].

We have shown genetic programming can be of assistance by splitting the highly skilled tasks of devising algorithms and coding them from coding and tuning high performance applications to perpetually novel hardware. We expect the approach can also be applied to re-configuring software as requirements change as well as adapting it for use with new hardware, unknown when the original code was written.

Correctly tuning one (originally hard coded) constant immediately gave speed ups of between 5% and a factor or 3.6 (median 2.1) (see Table 4). In all cases, where genetic programming was able to run, it was able to build on this. Not only are the newer GPUs faster in themselves but the speed up achieved by GP was also larger on the newer GPUs. With final speed up varying from 5% for the oldest (which was contemporary with the original code) to a factor of more than 6.8 for the newest (median 4.0).

Future new requirements of StereoCamera might be dealing with: colour, moving images (perhaps with time skew), larger images, greater frame rates and running on mobile robots, 3D telephones, virtual reality gamesets or other low energy portable devices. We can hope our GP system could be used to automatically create new versions tailored to new demands and new hardware.

The grammar based genetic programming system is available via `ftp.cs.ucl.ac.uk` file `genetic/gp-code/StereoCamera_1_1.tar.gz` and training images are in StereoImages.tar.gz The improved version of StereoCamera supports CUDA 5 and later. It can be run on nVidia Tesla devices (which lack a graphical interface) by sending graphical output to a file. Earlier versions were only able to display graphical output immediately. The new code is available in `StereoCamera_v1_1c.zip`.

Acknowledgements

I am grateful for the assistance of njuffa, Istvan Reguly, vyas of nVidia, Ted Baker, and Allan MacKinnon.

GPUs were given by nVidia. Funded by EPSRC grant EP/I033688/1.

References

[Archanjo and Von Zuben, 2012] Gabriel A. Archanjo and Fernando J. Von Zuben. Genetic programming for automating the development of data management algorithms in information technology systems. *Advances in Software Engineering*, 2012.

[Cody-Kenny and Barrett, 2013] Brendan Cody-Kenny and Stephen Barrett. The emergence of useful bias in self-focusing genetic programming for software optimisation. In Guenther Ruhe and Yuanyuan Zhang, editors, *Symposium on Search-Based Software Engineering*, volume 8084 of *Lecture Notes in Computer Science*, pages 306–311, Leningrad, August 24-26 2013. Springer. Graduate Student Track.

[Colmenar *et al.*, 2011] J. Manuel Colmenar, Jose L. Risco-Martin, David Atienza, and J. Ignacio Hidalgo. Multi-objective optimization of dynamic memory managers using grammatical evolution. In Natalio Krasnogor, Pier Luca Lanzi, Andries Engelbrecht, David Pelta, Carlos Gershenson, Giovanni Squillero, Alex Freitas, Marylyn Ritchie, Mike Preuss, Christian Gagne, Yew Soon Ong, Guenther Raidl, Marcus Gallager, Jose Lozano, Carlos Coello-Coello, Dario Landa Silva, Nikolaus Hansen, Silja Meyer-Nieberg, Jim Smith, Gus Eiben, Ester Bernado-Mansilla, Will Browne, Lee Spector, Tina Yu, Jeff Clune, Greg Hornby, Man-Leung Wong, Pierre Collet, Steve Gustafson, Jean-Paul Watson, Moshe Sipper, Simon Poulding, Gabriela Ochoa, Marc Schoenauer, Carsten Witt, and Anne Auger, editors, *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1819–1826, Dublin, Ireland, 12-16 July 2011. ACM.

[Cotillon *et al.*, 2012] Alban Cotillon, Philip Valencia, and Raja Jurdak. Android genetic programming framework. In Alberto Moraglio, Sara Silva, Krzysztof Krawiec, Penousal Machado, and Carlos Cotta, editors, *Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012*, volume 7244 of *LNCS*, pages 13–24, Malaga, Spain, 11-13 April 2012. Springer Verlag.

[Deb *et al.*, 2002] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, Apr 2002.

[Harman *et al.*,] Mark Harman, Yue Jia, William B. Langdon, Tim Menzies, and Shin Yoo. ALERTI: adaptive learning to evolve radical tuning iteratively. *International Journal on Software Tools for Technology Transfer*. Invited opinion corner.

[Harman *et al.*, 2012] Mark Harman, William B. Langdon, Yue Jia, David R. White, Andrea Arcuri, and John A. Clark. The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs. In *The 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 12)*, pages 1–14, Essen, Germany, September 3-7 2012. ACM.

[Harman *et al.*, 2013] Mark Harman, William B. Langdon, and Westley Weimer. Genetic programming for reverse engineering. In Rocco Oliveto and Romain Robbes, editors, *20th Working Conference on Reverse Engineering (WCRE 2013)*, Koblenz, Germany, 14-17 October 2013. IEEE. Invited Keynote.

[Jia *et al.*, 2013] Yue Jia, Mark Harman, and Bill Langdon. The GISMOE architecture. In Yan Hu, Xiaochen Lai, Zhilei Ren, and Jifeng Xuan, editors, *2nd Chinese Search Based Software Engineering workshop*, Dalian, China, 8-9 June 2013. Invited keynote.

[Koza, 1992] John R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection*. MIT press, 1992.

[Langdon and Harman,] William B. Langdon and Mark Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*. Accepted.

[Langdon and Harman, 2010] W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In Pilar Sobrevilla, editor, *2010 IEEE World Congress on Computational Intelligence*, pages 2376–2383, Barcelona, 18-23 July 2010. IEEE.

[Langdon and Harman, 2014] W. B. Langdon and M. Harman. Genetically improved CUDA C++ software. In Miguel Nicolau, Krzysztof Krawiec, and Malcolm Heywood, editors, *Proceedings of the 17th European Conference on Genetic Programming, EuroGP 2014*, LNCS, Spain, 23-25 April 2014. Springer Verlag. Forthcoming.

[Langdon *et al.*, 2010] William B. Langdon, Mark Harman, and Yue Jia. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software*, 83(12):2416–2430, December 2010.

[Langdon, 2010] W. B. Langdon. A many threaded CUDA interpreter for genetic programming. In Anna Isabel Esparcia-Alcazar, Aniko Ekart, Sara Silva, Stephen Dignum, and A. Sima Uyar, editors, *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010*, volume 6021 of *LNCS*, pages 146–158, Istanbul, 7-9 April 2010. Springer.

[Langdon, 2011] W. B. Langdon. Graphics processing units and genetic programming: An overview. *Soft Computing*, 15:1657–1669, August 2011.

[Langdon, 2013] W. B. Langdon. Which is faster: Bowtie2GP > Bowtie > Bowtie2 > BWA. In Francisco Luna, editor, *GECCO 2013 Late breaking abstracts workshop*, pages 1741–1742, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[Le Goues *et al.*, 2012] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, January-February 2012.

[Merrill *et al.*, 2012] Duane Merrill, Michael Garland, and Andrew Grimshaw. Policy-based tuning for performance portability and library co-optimization. In *Innovative Parallel Computing (InPar), 2012*. IEEE, May 2012.

[Moore, 1965] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 19 1965.

[Orlov and Sipper, 2011] Michael Orlov and Moshe Sipper. Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, April 2011.

[Owens *et al.*, 2008] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008. Invited paper.

[Petke *et al.*, 2013] Justyna Petke, William B. Langdon, and Mark Harman. Applying genetic improvement to MiniSAT. In Guenther Ruhe and Yuanyuan Zhang, editors, *Symposium on Search-Based Software Engineering*, volume 8084 of *Lecture Notes in Computer Science*, pages 257–262, Leningrad, August 24-26 2013. Springer. Short Papers.

[Petke *et al.*, 2014] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. Using genetic improvement & code transplants to specialise a C++ program to a problem class. In Miguel Nicolau, Krzysztof Krawiec, and Malcolm Heywood, editors, *Proceedings of the 17th European Conference on Genetic Programming, EuroGP 2014*, Spain, 23-25 April 2014. Accepted.

[Poli *et al.*, 2008] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[Sitthi-amorn *et al.*, 2011] Pitchaya Sitthi-amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Transactions on Graphics*, 30(6):article:152, December 2011. Proceedings of ACM SIGGRAPH Asia 2011.

[Stam, 2008] Joe Stam. Stereo imaging with CUDA. Technical report, nVidia, V 0.2 3 Jan 2008.

[Syswerda, 1989] Gilbert Syswerda. Uniform crossover in genetic algorithms. In J. David Schaffer, editor, *Proceedings of the third international conference on Genetic Algorithms*, pages 2–9, George Mason University, 4-7 June 1989. Morgan Kaufmann.

[Tiwari *et al.*, 1994] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, Dec 1994.

[White *et al.*, 2008] David R. White, John Clark, Jeremy Jacob, and Simon M. Poulding. Searching for resource-efficient programs: low-power pseudorandom number generators. In Maarten Keijzer, Giuliano Antoniol, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Nikolaus Hansen, John H. Holmes, Gregory S. Hornby, Daniel Howard, James Kennedy, Sanjeev Kumar, Fernando G. Lobo, Julian Francis Miller, Jason Moore, Frank Neumann, Martin Pelikan, Jordan Pollack, Kumara Sastry, Kenneth Stanley, Adrian Stoica, El-Ghazali Talbi, and Ingo Wegener, editors, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1775–1782, Atlanta, GA, USA, 12-16 July 2008. ACM.

[White *et al.*, 2011] David R. White, Andrea Arcuri, and John A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, August 2011.

A StereoKernel tuned for K20c Tesla

`StereoCamera_v1_1c.zip` contains the following CUDA kernel, in addition to the complete Stereo Camera system.

```
/*******
stereoKernel
Now for the main stereo kernel:  There are four parameters:
disparityPixel points to memory containing the disparity value (d)
for each pixel.
width & height are the image width & height, and out_pitch specifies
the pitch of the output data in words (i.e. the number of floats
between the start of one row and the start of the next.).
disparityMinSSD removed by GP
*********/

__attribute__((global)) void stereoKernel(
  // pointer to the output memory for the disparity map
  float * __restrict__ disparityPixel,
  // the pitch (in pixels) of the output memory for the disparity map
  const size_t out_pitch,
  const int width,
  const int height,
  unsigned int * __restrict__ timer,   //For GP timing only
  int * __restrict__ sm_id             //For GP timing only
)
{
FIXED_init_timings(timer,sm_id);       //For GP timing only
extern __attribute__((shared)) float disparityPixel_S[];

int* const disparityMinSSD = (int*)&disparityPixel_S[ROWSperTHREAD*BLOCK_W];
 // column squared difference functions
int* const col_ssd = &disparityMinSSD[ROWSperTHREAD*BLOCK_W];
float d; // disparity value
float d0,d1;
float dmin;

int diff;    // difference temporary value
int ssd;     // total SSD for a kernel
float x_tex; // texture coordinates for image lookup
float y_tex;
int row;     // the current row in the rolling window
int i;       // for index variable
const int dthreadIdx = threadIdx.x % BLOCK_W;
const int dblockIdx  = threadIdx.x / BLOCK_W;

//bugfix force subsequent calculations to be signed
const int X = (__mul24(blockIdx.x,(BLOCK_W-2*RADIUS_H)) + dthreadIdx);
const int ssdIdx = threadIdx.x;
int* const reduce_ssd = &col_ssd[(BLOCK_W )*dperblock-BLOCK_W];
const int Y = (__mul24(blockIdx.y,ROWSperTHREAD));
```

```
//int extra_read_val = 0; no longer used
//if(dthreadIdx < (2*RADIUS_H)) extra_read_val = BLOCK_W + ssdIdx;

// initialize the memory used for the disparity and the disparity difference
//Uses first group of threads to initialise shared memory
if(threadIdx.x<BLOCK_W-2*RADIUS_H)
if(dblockIdx==0)
if(X<width )
{
  for(i = 0;i<ROWSperTHREAD && Y+i < height;i++)
  {
    // initialize to -1 indicating no match
    disparityPixel_S[i*BLOCK_W +threadIdx.x] = -1.0f;
    //ssd += col_ssd[i+threadIdx.x];
    disparityMinSSD[i*BLOCK_W +threadIdx.x] = MIN_SSD;
  }
}
__syncthreads();

x_tex = X - RADIUS_H;
for(d0 = STEREO_MIND;d0 <= STEREO_MAXD;d0 += STEREO_DISP_STEP*dperblock)
{
  d = d0 + STEREO_DISP_STEP*dblockIdx;
  col_ssd[ssdIdx] = 0;

  // do the first row
  y_tex = Y - RADIUS_V;
  for(i = 0;i <= 2*RADIUS_V;i++)
  {
    diff = readLeft(x_tex,y_tex) - readRight(x_tex-d,y_tex);
    col_ssd[ssdIdx] += SQ(diff);
    y_tex += 1.0f;
  }
  __syncthreads();

  // now accumulate the total
  if(dthreadIdx<BLOCK_W-2*RADIUS_H)
  if(X < width && Y < height)
  {
    ssd = 0;
    for(i = 0;i<=(2*RADIUS_H);i++)
    {
      ssd += col_ssd[i+ssdIdx];
    }
  }
  if(dblockIdx!=0) reduce_ssd[threadIdx.x] = ssd;
  __syncthreads();

  //Use first group of threads to set ssd to smallest SSD for d1<d0+dperblock
  if(threadIdx.x<BLOCK_W-2*RADIUS_H)
  if(X < width && Y < height)
  {
```

```
    dmin = d;
    d1 = d + STEREO_DISP_STEP;
    for(i = threadIdx.x+BLOCK_W;i < blockDim.x;i += BLOCK_W) {
      if(d1 <= STEREO_MAXD && reduce_ssd[i] < ssd) {
        ssd = reduce_ssd[i];
        dmin = d1;
      }
      d1 += STEREO_DISP_STEP;
    }
    //if ssd is smaller update both shared data arrays
    if( ssd < disparityMinSSD[0*BLOCK_W +threadIdx.x])
    {
     disparityPixel_S[0*BLOCK_W +threadIdx.x] = dmin;
     disparityMinSSD[0*BLOCK_W +threadIdx.x] = ssd;
    }
  }
  __syncthreads();

  // now do the remaining rows
  y_tex = Y - RADIUS_V; // this is the row we will remove
  #pragma unroll 11
  for(row = 1;row < ROWSperTHREAD && (row+Y < (height+RADIUS_V));row++)
  {
    // subtract the value of the first row from column sums
    diff = readLeft(x_tex,y_tex) - readRight(x_tex-d,y_tex);
    col_ssd[ssdIdx] -= SQ(diff);

    // add in the value from the next row down
    diff =  readLeft(x_tex,  y_tex + (float)(2*RADIUS_V)+1.0f) -
           readRight(x_tex-d,y_tex + (float)(2*RADIUS_V)+1.0f);
    col_ssd[ssdIdx] += SQ(diff);
    y_tex += 1.0f;
    __syncthreads();

    if(dthreadIdx<BLOCK_W-2*RADIUS_H)
    if(X<width && (Y+row) < height)
    {
      ssd = 0;
      for(i = 0;i<=(2*RADIUS_H);i++)
      {
        ssd += col_ssd[i+ssdIdx];
      }
    }
    if(dblockIdx!=0) reduce_ssd[threadIdx.x] = ssd;
    __syncthreads();

    //Use 1st group threads to set ssd/dmin to smallest SSD for d1<d0+dperblock
    if(threadIdx.x<BLOCK_W-2*RADIUS_H)
    if(dblockIdx==0)
    {
      dmin = d;
      d1 = d + STEREO_DISP_STEP;
```

```
      for(i = threadIdx.x+BLOCK_W;i < blockDim.x;i += BLOCK_W) {
        if(d1 <= STEREO_MAXD && reduce_ssd[i] < ssd) {
          ssd = reduce_ssd[i];
          dmin = d1;
        }
        d1 += STEREO_DISP_STEP;
      }
      //if smaller SSD found update shared memory
      if(ssd < disparityMinSSD[row*BLOCK_W +threadIdx.x])
      {
        disparityPixel_S[row*BLOCK_W +threadIdx.x] = dmin;
        disparityMinSSD[row*BLOCK_W +threadIdx.x] = ssd;
      }
    }//endif first group of thread
  }// for row loop
}// for d0 loop

//Write answer in shared memory to global memory
if(threadIdx.x<BLOCK_W-2*RADIUS_H)
if(dblockIdx==0)
if(X < width) {
#pragma unroll 3
  for(row = 0;row < ROWSperTHREAD && (row+Y < height);row++)
  {
    disparityPixel[__mul24((Y+row),out_pitch)+X] =
                disparityPixel_S[row*BLOCK_W +threadIdx.x];
  }
}
FIXED_report_timings(timer,sm_id);      //For GP timing only
}
```

Comments added by hand. Modifications to the openVidia CUDA Stereo Camera code distributed by SourceForge are described in Section 11 (page 14) etc.