EuroGP-2014, Miguel Nicolau, Krzysztof Krawiec, Malcolm Heywood *Eds.*, LNCS,
Granada, Spain 23-25 April, Springer.

Genetically Improved CUDA C++ Software

W. B. Langdon and M. Harman

W.Langdon@cs.ucl.ac.uk CREST, Department of Computer Science,
University College London Gower Street, London WC1E 6BT, UK

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 Introduction

Genetic Programming (GP) [1] is increasingly being used in Software Engineering [2]. We are using GP to make software more adaptable [3] and are particularly interested in GP to generate code [4; 5], for bug fixing [6] and for improving existing code [7; 8; 9; 10; 11; 12; 13]. 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 [14; 8]. 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

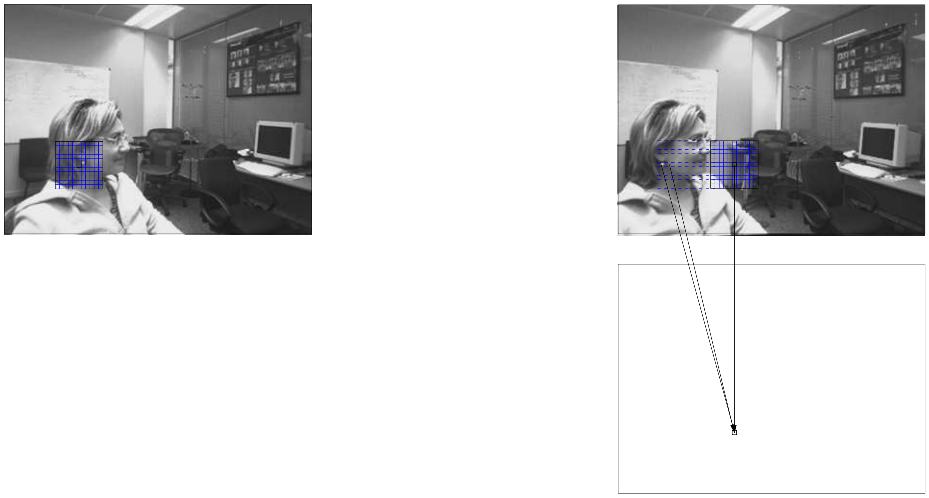

Fig. 1. For each pixel we calculate the sum of squared differences (SSD) between 11×11 regions centred on the pixel in the left image and the same pixel in the right hand image. The right hand 11×11 region is moved one place to the left and new SSD is calculated This is repeated 50 times. Each time a smaller SSD is found, it is saved.

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	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

cards (GPUs). Although originally devised for the consumer market, they are increasingly being used for general purpose computing on GPUs (GPGPU) [15] 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. [16], have advocated writing highly parametrised parallel code which can then be automatically tuned. Unfortunately this throws the load back on to the coder [17]. 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).

2 Source Code: StereoCamera

The StereoCamera system was written by nVidia's stereo image processing expert Joe Stam [18] for the first version of CUDA. V1.0b is available from SourceForge but, despite Moore's Law [19], and except for my bugfix, it has not been updated since 2008.

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 1). 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. SSD is calculated for 0 to 50 displacements and the one with the smallest SSD is reported.

Considerable savings can be made by reducing the total number of calculations by sharing intermediate calculations [18, Fig. 3]. Each SSD calculation 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. 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). 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. Ideally 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. In practise all these factors interact in non-obvious hardware dependent ways.

3 Example Stereo Pairs from Microsoft's I2I Database

Microsoft's I2I database contains 3010 stereo images. Figure 1 (top) is a typical example. Many of these are in the form of movies taken in an office environment. Almost images all are 320×240 pixels. We took 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 8) the improved GPU code is indeed tuned to 320×240 images but still works well on the other I2I stereo pairs.

4 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. 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 5.2, is not initially enabled.) As with [6] and our GISMOE approach [10], after GP has run the best GP individual from the last generation is minimised. 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 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 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.

5 Alternative Implementations

5.1 Avoiding Reusing Threads: XHALO

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. Thus diverting ten threads to deal with the halo causes all the remaining 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.

5.2 Parallel of Discrepancy offsets: DPER

The original code (Section 2) 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 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 saving some I/O.

6 Parameters Accessible to Evolution

The GISMOE GP system [10] 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.

6.1 Fixed Configuration Parameters

OUT_TYPE The return value should be in the range -1 to 50. Originally this is coded as a `float`. OUT_TYPE gives GP the option of trying other types.

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	
STORE_Pixel	GLOBAL	GLOBAL, SHARED, LOCAL	
STORE_MinSSD	GLOBAL	GLOBAL, SHARED, LOCAL	
DPER	disabled		Section 5.2
XHALO	disabled		Section 5.1
__mul24(a,b)	__mul24	__mul24, *	fast 24-bit multiply
GPtexturereadmode	Normalized Float	NormalizedFloat, ElementType, no Textures	Section 6.1
texturefilterMode	Linear	Linear, Point	
textureaddressMode		Clamp, Mirror, Wrap	
texturenormalized		0, 1	

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.
__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.
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.

7 Evolvable Code

Following the standard GISMOE approach [10], 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.

The source code, including XHALO and DPER (Sections 5.1 and 5.2), is automatically translated line by line into a BNF grammar (see Figure 2). 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++

```
<KStereo.cuh_159>  ::= "{\n"
<KStereo.cuh_160>  ::= "" <_KStereo.cuh_160> "\n"
<_KStereo.cuh_160> ::= "init_disparityPixel(X,Y,i);"
<KStereo.cuh_161>  ::= "" <_KStereo.cuh_161> "\n"
<_KStereo.cuh_161> ::= "init_disparityMinSSD(X,Y,i);"
```

Fig. 2. 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 stereoKernel.

source code. There are 55 variable lines, 27 IF and 10 of each of the three parts of C `for` loops. There are also 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. 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.

7.1 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.

7.2 Weights

Normally each line of code is equally likely to be modified. However, only as part of creating the 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.)

7.3 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 [10]. The additional lines of code are not random but are copied from stereoKernel itself. This is like [6] except we use the grammar.

7.4 Crossover

As in the GISMOE frame work [10], crossover creates a new GP individual from two different members of the better half of the current population. The child inherits each of the 12 fixed parameters (Table 2) at random from either parent (uniform crossover [20]). Whereas in [10] 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 [21]. 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.

7.5 Fitness

To avoid over fitting and to keep run times manageable, each generation one of the two hundred training images pairs is chosen [22]. Each GP modified kernel in the population is tested on that image pair.

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× the maximum reasonable value for a good kernel.

Timing Each of the Multiprocessors (MPs) within the GPU chip has its own independent clock. To get a robust timing scheme, each kernel block records both its own start and end times and the MP unit it is running on. 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. The total duration of the kernel is the longest time taken by any of the MPs used.

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.
Population:	Panmictic, non-elitist, generational. 100 members.
Parameters:	Initial population of random single mutants heavily weighted towards the kernel header and shared variables. 50% truncation selection. 50% crossover, 50% mutation. No size limit. 50 generations.

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

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.

If the unmodified kernel did not return a value 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.

7.6 Selection

As with the GISMOE framework [10] 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 7.3) and another by crossover with another selected parent (Section 7.4). The complete GP parameters are summarised in Table 3.

8 Results

Table 4 gives the speed up for six types of GPUs. By reducing `ROWSperTHREAD` from the original 40 to 5, pretuning (Section 4) itself gave considerable speed ups (columns 4-5 in Table 4). However for NVS 290, tuning `ROWSperTHREAD`

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

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.

With CUDA 5.0 memcheck (Section 7.5), 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 [4]. 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 and retuned (Section 4). 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 3. 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 ± columns in Table 4).

8.1 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.

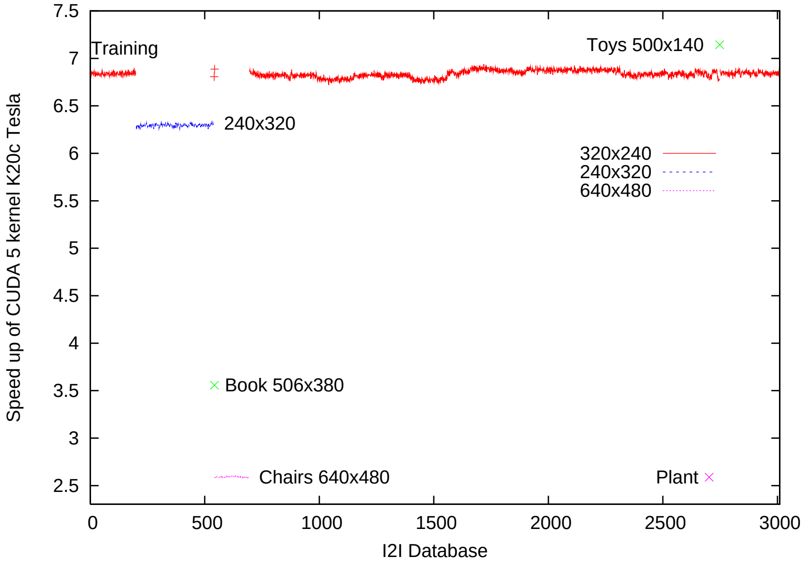

Fig. 3. Performance of GP improved K20c Tesla kernel on all 3010 stereo pairs in Microsoft's I2I database relative to original kernel on the same image pair on the same GPU. Fifty of first 200 pairs used in training. The evolved kernel is always much better, especially on images of the same size and shape as it was trained on.

9 Evolved Tesla K20c CUDA Code

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 4) four configuration parameters were changed and there were six code changes. See Figures 4 and 5.

DPER is enabled and the new kernel calculates two disparity values in parallel, Section 5.2. disparityPixel and disparityMinSSD are stored in shared memory, Section 6.1 and XHALO is enabled, Section 5.1.

The final code changes, Figure 5, are:

- disable volatile, Section 7.
- insert `#pragma unroll 11` before the `for` loop that steps through the `ROWSperTHREAD` - 1 other rows (Section 2).
- insert `#pragma unroll 3` before the `for` loop that writes each of the `ROWS perTHREAD` 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 it gave a kernel whose run time was exactly on the removal threshold.

```
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>
```

Fig. 4. Best GP individual in generation 50 of K20c Tesla run after minimising, Section 4, removed less useful components. (Auto-tuning made no further improvements.)

*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`

Fig. 5. Evolved changes to K20c Tesla StereoKernel. (Produced by GP grammar changes in Figure 4). 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.

– 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 one, 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.
– 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 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()`.

10 Conclusions

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.

Acknowledgements

I am grateful for the assistance of njuffa, Istvan Reguly, vyas of nVidia, Ted Baker, and Allan MacKinnon. The grammar based genetic programming system is available via `ftp.cs.ucl.ac.uk` file `genetic/gp-code/StereoCamera_1_1.tar.gz`

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

References

1. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. `http://www.gp-field-guide.org.uk` (2008)
2. Harman, M., Langdon, W.B., Weimer, W.: Genetic programming for reverse engineering. In: WCRE 2013, Koblenz, Germany, IEEE, Invited Keynote.
3. Harman, M., Langdon, W.B., Jia, Y., White, D.R., Arcuri, A., Clark, J.A.: The GISMOE challenge. In: ASE 2012, Essen, Germany, ACM 1–14
4. Langdon, W.B., Harman, M.: Evolving a CUDA kernel from an nVidia template. In WCCI-2010, Barcelona, IEEE 2376–2383
5. Archanjo, G.A., Von Zuben, F.J.: Genetic programming for automating the development of data management algorithms in information technology systems. Advances in Software Engineering (2012)
6. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: GenProg: A generic method for automatic software repair. IEEE Trans. on Soft. Eng. **38**(1) (2012) 54–72
7. Sitthi-amorn, P., Modly, N., Weimer, W., Lawrence, J.: Genetic programming for shader simplification. ACM Trans. on Graphics **30**(6) (2011) article:152
8. White, D.R., Arcuri, A., Clark, J.A.: Evolutionary improvement of programs. IEEE Trans. on EC **15**(4) (2011) 515–538
9. Orlov, M., Sipper, M.: Flight of the FINCH through the Java wilderness. IEEE Trans. on EC **15**(2) (2011) 166–182
10. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE Trans. on EC, Accepted.
11. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Using genetic improvement & code transplants to specialise a C++ program. In EuroGP 2014
12. Cotillon, A., Valencia, P., Jurdak, R.: Android genetic programming framework. In EuroGP 2012. LNCS 7244, Malaga, Spain, Springer, 13–24
13. Cody-Kenny, B., Barrett, S.: The emergence of useful bias in self-focusing genetic programming for software optimisation. In SSBSE 2013. LNCS 8084. 306–311
14. Tiwari, V., Malik, S., Wolfe, A.: Power analysis of embedded software: A first step towards software power minimization. IEEE Trans. on VLSI **2**(4) (1994) 437–445
15. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proceedings of the IEEE **96**(5) (2008) 879–899 Invited paper.
16. Merrill, D., Garland, M., Grimshaw, A.: Policy-based tuning for performance portability and library co-optimization. In: InPar, 2012, IEEE
17. Langdon, W.B.: Graphics processing units and genetic programming: An overview. Soft Computing **15** (2011) 1657–1669
18. Stam, J.: Stereo imaging with CUDA. Technical report, nVidia (2008)
19. Moore, G.E.: Cramming more components onto integrated circuits. Electronics **38**(8) (1965) 114–117
20. Syswerda, G.: Uniform crossover in genetic algorithms. In ICGA-1989, 2–9
21. Koza, J.R.: Genetic Programming. MIT press (1992)
22. Langdon, W.B.: A many threaded CUDA interpreter for genetic programming. In EuroGP 2010. LNCS 6021., Istanbul, Springer 146–158