# Graphics Processing Units for Trajectory Planning of Kinematically Redundant Manipulators

Brendan C. Wood[1], Juan A. Carretero[2]

[1] *Department of Mechanical Engineering, University of New Brunswick, b.wood@unb.ca*

[2] *Department of Mechanical Engineering, University of New Brunswick, Juan.Carretero@unb.ca*

## Abstract

GPU programming is introduced with specific attention to its use in engineering computation. GPU programming is put in context against traditional sequential programming and parallel programming strategies. One practical example of GPU programming in engineering is provided, with a discussion of the achieved speedups and suggestions for how to increase speedup further. A general approach for parallelising engineering problems is presented, and guidelines for maximising performance are given.

**Keywords:** GPU computing, general purpose graphics computing, parallel computing, engineering design, engineering analysis, path planning, kinematic redundancy.

## Processeurs Graphiques Utilisés pour la Planification des Trajectoires des Manipulateurs Redondents

## Résumé

La programmation sur des processeurs graphiques (GPU) est présenté avec une attention particulière à son utilisation dans les calculs d'ingénierie. La programmation sur les GPU est comparée à la programmation séquentielle et les stratégies traditionnelles de programmation parallèle. Un exemple pratique de la programmation sur les GPU en génie est fourni, avec une discussion sur les accélérations obtenues. Quelques suggestions pour augmenter des vitesses computationelles supplémentaire sont aussi fournies. Une approche générale pour aider la formulation en parallèle des problèmes d'ingénierie est présentée, et règles générales pour optimiser leur performance sont donnés.

**Mots-clé:** processeurs graphiques, calcul générique sur processeurs graphiques, calcul parallèl, conception en génie, analyse en génie, planification des trajectoires, redondance cinématique

## 1 INTRODUCTION

As modern engineering problems are becoming more complex, computer-aided engineering is becoming increasingly ubiquitous and it is more important for engineers to use computational tools efficiently and effectively. Engineering simulation and optimisation techniques are computationally intensive, which typically become more expensive with increasing precision or problem size.

Maximising computational performance is a difficult optimisation problem itself. Recently, new tools for improving performance have become available – namely, graphics processing units (GPUs). Using these new tools, however, is not easy for engineers who have relatively little background in computer programming. In fact, if one is only familiar with traditional programming methods, then GPU programming will require a complete paradigm shift.

The purpose of this paper is to introduce GPU programming from an engineering perspective and place GPUs in a historical context with respect to more traditional computational methods. We then present an example of an engineering problem that requires a large amount of computation to solve, and report on efforts to parallelise[3] its computation for use on a GPU. In particular, the example in this paper is overall path planning for a 3-$\underline{RP}$RR manipulator.

The rest of the paper is organised as follows. Section 2 provides an introduction to engineering computation, discussing traditional software practices, the advent of parallel computing, and finally how GPUs fit into the picture. Section 3 presents the overall motion planning problem, which exhibits a high degree of data parallelism, and discusses two strategies for its implementation on a GPU. Section 4 considers the results of the two implementation strategies, explains the differences between them, and presents a general approach that can be taken to attempt parallelisation of other problems. Finally, Section 5 provides conclusions and outlines potential future directions for investigating the implementation of engineering simulation problems on GPUs.

## 2 ENGINEERING COMPUTATION

Computational fluid dynamics (CFD) and finite element analysis (FEA) are two of the most common and well-known simulation techniques used in mechanical engineering, and also have a very large computational cost. These methods involve solving large systems of equations simultaneously, which is a major component of the computational cost. Another computationally intensive method typically used in engineering design and analysis is numerical optimisation. Optimisation algorithms attempt to minimise or maximise a particular objective function by finding the optimal combination of values for its variables. For objective functions that are expensive to evaluate (requiring a complete system simulation, for example), these optimisation algorithms run into the same computational cost problems as large simulations.

There are four ways in which one can increase the speed of a computation.

1. Reformulate the problem with simplifying assumptions or a less computationally-intensive model.

2. Reprogram the software in a lower-level language and a more efficient manner.

[3]To parallelise a problem is to reformulate it such that there are several independent sub-problems that can be solved in isolation of each other. Describing a problem in this way allows for multiple agents to work on the problem simultaneously without significant overlap. This word, and all its variations in this paper, refer to this concept.

3. Parallelise the algorithm so that multiple systems can share the workload simultaneously.

4. Wait for microprocessor technology to advance and produce faster processors.

Reformulation is not always feasible if the model has already been validated or is the only way in which to achieve a particular level of precision and accuracy. Reprogramming more efficiently or in a faster language is an excellent option, but will not yield much improvement if the software is already well-coded or written in a low-level language such as C.

The truth is that there is no magic bullet for increasing performance, but considerable gains can be made through combinations of the four approaches mentioned above. Of these four steps, parallelisation is the only method that theoretically allows for unlimited speedup in the short-term. As long as the algorithm is appropriately parallelisable, the attainable speedup is only bounded by the resources one is willing to devote to building a parallel computer.

### 2.1 Traditional Approach

Traditional programming is limited to sequential algorithms. Instructions are executed individually in a specified order, even if subsequent instructions are independent of previous. The sequential orientation of traditional programs is due to the way that traditional processors worked. Each processor contained only a single processing core, which was only capable of executing a single instruction at a time.

Improving performance of sequential programs can be achieved through steps 1, 2, and 4 as outlined in Section 2. Once steps 1 and 2 have been exploited, the only traditional option is to wait for more advanced processors. At this point, it is no longer possible to scale to larger problems without devoting more time or sacrificing precision.

### 2.2 Parallel Computing

As stated in [1], "the basic driving force for parallel processing is the desire and prospect for greater performance." This is achieved through division of the workload in a manner such that several processors or systems can work together on the same problem simultaneously. For a highly parallelisable algorithm, this allows for enormous speedup dependent on the number of processors available. This speedup can be used to either reduce the overall computation time or tackle larger problems.

In the past, multi-processor systems have been characterised by large computer clusters with hundreds or even thousands of independent processing units linked together by a communication network to co-ordinate workflow [2]. As with standard single-core processors, performance improvements were driven by the development of faster processors by microprocessor manufacturers, such as Intel, AMD, and Motorola.

More recently, however, microprocessor manufacturers have run into physical limitations related to energy consumption and heat dissipation for individual processing cores, and such dramatic increases in performance over time as characterised by Moore's Law are no longer possible for individual processor cores [3]. Manufacturers have instead begun placing multiple processing cores on individual processors. These cores operate independently of one another and special programming techniques are required to utilise them more effectively. In many cases, it isn't even

feasible to divide a program's execution across multiple cores.

It is beyond the scope of this paper to discuss the foundations of parallel computing in-depth, but the interested reader may refer to [2].

### 2.3 Graphics Processing Units

A graphics processing unit (GPU) is a type of processor that specialises in high-throughput computations that exploit data parallelism. Due to their specialised design, GPUs are theoretically capable of achieving significantly higher performance than their central processing unit (CPU) counterparts. According to Kirk and Hwu in [4], the peak floating point performance for GPUs can be ten times higher than that of multi-core CPUs. To be clear, this is an overly simplified comparison and does not necessarily translate into real-world results, but it still demonstrates the potential of GPUs. On the other hand, there has also been recent work to characterise the existence of a GPU-CPU performance gap as a myth [5].

GPUs are really a type of parallel processing unit that can have up to hundreds of processor cores, which can be programmed to work together on a problem. However, parallel programming is significantly more difficult than traditional sequential programming and it's not always possible to parallelise a particular algorithm [1]. Maximum speedup can be estimated by Amdahl's Law [6], which is dependent on the ratio of the size of the parallelisable portion of the program to the necessarily sequential part. There are several types of parallelism which may be exploited in parallel programming, but GPUs are best suited to data parallelism, in which a large number of identical independent calculations are to be performed on different elements of one or more data sets. In particular, GPUs are designed to leverage "massive" data parallelism, involving thousands or even millions of identical independent calculations [7]. The processing cores in modern GPUs are organised into groups called streaming multiprocessors, or SM units for short. On the NVIDIA Fermi architecture, each SM unit contains 32 cores.

Due to the nature of data parallelism, some problems are well-suited for GPU acceleration, while many others are not. Even some problems which exhibit perfect data parallelism may not be worthwhile parallelising on a GPU. In engineering simulation and optimisation, however, there are often opportunities to exploit data parallelism. Monte-Carlo methods, evolutionary algorithms, finite element modelling, computational fluid dynamics, and molecular dynamics are all excellent candidates for parallelisation.

One important point to make about GPUs is that they specialise in single-precision floating point operations, rather than the double-precision commonly used on CPUs. If high precision is important in calculations (as it often is with engineering simulation work), it is possible to use double-precision on GPUs, but performance is approximately halved on the latest architecture. GPUs specialise in single precision due to their typical use in computer graphics and gaming, in which high precision is not usually required [4]. For a more detailed look at GPUs, Owens *et al.* present a comprehensive introduction in [8].

## 3 EXAMPLE APPLICATION

This section presents the problem of motion planning for a kinematically redundant planar parallel manipulator (the 3-RPRR manipulator), which is used as an example of the parallelisation process.

The purpose of this example is to provide a practical demonstration of the concepts covered in this paper, and motivate discussion of the difficulties associated with parallelisation.

### 3.1 Hardware and API

There are several application programming interfaces (APIs) that can be used for programming modern GPUs, but they are all similar from conceptual, functionality, and performance standpoints. They are all implemented in C/C++ or an extended version thereof. For this research, CUDA (the API from NVIDIA) was chosen due to its maturity and availability of documentation. It should be noted that these parallelisation examples are still very basic in nature and do not yet exploit some of the more advanced optimisation techniques on GPUs that can be used to further increase performance. For example, this software makes no use of shared memory on the SM units nor does it coalesce global memory accesses or manually unroll loops [9].

Each of the problems below are implemented in C on the CPU as a performance baseline, and acceleration is attempted using CUDA on the GPU. The specifications for the test system are as follows:

- AMD Athlon 64 X2 5000+
- 4GB DDR RAM
- NVIDIA GeForce GTX 465 at 1.25GHz, 1GB RAM (352 processing cores)
- Ubuntu Linux version 10.10
- NVIDIA display driver version 260.19.29

It is important to note that, at the time of this writing, the GPU is relatively new and on the medium to high-end of GPUs while the CPU is already is in the low to mid-end of the performance spectrum [10].

### 3.2 Overall Motion Planning for a 3-RPRR Manipulator

The 3-RPRR parallel manipulator is a planar kinematically redundant parallel mechanism proposed by Ebrahimi *et al.* [11] inspired by the work of Gosselin *et al.* in modelling the non-redundant 3-PRR [12]. Due to the kinematic redundancy, there are infinite solutions to the inverse displacement problem for most points within the reachable workspace. This redundancy allows for optimisation of joint trajectories based on a particular criterion, such as obstacle avoidance [13] or kinematic singularity avoidance [11]. As shown by Carretero *et al.* [14], this optimisation can be achieved via a differential evolution technique [15] and an appropriately formulated objective function. The criterion used here is singularity avoidance by attempting to maximise the normalised scaled incircle radius (NSIR) along an entire trajectory defined by a set of control points.

The objective function is quite large, with each evaluation requiring the inverse displacement problem to be solved a number of times, and the NSIR (non-dimensional measure of proximity to kinematic singularities [11]) calculation requiring determinants of matrices, LU decomposition of matrices, and matrix rank computation, among other things. The complete code for evaluating the

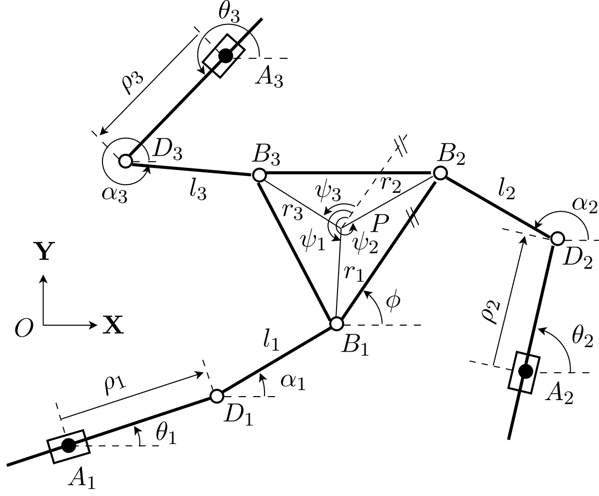


Figure 1: The 3-<u>RP</u>RR planar 6-DOF kinematically redundant parallel manipulator [11].

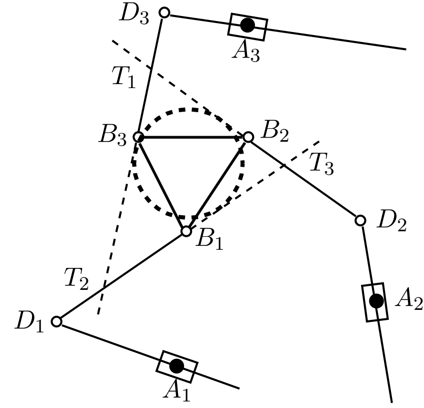


Figure 2: Incircle of the triangle created by the collinear lines passing through the distal links of the 3-<u>RP</u>RR manipulator [11].

objective function in C is approximately 1200 lines in length, containing many loops and function calls. This is not necessarily a true indication of complexity (total floating point operations would be more appropriate), but it provides a sense of scale for this example.

### 3.2.1 Architecture and Kinematics

The three degrees of kinematic redundancy (DOKR) provided by the 3-<u>RP</u>RR allows the manipulator to avoid all direct kinematic singularities, while also improving dexterity and increasing the size of the reachable and dexterous workspaces [11]. The architecture can be seen in Figure 1.

Ebrahimi *et al.* discuss the kinematics in detail in [11]. The inverse displacement problem is treated by modelling the branches with vector loop equations. The vector loop equation for branch $i$ can be written as:

$$\overline{D_iB_i} = \overline{D_iA_i} + \overline{A_iO} + \overline{OP} + \overline{PB_i} \tag{1}$$

$$l_i^2 = (-\rho_i c_{\theta_i} - x_{A_i} + x_p + r_i c_{(\phi+\psi_i)})^2 + (-\rho_i s_{\theta_i} - y_{A_i} + y_P + r_i s_{(\phi+\psi_i)})^2 \tag{2}$$

$$l_i^2 = x_{l_i}^2 + y_{l_i}^2 = (l_i c_{\alpha_i})^2 + (l_i s_{\alpha_i})^2 \tag{3}$$

where $c_\measuredangle$ and $s_\measuredangle$ represent $\cos\measuredangle$ and $\sin\measuredangle$, respectively, while $x_*$ and $y_*$ represent the $x$ and $y$ coordinates (or components) of point (or vector) $*$. All other parameters are defined in Figure 1.

Ebrahimi *et al.* develop the Jacobian matrices $\mathbf{J_x}$ and $\mathbf{J_q}$ to model the direct and inverse kinematic singularities, as follows:

$$\mathbf{J_x} = \begin{bmatrix} l_1c_{\alpha_1} & l_1s_{\alpha_1} & l_1r_1s_{(\alpha_1-\phi-\psi_1)} \\ l_2c_{\alpha_2} & l_2s_{\alpha_2} & l_2r_2s_{(\alpha_2-\phi-\psi_2)} \\ l_3c_{\alpha_3} & l_3s_{\alpha_3} & l_3r_3s_{(\alpha_3-\phi-\psi_3)} \end{bmatrix}_{3\times3} \tag{4}$$

$$\mathbf{J_q} = \begin{bmatrix} u_1 & v_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & u_2 & v_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & u_3 & v_3 \end{bmatrix}_{3\times6} \tag{5}$$

where $u_i = c_{\theta_i} a_{i1} + s_{\theta_i} a_{i2}$ and $v_i = -\rho_i s_{\theta_i} a_{i1} + \rho_i c_{\theta_i} a_{i2}$ with $a_{i1} = x_p - x_{A_i} - \rho_i c_{\theta_1} + r_i c_{(\phi+\psi_i)}$ and $a_{i2} = y_p - y_{A_i} - \rho_i s_{\theta_1} + r_i s_{(\phi+\psi_i)}$.

#### 3.2.2 Proximity to Singular Configurations

Proximity to a singular configuration can be measured using the normalised scaled incircle radius (NSIR) [11]. As explained below, this measure accounts for proximity to direct and inverse kinematic singularities.

In the 3-RPRR manipulator, direct kinematic singularities occur when lines collinear to the distal links of the mechanism meet at a common point. When they do not meet at a common point (and are also not parallel), the intersections of the collinear lines form a triangle, within which a circle is inscribed. A diagram depicting this can be seen in Figure 2. As the radius $r$ of the incircle approaches zero, the mechanism approaches a direct kinematic singularity. Conversely, the incircle is at its largest $r_{\text{max}}$ when it passes through all three attachment points of the distal links to the end-effector (*i.e.*, point $B_i$), so it is possible to normalise the size of the incircle to be independent of manipulator size and frame or reference:

$$r_{\text{norm}} = \frac{r}{r_{\text{max}}} \tag{6}$$

Proximity to inverse kinematic singularities can be measured in terms of the magnitude of the inverse Jacobian. As the incircle, this can be bound by its maximum value. That is:

$$\sqrt{\left|\mathbf{J_q}\mathbf{J_q}^T\right|} = \prod_{i=1}^{3} l_i \sqrt{(c_{(\alpha_i-\theta_i)})^2 + \left(\frac{\rho_i}{\rho_{\text{max}}} s_{(\alpha_i-\theta_i)}\right)^2} \tag{7}$$

$$\sqrt{\left|\mathbf{J_q}\mathbf{J_q}^T\right|_{\text{max}}} = \prod_{i=1}^{3} l_i \tag{8}$$

Therefore,

$$\xi = \sqrt{\frac{\left|\mathbf{J_q}\mathbf{J_q}^T\right|}{\left|\mathbf{J_q}\mathbf{J_q}^T\right|_{\text{max}}}} \tag{9}$$

is a quantity between zero and one that shows the proximity to inverse kinematic singularities.

Combining the measures in equations (6) and (9) one obtains

$$\mathcal{N} = \xi r_{\text{norm}} \tag{10}$$

which is the normalised scaled incircle radius (NSIR), a non-dimensional quantity bound between zero and one.

#### 3.2.3 Algorithm

For a given trajectory of the 3-RPRR end-effector in Cartesian space, a corresponding trajectory must be generated in the joint space for each of the active joints. Due to the three DOKR of the 3-RPRR, there are infinite solutions to the inverse displacement problem for every position along the end-effector trajectory. Therefore, the trajectory can be optimised for a particular criterion.

Carretero *et al.* [14] describe an algorithm to find appropriate trajectories in the joint space, as follows:

1. Find the locus of inverse displacement solutions for the given trajectory for each actuated joint. In this case, there are two actuators per kinematically redundant limb (KRL).
2. From the locus of inverse displacement solutions for each end-effector pose, determine the range of motion of all actuators for each KRL.
3. Select one of the actuators of each KRL as the optimisation variable. In this case, the prismatic joint $\rho_i$ for each KRL is selected.
4. Choose an initial set of values for the $n + 1$ control points (CPs) for each KRL.
5. Connect the $n + 1$ CPs with $n$ line segments to create the polyline approximation of the actuator trajectory.
6. Verify if any point along the polyline violates the actuator boundaries. If so, replace the infeasible actuator values with the corresponding boundary values.
7. Discretise each line segment of the polyline into $N$ points and calculate the profile for the optimised actuator of each KRL, and then calculate the other actuators' profiles.
8. Compute an objective function at each discretised point along the entire trajectory. In this case, the objective function to minimise is the proximity to singular configurations.
9. Using an optimisation algorithm, find the optimal location of the $n$ CPs. Steps 5 through 8 are repeated every time the objective function is calculated. In this case, differential evolution is the optimisation algorithm used to determine the optimal location of the control points.

This algorithm is parallelisable in several aspects. Differential evolution is a population-based global search metaheuristic, in which the individual mutation, crossover, and objective function evaluations are independent calculations which can be performed in parallel. Generations, however, must be executed in a serial fashion because each offspring population is formed by recombination of the parent population. At the population level, this is an example of "embarrassing parallelism". Using this scheme, a population of $n$ candidate solutions can be broken into $n$ identical independent calculations to be run in parallel for each generation. The objective function (described in Section 3.2.4) itself is also parallelisable, so it is possible to reformulate the objective function to expose more degrees of parallelism.

For this example, the differential evolution is parallelised at two levels: that of the population, and that of the objective function itself. The formulation and performance of these two approaches are compared in Sections 3.2.5–3.2.7.

#### 3.2.4 Objective Function

Motion planning for a given Cartesian trajectory of the end-effector is optimised for singularity avoidance. For a coarse trajectory of control points in the joint space that allow the end-effector to follow the given Cartesian trajectory, a number of intermediate via-points are interpolated. The NSIR $\mathcal{N}$ is computed at each of these via-points, and are finally reduced to a single scalar value for use as fitness in the differential evolution algorithm.

As mentioned above, the NSIR must be calculated for each discretised point along the trajectory, and these are combined to compute an overall metric for the entire trajectory. This is computed as follows:

$$F = \bar{\mathcal{N}}\mathcal{N}_{\text{min}} \tag{11}$$

where $\bar{\mathcal{N}}$ is the average value of $\mathcal{N}$ for the entire trajectory, $\mathcal{N}_{\text{min}}$ is the minimum value, and $F$ is the final value of the objective function for the candidate trajectory. This encourages the trajectory to have a high $\mathcal{N}$ on average, while also ensuring that no individual via-point on the trajectory is too close to a singularity.

#### 3.2.5 Population-level Parallelism

At the population level, the number of threads is equal to the number of candidate solutions in the population. Large populations in differential evolution correspond to larger population inertia, which result in slower population convergence, which is not always desirable. In this case, Carretero *et al.* have shown that a population of size 100 will frequently converge to the global solution solution after only a few hundred generations, so a population of several thousand will likely make convergence more difficult. This is too few threads to take full advantage of a GPU with multiple SM units, so the program has been written to execute on only a single SM unit out of the available eleven (the GTX 465 has a total of 352 cores). Therefore, if there were reason enough to scale the population size to use all SM units, or reconfigure the program to evolve eleven distinct populations independently, the run-time would be roughly the same.

#### 3.2.6 Objective Function Parallelism

The objective function is itself parallelisable due to the NSIR calculations along each candidate trajectory. In the present case, 21 control points are used for each leg, which was the number at which Carretero *et al.* reported no further increase in solution quality. The 21 control points are linearly interpolated to provide further precision in calculating the NSIR over the trajectory. This interpolation results in 121 via-points for the entire trajectory, which corresponds to 121 NSIR computations. These computations are independent of one another and form the bulk of the work in the objective function. Therefore, the objective function evaluations for each generation can be split into $121 \times n$ threads, where $n$ is the number of candidate solutions in the population. This provides more than enough threads to satisfy all SM units of the GPU at one time.

The evolution parameters are as follows: population size of 128, 1000 generations as a stopping criterion, 63 design variables, crossover ratio of 0.7, and a mutation scaling factor of 2.0.

#### 3.2.7 Results

Carretero *et al.*'s original version of this algorithm was coded in Matlab. On the test computer used for these examples, it takes approximately 140 minutes to evolve through 1500 generations with a population size of 100. This is far too slow for planning a manipulator trajectory, so the algorithm was ported to a lower level language, namely C, and later parallelised for the GPU in an attempt to speed up the process.

The C version of the algorithm running on the CPU provided a speedup of 72 over the Matlab code. This is not very surprising, considering the significant overhead associated with interpreted languages versus compiled languages and Matlab's pass-by-value approach to memory manage-

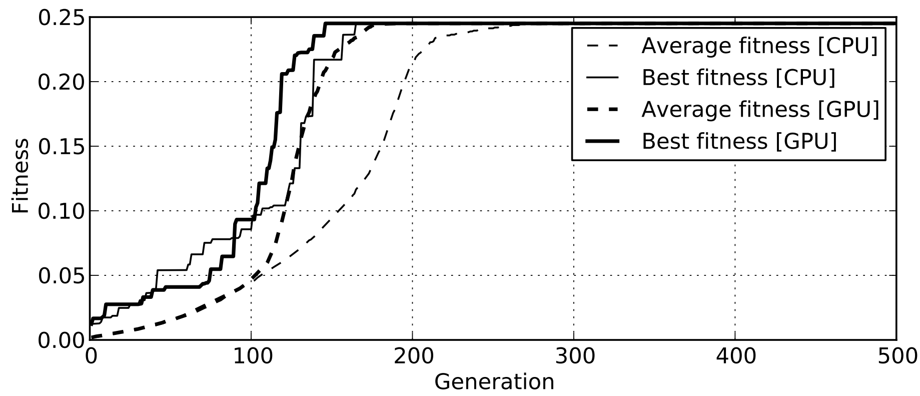

Figure 3: Best and average of CPU and GPU populations by generation

ment. The speedup of interest is that of the GPU with respect to the CPU version written in C. Using only the population-level parallelism on one SM unit (*i.e.*, only 32 out of 352 on the GTX 465), the GPU achieves a speedup of 4.2, without including any advanced GPU optimisation techniques. Parallelizing the objective function to take advantage of all SM units increases the speedup to 10.5.

As is shown in Figure 3, both the GPU and CPU converge to the same global solution, but the GPU optimisation converges about 100 generations earlier than the CPU. Both execute the exact same algorithm and use precisely the same pseudo-random generator, and the difference seen here is due to differing initial seed. The CPU could have just as easily found the solution earlier than the GPU.

To test the assertion that the GPU should be able to scale to a population eleven times larger with very little change in run-time, a population of size 1408 is evolved on the GPU and its run-time is compared to the original population. Though the amount of computational work has increased eleven-fold, the run-time has only increased by a factor of 1.3. Even with 1408 threads, the GPU still is not operating at its full potential.

## 4 DISCUSSION

Using a finer grain of parallelism resulted in higher performance for the example in Section 3; the difference is that more threads allowed a more complete utilisation of the GPU. Though the population-level implementation used only one out of eleven SM units, the parallelised objective function did not achieve eleven times the performance. This is because the division of work for splitting the objective function was not perfectly parallel, and required a larger memory usage. There are more variables to consider than the number of threads, though that is arguably one of the most important parameters.

As GPUs become more powerful and acquire more processing cores in the coming years, this problem will become more important because more threads will be needed at any given time to saturate the GPU. In the interests of developing scalable software that will benefit from advances in GPU technology, it would be prudent to consider algorithms that have the potential to generate hundreds of thousands of threads, or more. Simulation problems that produce more useful results as a function of problem size are excellent candidates, because they will scale more easily with GPUs and there will always be a reason to scale further and generate more threads. For example,

a contact dynamics simulation could produce more threads by modelling more bodies simultaneously, while a fluid dynamics simulation could increase precision or include more realistic effects.

This paper has just scratched the surface of GPU computing, and all techniques presented are just the first steps towards maximising performance of GPUs. A quick summary is as follows:

1. Choose an algorithm to consider parallelising.
2. Analyse the algorithm for repetitive calculations, such as loops or matrix operations.
3. Estimate how many identical independent calculations need to be performed for typical use.
4. Consider the amount of work involved in each independent calculation.
5. Based on Steps 3 and 4 above, is there enough parallelism to justify implementing?
6. If so, begin implementation and try to find a balance between number of threads and work assigned per thread.
7. Test the GPU implementation against equivalent C code on the CPU. Is there good speedup?
8. Play with the execution configuration to find a good combination of parameters.
9. If further optimisations are desired, consider making use of shared memory in the kernel, coalescing global memory accesses, increasing data locality, changing register usage, and looking for other ways to improve performance.

The last step may sound simple, but in practice its implementation can be difficult as current APIs do not do this automatically. There are trade-offs at every turn, where changing one parameters for the better may negatively affect others. Finding an excellent execution configuration is a significant optimisation problem itself, and there is a bit of an art to it. Hands-on experience will make it easier.

## 5 CONCLUSIONS

Using GPUs for engineering computation is a promising avenue that has the potential to accelerate the pace of engineering. For highly parallelisable algorithms, GPUs offer the possibility of massive scalability, beyond the abilities of a single sequential processor. The computer industry is rapidly shifting from multi-core (*i.e.*, less than a dozen cores) processors to many-core (*i.e.*, hundreds of cores), and the old practices in engineering computation are becoming obsolete.

Future work will involve investigating methods of fine-tuning parallel algorithms on GPUs, and developing heuristic rules to ease their application. Since fine-tuning is such a difficult problem, the feasibility of automating parts of the process will also be studied.

This paper has presented a short introduction to parallel programming on GPUs, with practical engineering examples to solidify concepts. The key point to take away is that producing good parallel software is significantly more difficult than producing sequential software, and should be approached intelligently. It is comparatively easy to produce a first version of a parallel algorithm on a GPU, but it is much more difficult to effectively use all the processing power that GPUs provide.

## REFERENCES


[1] G. S. Almasi and A. Gottlieb. *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[2] Thomas Rauber and Gudula Rünger. *Parallel Programming for Multicore and Cluster Systems*. Springer, 2010.

[3] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005.

[4] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010.

[5] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38:451–460, June 2010.

[6] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[7] Michael Garland and David B. Kirk. Understanding throughput-oriented architectures. *Commun. ACM*, 53:58–66, November 2010.

[8] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[9] NVIDIA. *CUDA C Best Practices Guide*, version 3.2 edition, 2010.

[10] PassMark Software. PC benchmark and test software. http://www.cpubenchmark.net/, last accessed: January 15, 2011.

[11] I. Ebrahimi, J. A. Carretero, and R. Boudreau. Kinematic analysis and path planning of a new kinematically redundant planar parallel manipulator. *Robotica*, 26(3):405–413, May 2008.

[12] C.M. Gosselin, S. Lemieux, and J.-P. Merlet. A new architecture of planar three-degree-of-freedom parallel manipulator. In *Proc. of Int. Conf. on Robotics and Automation, 1996*, V. 4, pp. 3738 –3743.

[13] Y. Nakamura. *Advanced robotics: redundancy and optimization*. Addison-Wesley, Reading, Massachusetts, USA, 1991.

[14] J. A. Carretero, I. Ebrahimi, and R. Boudreau. A new motion planning strategy for kinematically redundant parallel manipulators. In *Proc. of the 2008 CCToMM Symposium on Mechanisms, Machines, and Mechatronics / 2008 CSME Forum*, Ottawa, May 21-23 2008.

[15] R. Storn and K. Price. Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. *J. of Global Optimization*, 11(4):341–359, Dec. 1997.