# High-Performance Dynamic Quantum Clustering on Graphics Processors

Peter Wittek*

*Swedish School of Library and Information Science, University of Borås, Borås, Sweden*

**Abstract**

Clustering methods in machine learning may benefit from borrowing metaphors from physics. Dynamic quantum clustering associates a Gaussian wave packet with the multidimensional data points and regards them as eigenfunctions of the Schrödinger equation. The clustering structure emerges by letting the system evolve and the visual nature of the algorithm has been shown to be useful in a range of applications. Furthermore, the method only uses matrix operations, which readily lend themselves to parallelization. In this paper, we develop an implementation on graphics hardware and investigate how this approach can accelerate the computations. We achieve a speedup of up to two magnitudes over a multicore CPU implementation, which proves that quantum-like methods and acceleration by graphics processing units have a great relevance to machine learning.

## 1. Introduction

Grouping similar objects together is one of the fundamental problems in statistical learning. Many approaches rely on distance metrics between the objects in an $d$-dimensional Euclidean space. The distance metric can be generalized further, and embedding the data points in an $L_2$ space of square integrable functions seems to have advantages in certain applications. This is also known as kernel density estimation. An unsupervised clustering method borrows the metaphor of super-paramagnetic phase from statistical physics, introducing Pott functions around the data points [7]. Clusters

*Corresponding address: Swedish School of Library and Information Science, University of Borås, Allegatan 1, Borås, S-501 90, Sweden were identified correctly in particularly problematic data sets. Taking this method a step further, relying on a quantum framework, Horn and Gottlieb introduced a Parzen window estimator, technically a Gaussian wave packet over data points. Viewing these as eigenfunctions of the time-independent Schrödinger equation, the authors found the optimal clusters by minimizing the potential [16]. This method is sensitive to the variance of the Gaussian, which is considered a parameter in the model. The method was later extended to estimate the parameter from distributions of K-nearest neighbours statistics [22]. Newtonian clustering uses a similar Parzen window estimator borrowing a metaphor from classical physics and modelling emergent clustering structure similar to molecular dynamics [8].

Dynamic quantum clustering (DQC) extends quantum clustering to include time evolution of the Hamiltonian of the Schrödinger equation [33]. This approach approximates potential energy of the Hamiltonian, and evolves the system iteratively to identify the clusters. The great advantage of this method is that the steps can be computed with simple linear algebra operations. The resulting evolving cluster structure is similar to flocking methods, which was inspired by biological systems [11], and it is also similar to Newtonian clustering with its pairwise forces [8].

Graphics processing units (GPUs) were originally designed to accelerate computer graphics through massive on-chip parallelism, but they have attracted massive interest where data-parallelism that is inherent in graphics applications is important. For instance, researchers have studied how GPUs can be applied to problem domains such as scientific computing [23, 31, 27] and visual applications [28, 32, 13], and successful applications achieve speedups of around a magnitude over multicore CPU-based implementations. With regard to dynamic approaches to clustering, the flocking method achieved a speedup of over 300x in certain parts of the algorithm [35]. Pairwise distance-based models such as molecular dynamics [4] and the Ising model [24] have also achieved acceleration, although to our knowledge these methods have not been applied to clustering yet.

Metaphors of classical and quantum physics belong to the wider class of algorithms known as computational intelligence. We believe there is more to such metaphors than improved effectiveness, especially when it comes to quantum probability and its geometric framework. QL methods open the way to truly scalable learning models, leading to unprecedented scaling. In this manuscript, we look at how DQC can be accelerated with graphics hardware, and present the results of extensive benchmarks. Certain steps of the algorithm may reach a speedup of two magnitudes.

The rest of this paper is organized as follows. To motivate quantum-inspired methods in machine learning, we discuss a few relevant aspects of computational intelligence (Section 2). We briefly overview the theoretical background of DQC in Section 3, and also overview the basic concepts of general-purpose programming on the GPU that are relevant to the paper (Section 4). Section 5 details the contribution of the paper, our GPU-based implementation of DQC. We discuss our experimental results in Section 6, and finally Section 7 outlines future work and concludes the paper.

## 2. Machine learning and computational intelligence

Machine learning is a field of artificial intelligence that seeks patterns in empirical data without forcing models on it. That is, the approach is data-driven, rather than model driven. A typical example is clustering: given a distance function between data instances, the task is to group similar items together using an iterative algorithm. Another example is fitting a multidimensional function on a set of data points to estimate the generating distribution. These methods are known as Parzen window estimators or kernel density models.

Computational intelligence is a related field that solves optimization problems by nature-inspired computational methodologies. These include swarm intelligence, force-driven methods, evolutionary computing, neural networks, and others. Quantum-like (QL) methods in machine learning are in a way nature inspired, hence they are related to computational intelligence.

QL methods found useful applications in areas where the system is displaying contextual behaviour. In such cases a quantum approach naturally incorporates this behaviour [18, 17]. Apart from contextuality, entanglement is successfully exploited where traditional models of correlation fail [9], and quantum superposition also accounts for unusual results of combining attributes of data instances [3].

QL learning methods crop up in unconnected applications. Arriving from evolutionary computing, there is a quantum version of particle swarm optimization [29]. The particles in a swarm are agents with simple behavioural patterns, each one is associated with a potential solution. Relying on only local information, the quantum variant is able to find the global optimum for the optimization problem in question.

Quantum neural networks exploit the interference [21] and the fuzziness [25] of quantum systems. The self-organizing map (SOM) is a two-dimensional grid of neurons that learns patterns in the data in a context-sensitive fashion [19]. While SOMs are grounded in classical neural networks, their sensitivity to contexts makes them related to QL methods.

Dynamic quantum clustering (DQC) emerged as a direct physical metaphor of evolving quantum particles. DQC evolves the Hamiltonian of a hypothetical quantum system to iteratively find the clusters [33]. This method has been successfully used in information retrieval [15]. This is not surprising, as a growing body of work shows that language technology has the most to benefit from QL models.

The works cited above highlight how the machine learning community may benefit from quantum metaphors, potentially gaining higher accuracy and effectiveness. We believe there is much more to gain. An attractive aspect of quantum theory is the inherent structure which unites geometry and probability theory in one framework. Reasoning and learning in a QL method are described by linear algebra operations. This in turn translates to computational advantages: software libraries of linear algebra routines are always the first to be optimized for emergent hardware. Contemporary high-performance computing (HPC) clusters are often equipped with graphics processing units (GPUs), which are known to accelerate many computations, including linear algebra routines, often by several magnitudes. As pointed out by [5], the overarching goal of the future of HPC should be to make it easy to write programs that execute efficiently on highly parallel computing systems. The metaphors offered by QL methods bring exactly this ease of programming HPC systems to machine learning.

## 3. Dynamic quantum clustering

Quantum clustering associates a Gaussian wave function with each data point $x_i$: $\psi_i(x) = e^{\frac{-(x_i-x)^2}{2\sigma^2}}$. This is essentially a mapping from the $d$-dimensional Euclidean space $\mathbb{R}^d$ to the infinite dimensional Hilbert space of square integrable functions $L_2$. In kernel density estimation, the sum of these functions, $\psi(x) = \sum_i e^{\frac{-(x_i-x)^2}{2\sigma^2}}$ is considered as a probability distribution that could have generated the data points. The local maxima correspond to cluster centres. Quantum clustering takes $\psi$ as the ground solution for the Schrödinger equation:

$$H\psi = (T + V(x))\psi = E_0\psi,$$

where $H$ is the Hamiltonian, $T$ is the kinetic energy, $V$ is the potential energy, and $E_0$ is the ground energy level. DQC evolves the Hamiltonian to identify the clustering structure by tracking the expectation values of the position operator $X$. The time-dependent expectation value of the position operator

$$\langle\psi(t)|x|\psi(t)\rangle = \int \psi(x,t)^* x\psi(x,t)dx$$

satisfies the equation

$$\frac{d^2\langle x(t)\rangle}{dt^2} = \langle\psi(t)|\nabla V(x)|\psi(t)\rangle,$$

according to the Ehrenfest theorem. This means that the expectation values of the position operator obey their corresponding classical equations of motion, that is, the centre of each wave packet rolls towards the nearest minimum of the potential according to Newton's law of motion.

The great advantage of DQC is that the steps involved in the calculations include only basic linear algebra operations, such as matrix multiplication and eigendecomposition. This makes DQC a good candidate for acceleration and large-scale deployment.

## 4. Graphics hardware in general-purpose problems

In order to discuss details concerning the implementation of algorithms on graphics hardware, we summarize some key aspects of the GPU device architecture in this section. Programming for the GPU is very different from general purpose programming on the CPU due to the extremely multi-threaded nature of the device. GPUs rely on the data parallel or single instruction, multiple data (SIMD) programming model. A computation is defined in terms of a sequence of instructions executed on multiple elements of a memory object.

GPU compute devices from different vendors and even from the same vendor have different characteristics (such as the number of compute units, memory bandwidth, clock rate, etc.), but they follow a similar design pattern (Figure 1). GPU compute devices comprise groups of compute units (also called streaming multiprocessors). Every compute unit contains numerous stream cores (also called streaming processors), which are responsible for executing kernels, that is, a small program operating on an independent data stream. Different GPU compute devices have different numbers of stream cores. For example, the ATI Radeon HD 5870 GPU has 20 compute units, each with 16 stream cores, resulting in a total of 320 cores, whereas the NVidia Tesla C2050 has 14 compute units, each with 32 stream cores, resulting in a total of 448 cores.

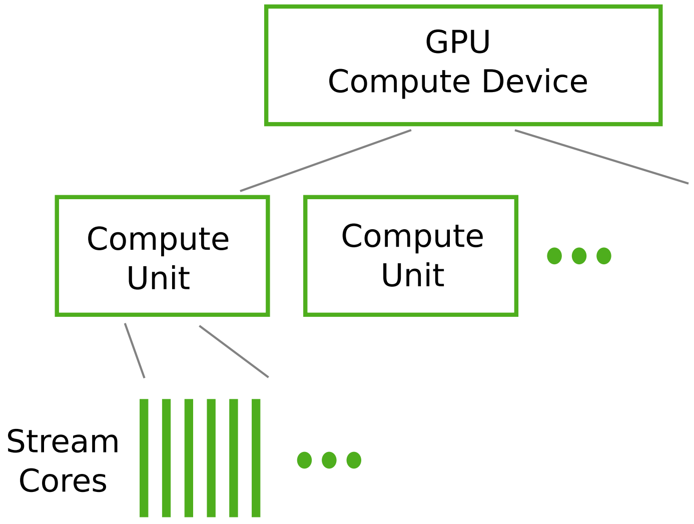


Figure 1: Generalized GPU Compute Device Structure [1]

All stream cores within a compute unit execute the same instruction sequence in lock-step, whereas different compute units can execute different instructions. Unlike CPU cores instructions are issued in order and there is no branch prediction and no speculative execution. Some architectures introduce cache memories, for instance, NVidia's Fermi architecture features a 64 Kbyte of shared L1 cache for each compute unit. Cache is still rare, however, and stream cores and compute units retain an essentially much simpler structure than the cores and the control mechanism of a contemporary CPU. Apart from the automatically managed cache, graphics hardware has a multi-layer memory hierarchy which has to be handled efficiently by the programmer. Each compute unit of the GPU device contains several local 32-bit registers per processor. This memory is shared by all stream cores in a multiprocessor. To accelerate random access to the global memory of the device, constant and texture caches are available. The slower global memory is shared among all compute units and is also accessible by the CPU. The bandwidth between the global memory and the compute units is still much higher than between the CPU and the main memory. To ensure that this bandwidth is saturated when the compute units access the global memory, consecutive memory locations should be fetched to the respective compute units. This is known as coalesced memory access.

When execution starts on a GPU device, many copies of the kernel – known as threads – are enumerated and distributed to the available compute units. These independent threads are organized into blocks which can contain anywhere from 32 to 512 threads each, but all blocks must be of the same size. Optimal performance requires a large number (typically hundreds) of blocks executing in parallel. When blocks are mapped to the hardware, they are organized into warps. A warp is a number of threads (e.g., 32) that can be executed simultaneously on the compute unit. At any given moment, the control unit examines which warps are ready to execute and chooses one. The current instruction of this warp is then executed on the compute unit which then moves on to another warp. In this fashion, the execution of threads on the device is not entirely simultaneous, but interleaved.

Warps enable the device to hide memory access latencies. The global memory on the device has a much higher bandwidth than that of between the RAM and CPU, but it has still a fairly high delay from the time a memory address is requested to the time it becomes available. During the latency of a device memory access, hundreds of arithmetic operations can be performed on a compute unit. Thus, warps which have read their data can be performing computations while other warps running on the same compute unit are waiting for their data to be fetched from the device memory.

Improvements in the programmability of GPUs have made graphics hardware an even more compelling platform for computationally demanding tasks. Compute Unified Device Architecture (CUDA, [2]) is a C-like language allowing the implementation of innovative data-parallel algorithms in general computing terms to solve many non-graphics applications such as database searching and sorting, medical imaging, protein folding, and fluid dynamics simulation. The disadvantage of CUDA is that it is primarily supported on NVidia devices, although cross-compilers exist for other architectures. A similar, vendor-specific language was Close To the Metal introduced by Advanced Micro Devices Inc. for ATI graphics cards, but it was later abandoned. Open Computing Language (OpenCL), which uses both task-based and data-based parallelism, is another framework designed for GPU programming supported by several hardware and software vendors, e.g. Apple, NVidia, Intel and AMD. Many CUDA based algorithms can now be ported to OpenCL, but OpenCL is still less popular than CUDA.

Programming in CUDA and OpenCL require extensive knowledge of the underlying hardware, especially if tuning for high performance is a top priority. Higher level libraries may help hiding some of the complexities. Scientific applications often benefit from accelerated linear algebra routines. These typically implement functions of the Basic Linear Algebra Subprograms (BLAS) or Linear Algebra PACKage (LAPACK). These can be provided by the vendor (such as CUBLAS) or third-party developers (such as in the case of Matrix Algebra on GPU and Multicore Architectures (MAGMA), which is based on CUDA [30], and ViennaCL, which is based on OpenCL [26]). We blend such libraries with our custom kernels to implement DQC.

## 5. Accelerating dynamic quantum clustering

DQC makes an ideal candidate for acceleration on multicore CPUs and on GPUs because the sequential parts of the algorithm are minimal. In fact, sequential operations appear only in disk I/O and hardware initialization. Algorithm 1 outlines the computational steps involved in the calculations. Each step involves a large number of steps that can be executed in parallel. When benchmarking the GPU implementation (see Section 6), we wanted to be fair and therefore the CPU version is also a multicore implementation. In this section, we give the details of our GPU variant of the algorithm, but where necessary, we also mention crucial issues with regard to the multicore CPU implementation.

The calculations start with constructing three sets of matrices. These matrices commonly use the Euclidean distance or an exponential thereof. These pairwise distances can be efficiently computed on a GPU [10]. The steps are decomposed into matrix level operations (see Algorithm 2 [20, 32]). Here Step 4 is a BLAS matrix multiplication, whereas the other steps can be calculated by custom kernels.

The first matrix to compute is essentially the Gram matrix of a radial basis function kernel:

$$N_{ij} = \langle \psi_i | \psi_j \rangle = e^{\frac{-(x_i - x_j)^2}{4\sigma^2}}$$

**Algorithm 1** The outline of the calculations in dynamic quantum clustering

```
Initialize Gram matrix $N$
Calculate Hamiltonian
Calculate position operator
Compute eigendecomposition of $N$
Compute square root of $N$
Basis transformation of Hamiltonian
Basis transformation of position operator
repeat
    Compute matrix exponential of transformed Hamiltonian at time $t_n$
    Compute expectation of value of position operator at time $t_n$
until
```

The elements are the inner products of the functions associated with the data points. In the DQC formulation, the functions are Gaussians, hence we deal with a non-orthogonal set of functions. The Gaussians do not have a compact support and thus the Gram matrix will be invariably dense. Implementing the above calculation on the GPU is straightforward, coalesced memory access can be ensured, the GPU kernel is a simple pairwise distance.

The second matrix to obtain is the Hamiltonian, which consists of a kinetic and a potential energy part:

$$H_{ij} = \langle \psi_i | \mathcal{H} | \psi_j \rangle = \langle \psi_i | (T + V(x)) | \psi_j \rangle.$$

The kinetic part is as easily calculated with a GPU kernel as the Gram matrix, with very similar constraints:

$$\langle \psi_i | T | \psi_j \rangle = \langle \psi_i | -\frac{\nabla^2}{2m} | \psi_j \rangle = \frac{1}{2m} \frac{(x_i - x_j)^2}{2\sigma^2} e^{\frac{-(x_i - x_j)^2}{4\sigma^2}}.$$

**Algorithm 2** Calculate the Euclidean distance matrix of data points with matrix operations ($\circ$ is the Hadamard product)

```
1: $v = (X \circ X)[1, 1 \ldots 1]^T$
2: $P_1 = [vv \ldots v]$
3: $P_2 = P_1^T$
4: $P_3 = XX^T$
5: $D = (P_1 + P_2 - 2P_3)$
```

The potential energy is a more complex formula. To calculate $\langle\psi_i|V(x)|\psi_j\rangle$, we need $V(x)$ first. We obtain it by solving $\mathcal{H}\psi = 0$. We get

$$V(x) = -\frac{d}{2} + \frac{1}{2\sigma^2 \sum_i e^{\frac{-(x-x_i)^2}{2\sigma^2}}} \sum_i (x-x_i)^2 e^{\frac{-(x-x_i)^2}{2\sigma^2}}.$$

The potential can be approximated with the first term of the Taylor series:

$$\langle\psi_i|V(x)|\psi_j\rangle = e^{-\frac{(x_i-x_j)^2}{4\sigma^2}} V(\frac{x_i+x_j}{2}).$$

To efficiently calculate the resulting formula, we introduced two intermediary arrays to store the values $e^{\frac{-(x_i-x_j)^2}{8\sigma^2}}$ and $(x_i-x_j)^2 e^{\frac{-(x_i-x_j)^2}{8\sigma^2}}$. These can be accessed in a coalesced fashion in the subsequent kernel calculation of $V$.

The last type of matrices to be calculated is the expectation values of the position operator:

$$X_{ij} = \langle\psi_i|x|\psi_j\rangle = \frac{x_i+x_j}{2} e^{-\frac{(x_i-x_j)^2}{4\sigma^2}}$$

This is again straightforward to implement on the GPU.

The rest of the process only involves eigendecompositions and matrix multiplications, so many data points can be evolved simultaneously and in parallel on a multicore CPU or a GPU. Since we have a non-orthogonal set of basis functions, we need to transform the basis to an orthogonal one. To do so, we find the eigenvectors of the symmetric matrix $N$ which correspond to states having eigenvalues larger than some cut-off value. This maps to the standard LAPACK operation `syevr`.

The next step is computing $N^{-\frac{1}{2}} = VD^{-\frac{1}{2}}V^{-1}$, where $V$ is the matrix of eigenvectors and $D$ is the diagonal matrix of eigenvalues. $N$ is a symmetric real-valued matrix, therefore the eigenvalues are real and $V$ is orthogonal, $V^{-1} = V^T$. This calculation is better taken in two steps. Instead of performing two matrix multiplications ($C = VD^{-\frac{1}{2}}$ first, where $C$ is a temporary matrix, then $N^{-\frac{1}{2}} = CV^T$), the first operation reduces to a much simpler kernel where the eigenvalues give a weight to the vectors of $V$.

Since $H$ and $X$ were calculated in a non-orthonormal basis, they have to be transformed to the new orthonormal basis, $H^{tr} = N^{-\frac{1}{2}}HN^{-\frac{1}{2}}$ and $X^{tr} = N^{-\frac{1}{2}}XN^{-\frac{1}{2}}$. These two calculations are two matrix multiplications each, where the first multiplication is between symmetric matrices.

To calculate the time evolution of $H$, we need the exponential of the matrix to construct $|\psi_i(t)\rangle = e^{-itH^{tr}}|\psi_i\rangle$ , such that $|\psi_i(t=0)\rangle = \psi_i\rangle$. While there are many ways of computing the exponential, we opt for finding the eigenvectors and eigenvalues of $H^{tr}$, and proceed in a similar fashion as in the matrix square root. Then the exponential can be calculated in a time-efficient manner of a range of intervals, eventually constructing the desired trajectories

$$\langle x_i(t)\rangle = \langle \psi_i|e^{itH^{tr}}X^{tr}e^{-itH^{tr}}|\psi_i\rangle.$$

## 6. Discussion of results

We used a non-distributed environment with a single workstation with 24 Gbyte of main memory, one quad-core 2.4GHz Intel Xeon E5620 CPU with HyperThreading enabled, an Nvidia Tesla C2050 computing processor with 448 cores and 3 Gbyte memory, running in a 64-bit Linux environment with CUDA 4.1 and GCC 4.3.4. For optimised multicore BLAS and LAPACK, we used ATLAS 3.8.4 [34], compiled with support for eight cores (that is, including the HyperThreaded cores). For the GPU-based BLAS, we tested both the vendor-provided CUBLAS and MAGMA 1.1.0 [30]. We found MAGMA marginally faster, and therefore we relied on MAGMA calls. LAPACK operations were also borrowed from MAGMA. The initial matrix constructions on the CPU were parallelized by exploiting loop-level parallelism by OpenMP directives [12]. When timing the execution time, we repeated the experiments ten times and averaged the results. Single-precision calculations used four-byte float variables, whereas double-precision relied on eight-byte doubles.

Since the effectiveness of DQC was already demonstrated in [33], we were only interested in the computational efficiency in the benchmarks. The data set therefore was randomly generated at different scales to enable accurate profiling, and it was inherently two-dimensional.

Since the data is dense, the memory requirements of the involved matrices are growing as a square function of the number of data points (Figure 2). Some of the linear algebra operations used do not allow in-place execution. For instance, a matrix product requires three matrices to be in the memory: the two matrices to be multiplied, and the result matrix. None of the matrices to be multiplied can be used as the matrix to store the result. This puts a severe limit on the GPU, as the device memory cannot easily accommodate three large matrices at the same time. Other operations

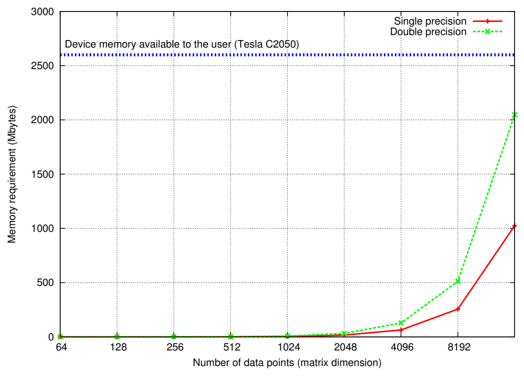


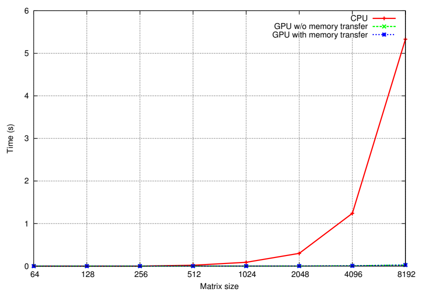
Figure 2: Memory requirements of square matrices

such as eigendecomposition are less hungry for memory, and almost the entire device memory can be allocated for the matrix to be decomposed. In the experiments below we were only interested in comparing pure GPU and pure CPU variants of the algorithm, but it might be worthwhile exploring a mixed approach in which the memory-intensive parts are always executed on the GPU. For instance, while matrix multiplication is much faster on the GPU, even the CPU version is reasonably fast, allowing larger matrices to be processed. The same large matrices can be decomposed on the GPU, which would give a great boost even with two-way memory transfers involved.

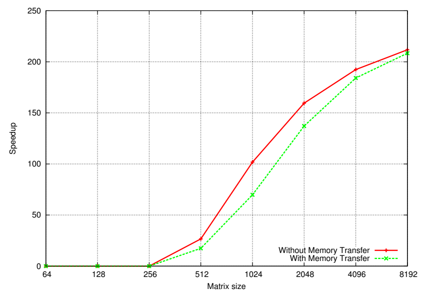
The results below show the execution time and speedup values for the major operations in the algorithm. We only included the results for one of the initial matrices, since the others follow a similar computational pattern. Also since the matrix exponential follows the same pattern as the square root of the matrix, we omit including the results.

Calculating the Gram matrix is a speedy process, barely taking seconds on the CPU even for large matrices (Figure 3). Nevertheless, the speedup using the GPU is spectacular, being two magnitudes faster. This shows the potential of a pure GPU implementation.

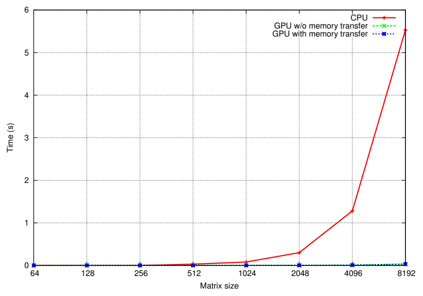
We encountered difficulties in calculating the eigendecomposition of the symmetric matrix $N$. The CPU variant computes the eigenvectors and eigenvalues with the standard `syevr` LAPACK call to ATLAS and we found the results fairly accurate. Unfortunately, MAGMA does not yet have the function `syevr`, only `syevd`, which uses a different numerical method. We found the outcome extremely inaccurate even for small matrices and even when



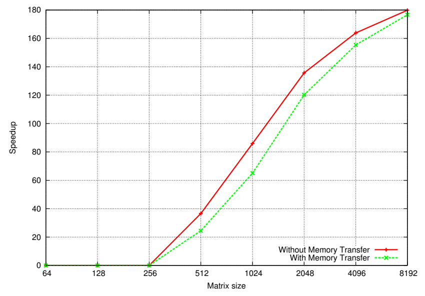
(a) Execution time of calculating the Gram matrix, single precision



(b) Speedup of calculating the Gram matrix, single precision



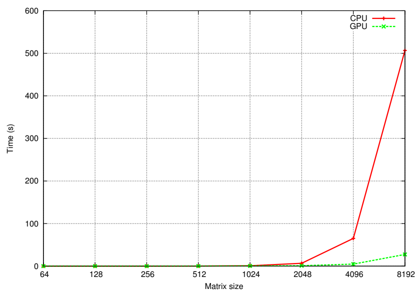
(c) Execution time of calculating the Gram matrix, double precision



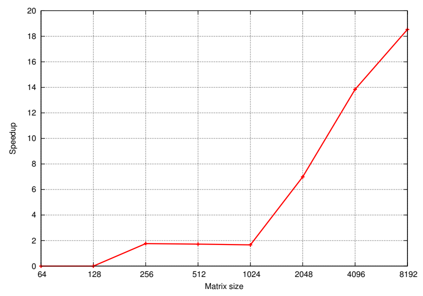
(d) Speedup of calculating the Gram matrix, double precision

Figure 3: Execution time and speedup of calculating the Gram matrix

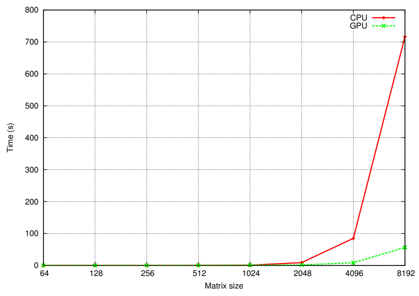
using double precision. We measured the time for the GPU version, but we used the results of the CPU version for the remaining steps. We believe this is a temporary problem with the library, and we hope that future versions will include an accurate `syevr` function. An additional issue was that two variants were provided by MAGMA for `syevd`: a GPU-resident version in which the input and output matrices reside in the device memory, and another one in which the input and output matrices are in the main memory, but calculations are performed on the GPU. The latter version hides memory transfers. We found the GPU-resident version unstable and used the second variant. Since we had no direct access for timing the memory transfers, the execution time and speedup results contain the time taken from calling until its return (Figure 4). This means that the numbers include a two-way memory transfer. With that in mind, we found an impressive 12.74x speedup for double precision, and 18.47x speedup for single precision for a matrix of 8192 dimensions. The single precision variant could scale to 16,384 dimensions,



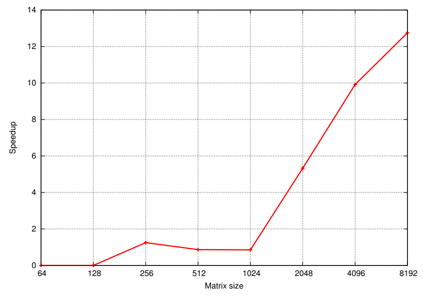
(a) Execution time of eigendecomposition, single precision



(b) Speedup of eigendecomposition, single precision



(c) Execution time of eigendecomposition, double precision



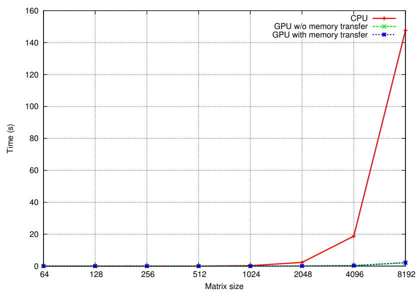
(d) Speedup of eigendecomposition, double precision

Figure 4: Execution time and speedup of eigendecomposition

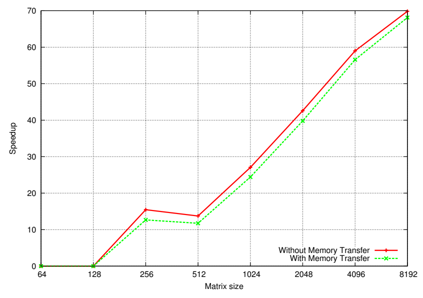
with a speedup of 22.98x (not included in the figure).

The execution time and speedup of square rooting the Gram matrix took much less time in total, and the speedup is even more convincing (Figure 5). The double precision variant achieves a speedup close 80x for larger matrices.

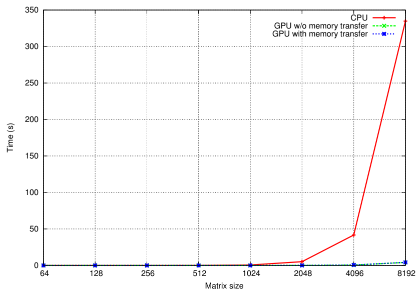
The basis transformation step had another surprise. The first matrix multiplication is between symmetric matrices, but we found the corresponding BLAS call (`symm`) slower than the general matrix multiplication call (`gemm`). This was true for both the CPU and the GPU variant. We do not fully understand the underlying reasons, but we proceeded with two general matrix multiplications per basis transformation. Figure 6 shows the timing and speedup results. The speedup of the single precision variant is especially notable.



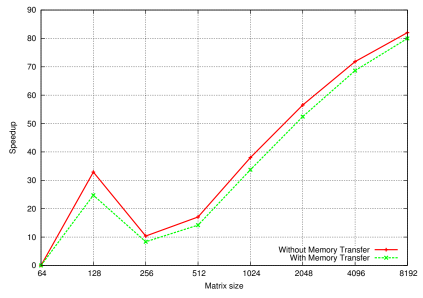
(a) Execution time of matrix square root, single precision



(b) Speedup of matrix square root, single precision



(c) Execution time of matrix square root, double precision



(d) Speedup of matrix square root, double precision

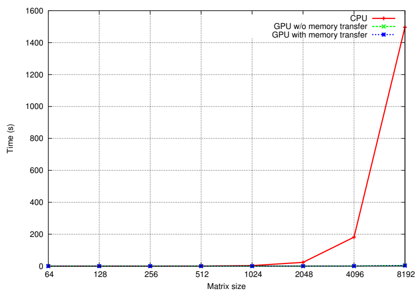
Figure 5: Execution time and speedup of matrix square root

## 7. Conclusions

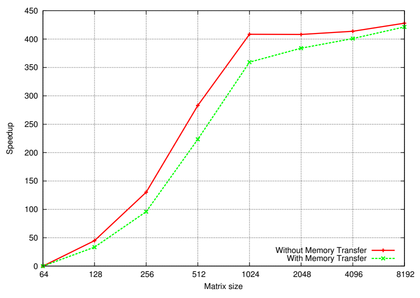
Computational physics as a field of inquiry has an untapped potential and relevance to machine learning and computational intelligence. This paper is an initial foray into this interdisciplinary domain, hoping to attract the attention of practitioners.

Graphics hardware reached up to our expectations and we achieved a speedup of up to two magnitudes in our implementation using a single GPU. We believe that this is significant, especially since we compared the execution time to a highly optimized multicore CPU implementation. Such acceleration greatly improves the usability of DQC in real-life application scenarios.

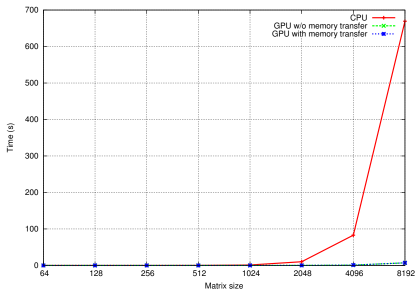
The bottleneck remains the limited memory of a GPU device. Since some operations require holding three large matrices in the device memory until a computation is finished, large data sets will need a slightly different approach. One option is to blend CPU and GPU code. For instance, the



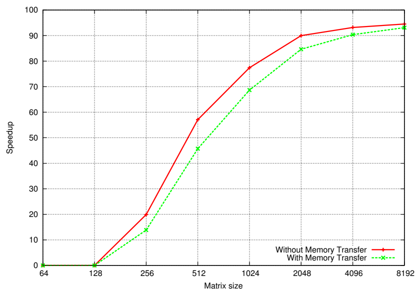
(a) Execution time of operator basis transformation, single precision



(b) Speedup of operator basis transformation, single precision



(c) Execution time of operator basis transformation, double precision



(d) Speedup of operator basis transformation, double precision

Figure 6: Execution time and speedup of operator basis transformation

eigendecomposition requires less memory and it is very taxing on the CPU; it can be efficiently performed on the GPU even with the overhead of two-way memory transfers. Basis transformation is also much faster on the GPU, but the execution time is still acceptable on the CPU. Given the memory requirements, this operation is better suited to the CPU.

Another option for further scaling is adapting the pure GPU algorithm to work in a distributed multi-GPU environment. In this case, the linear algebra operations have to be broken down further, possibly using a divide-and-conquer strategy. Such strategy is already used implicitly in our implementation, for instance, the numerical eigendecomposition relies on one such method. Some early results show that eigendecomposition does not scale linearly beyond a small number of nodes. An alternative method, the so-called Trotter-Suzuki algorithm [14], avoids the decomposition in calculating the evolution of a quantum system. Highly optimized CPU and GPU kernels for a single node have already been developed [6]. Our future work would like to extend the implementation to a distributed environment. Library support for such efforts is limited, extensive work with GPU-specific features is necessary for an efficient algorithm.

[1] AMD accelerated parallel processing – OpenCL programming guide, December 2011.

[2] NVida Compute Unified Device Architecture Programming Guide 4.1, November 2011.

[3] D. Aerts and M. Czachor. Quantum aspects of semantic analysis and symbolic artificial intelligence. *Journal of Physics A: Mathematical and General*, 37:L123–L132, 2004.

[4] J.A. Anderson, C.D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342–5359, 2008.

[5] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, and S.W. Williams. The landscape of parallel computing research: A view from Berkeley. Technical report, University of California at Berkeley, 2006.

[6] C.S. Bederián and A.D. Dente. Boosting quantum evolutions using Trotter-Suzuki algorithms on GPUs. In *Proceedings of HPCLatAm-11, 4th High-Performance Computing Symposium*, Córdoba, Argentina, December 2011.

[7] M. Blatt, S. Wiseman, and E. Domany. Superparamagnetic clustering of data. *Physical Review Letters*, 76(18):3251–3254, 1996.

[8] K. Blekas and IE Lagaris. Newtonian clustering: An approach based on molecular dynamics and global optimization. *Pattern Recognition*, 40(6):1734–1744, 2007.

[9] P.D. Bruza and R.J. Cole. Quantum logic of semantic space: An exploratory investigation of context effects in practical reasoning. In S. Artemov, H. Barringer, A. S. d'Avila Garcez, L.C. Lamb, and J. Woods, editors, *We Will Show Them: Essays in Honour of Dov Gabbay*. College Publications, 2005.

[10] D. Chang, N.A. Jones, D. Li, M. Ouyang, and R.K. Ragade. Compute pairwise Euclidean distances of data points with GPUs. In *Proceedings of CBB-08, International Symposium on Computational Biology and Bioinformatics*, pages 278–283, Orlando, FL, USA, November 2008. ACTA Press.

[11] X. Cui, J. Gao, and T.E. Potok. A flocking based algorithm for document clustering analysis. *Journal of Systems Architecture*, 52(8):505–515, 2006.

[12] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering*, 5(1):46–55, 1998.

[13] B. Daróczy, R. Pethes, and A.A. Benczúr. SZTAKI @ ImageCLEF 2011. In *Proceedings of CLEF-11, Conference on Multilingual and Multimodal Information Access Evaluation*, Amsterdam, The Netherlands, September 2011.

[14] H. De Raedt. Computer simulation of quantum phenomena in nanoscale devices. *Annual Reviews of Computational Physics*, 4:107–146, 1996.

[15] E. Di Buccio and G. Di Nunzio. Distilling relevant documents by means of dynamic quantum clustering. In *Proceedings of ICTIR-11, 3rd International Conference on the Theory of Information Retrieval*, pages 360–363, Bertinoro, Italy, September 2011.

[16] D. Horn and A. Gottlieb. Algorithm for data clustering in pattern recognition problems based on quantum mechanics. *Physical Review Letters*, 88(1):18702, 2001.

[17] A.Y. Khrennikov. *Ubiquitous quantum structure: from psychology to finance.* Springer Verlag, 2010.

[18] K. Kitto. Why quantum theory? In *Proceedings of QI-08, 2nd International Symposium on Quantum Interaction*, pages 11–18, Oxford, UK, March 2008.

[19] T. Kohonen, S. Kaski, K. Lagus, J. Salojärvi, J. Honkela, V. Paatero, and A. Saarela. Self organization of a massive text document collection. *IEEE Transactions on Neural Networks*, 11(3):574–585, 2000.

[20] Q. Li, V. Kecman, and R. Salman. A chunking method for Euclidean distance matrix calculation on large dataset using multi-GPU. In *Proceedings of ICMLA-10, 9th International Conference on Machine Learning and Applications*, pages 208–213, Washington, DC, USA, December 2010.

[21] A. Narayanan and T. Menneer. Quantum artificial neural network architectures and components. *Information Sciences*, 128(3):231–255, 2000.

[22] N. Nasios and A.G. Bors. Kernel-based classification using quantum mechanics. *Pattern Recognition*, 40(3):875–889, 2007.

[23] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, and T.J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[24] T. Preis, P. Virnau, W. Paul, and J.J. Schneider. GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model. *Journal of Computational Physics*, 228(12):4468–4477, 2009.

[25] G. Purushothaman and N.B. Karayiannis. Quantum neural networks (QNNs): inherently fuzzy feedforward neural networks. *IEEE Transactions on Neural Networks*, 8(3):679–693, 1997.

[26] K. Rupp, F. Rudolf, and J. Weinbub. ViennaCL - a high level linear algebra library for GPUs and multi-core CPUs. In *Proceedings of GPUScA-10, 2nd International Workshop on GPUs and Scientific Applications*, pages 51–56, Galveston Island, TX, USA, October 2010.

[27] J.E. Stone, D.J. Hardy, I.S. Ufimtsev, and K. Schulten. GPU-accelerated molecular modeling coming of age. *Journal of Molecular Graphics and Modelling*, 29(2):116–125, 2010.

[28] G. Strong and M. Gong. Browsing a large collection of community photos based on similarity on GPU. *Advances in Visual Computing*, pages 390–399, 2008.

[29] J. Sun, B. Feng, and W. Xu. Particle swarm optimization with particles having quantum behavior. In *Proceedings of CEC-04, Congress on Evolutionary Computation*, volume 1, pages 325–331, June 2004.

[30] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proceedings of IPDPSW-10, 24th IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1–8, Atlanta, GA, USA, April 2010.

[31] I.S. Ufimtsev and T.J. Martínez. Graphical processing units for quantum chemistry. *Computing in Science & Engineering*, 10(6):26–34, 2008.

[32] K.E.A. van de Sande, T. Gevers, and C.G.M. Snoek. Empowering visual categorization with the GPU. *IEEE Transactions on Multimedia*, 13(1):60–70, 2011.

[33] M. Weinstein and D. Horn. Dynamic quantum clustering: A method for visual exploration of structures in data. *Physical Review E*, 80(6):066117, 2009.

[34] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.

[35] Y. Zhang, F. Mueller, X. Cui, and T. Potok. Large-scale multi-dimensional document clustering on GPU clusters. In *Proceedings of IDPDS-10, 24th International Parallel and Distributed Computing Symposium*, Atlanta, GA, USA, April 2010.