PARALLISING K-MEANS CLUSTERING

***Abstract*—This document describes the process of parallising K-means clustering. The parallel solution is written in CUDA for compatibility with Nvidia graphics processors. The design is split into four revisions. Each revision utilises a different aspect of the Fermi memory hierarchy to improve memory throughput and load/store times. Each revision is characterised using Nvidias Visual Profiler to assess the next step for optimisation. The target platform is a Tesla C2075. The details of the serial solution can be found in the previous report.**

I. SERIAL ALGORITHM

A. Initial Parallel Proposal

K-means attempts to partition a feature space of N data points into K clusters. K-means' execution speed is limited by the distance metric. The distance calculation is a compute bound problem executing $N \times K$ Euclidean calculations. The complexity is of order $O(N^{dk+1} \times log(n))$. The design focus was to parallise the distance metric and classification of feature points. A master-slave design was proposed. It will format data for processing on the device, and collect/arrange data for processing on the host. Each thread will be responsible for assigning N/p feature points to k clusters. In addition the summation part of the convergence mean calculation will be done in parallel, and the division on the CPU. The host will vectorise data to utilise SIMD features during the CPU work.

B. Feedback Form

Each iteration requires memory to be copied from the host to device for work to be split between the GPU and CPU. It was suggested the majority of work should be completed on the GPU. In addition the parallel design was over-complicated.

C. Second Proposal

The parallel design requires three sets of data to be formatted and transferred to/from the GPU every iteration. The design will allocate host memory on the pinned buffer to improve the host to device throughput. In addition not all sets of data is changed on each iteration. Feature points do not change, and are copied to the GPU once. In addition the feature points classes are assigned on the GPU. Therefore an array is allocated on the GPU, updated by the kernel and copied to the host every iteration. Similarly centroid positions are updated on the CPU and copied to the device. This reduces the 6 GPU to/from host memory transfers initially required to 2.

II. NVIDIA DEVICE INFORMATION AND MEMORY HIERARCHY

The algorithm has been developed on a GeForce GTX 680. The differences in thread support and memory space between the GTX 680 and C2075 are shown in fig. 14.

	GTX 680	Tesla C2075
MPs	8	14
No. Cores per MP	192	32
Max. Threads per MP	2048	1534
Registers per Block	65536 B	32768 B

Fig. 1: Development and target platform specifications.

III. DIVIDING PROBLEM FOR TARGET ARCHITECTURE

A. Host-to-Device Management

The serial solution stores feature data in a vector of STL pairs. STL libraries are not compatible with CUDA. The first aspect of the design focuses on formatting host memory in such a way that the device can load/store it onto global memory.

Three sets of host memory are important. A cluster's centroid (X, Y) position, feature point (X, Y) positions and the cluster a feature points belong to. This data is formatted into three 1D arrays. As explained earlier the storage format reduces the number of host-device transfers. The arrays store data in a $[X_1Y_1X_2Y_2 \ldots X_nY_n]$ format. Threads in a half warp will always read adjacent memory addresses in parallel when accessing a feature point/centroid X or Y data. These coalescence access patterns minimise bank conflicts and latency.

By default the host allocates non-locked virtual memory to variables on the disk. When the host transfers non-locked data to global memory, data is parsed to the pinned buffer, then forwarded to the GPU via the DMA sequentially. The parallel design allocates three page-locked arrays directly onto the pinned buffers. The

device to/from host memory throughput is improved as the transfers to/from the disk are eliminated.

B. Memory Utilisation

Global memory read/write speeds may restrict the speed of the ALU. The parallel design avoids using global memory to minimise memory latency. The Tesla has 64kB of constant memory. The L2 cache handles overflowed data. It is more likely the user will increase the number of feature points than the number of clusters. 4000 double floating-point precision (8 Bytes) clusters can be stored in constant memory before memory overflow. In addition constant memory access speeds are as slow as global access unless all threads in a half-warp (16 threads) read the same memory address simultaneously. In these cases the memory address is read once and broadcasted to all threads in the half warp. All threads execute the same kernel code and have no opportunity to diverge. In addition centroid data is shared amongst all threads while feature data differs between threads. Therefore centroid data will benefit more from constant memory than feature data due to its read frequency. Furthermore threads in a half warp are likely to request the same centroid data simultaneously, utilising broadcast access speeds.

Both platforms distribute 64kB of shared memory to each block. The C2075s launch configuration is limited to 768 threads per block if it is to run the maximum 1534 threads concurrently on an MP. Therefore if each thread handles a single feature point, 24kB of shared memory is required per block. This allows any launch configuration to be used without consequence (from shared memories point of view). The L1:shared memory partition will be configured to favour the L1 cache. A larger L1 partition will cache more feature data, having faster access speeds than shared memory and enough space to handle register overflow. Blocking will be used to move feature points from the global memory to shared memory. Threads will only move data they're responsible for. Blocking will also benefit from the coalescence memory reads.

Register memory has the fastest read/write speeds. When register space overflows the GPU will use the L1 cache, then global memory. The Tesla is limited to 42 Bytes per thread when launching 768 threads per block. In reality this may be smaller as there is an overhead for storing temporary values in local memory. The serial algorithm requires 52 Bytes per thread. The serial distance calculation stores the distance between two points dimensions in a variable of type double. This will be replaced with the original distance calculation. This reduces the register consumption by 16 bytes. Increasing the number of memory reads increases the algorithms sensitivity to latency. This will quickly become a memory bound problem. This is more suitable as the arithmetic nature of the algorithm cannot be improved. The load/store speeds gained using the memory hierarchy will retain enough resources to achieve a theoretical occupancy of 100%, while reducing load/store latency.

C. Access Patterns

Shared memory stores data in banks on the Tesla C2075. Memory reads are sequential when threads request words from the same bank, and in parallel when words are requested from different banks. The [X_1 X_N Y_1 Y_N] storage format enables threads to read adjacent memory addresses simultaneously in a [1 2 3 ... N] access pattern. A two-way bank conflict will occur as the kernel stores data using 8 bytes (double).

D. Launch Configuration

The algorithm attempts to minimise the number of idle cores by launching the maximum number of threads possible. The design will distribute registers in such a way that an MP will always launch the maximum number of concurrent threads. The block size is selected dynamically, and will always be divisible by the number of threads per MP, M. Both platforms will run 2 blocks concurrently on each MP: The C2075 will launch 768 threads per block, T, and the GTX 680 will launch 1024 threads per block. The number of queued blocks is always the number of feature points divided by the threads per block. Over launching blocks is not an issue as scheduling is handled implicitly.

IV. Implementation Process

The design phase is split across four software revisions. Each revision trailed designs using Nvidia's visual profiler on the development platforms. Revision details are found in fig. 14.

Revision	Details
1.0	Global memory and concurrency
1.1	Constant memory and pinned memory
1.2	L1/Shared memory and blocking
1.3	Register improvements, referencing index

Fig. 2: Software revision history.

A. Revision 1.0

This revision address the formatting of data for memory transfers between the host and device. The revision moves host memory onto the GPUs global memory. Kernels were re-written in such a way that threads will calculate the cluster of a single feature point. An if statement is included to terminate over launched threads. The launch configuration is set statically. The number of blocks is set dynamically based on the supported threads per block and number of feature points. Launching the minimum number of blocks reduces the number of idle cores and the frequency of thread divergence as threads are unlikely to be terminated by the base case. The static launch configuration achieved a practical occupancy of 83%. However the launch configuration is restricted by the available register resources. There are 42B of register space available per thread for 68B of consumed memory.

Registers/Thread	18	18
Shared Memory/Block	0 B	0 B
◢ Occupancy		
Achieved	83.8%	89.2%
Theoretical	100%	100%

Fig. 3: Resources with/without local feature distances.

The Euclidean calculation stores the distance between features in a variable before a summation. The variables were replaced with their arithmetic counterparts. The required register space reduced from 68B to 52B with 12B of headroom for storing temporary values locally. The additional resources allowed the GPU to launch 5% more concurrent warps.

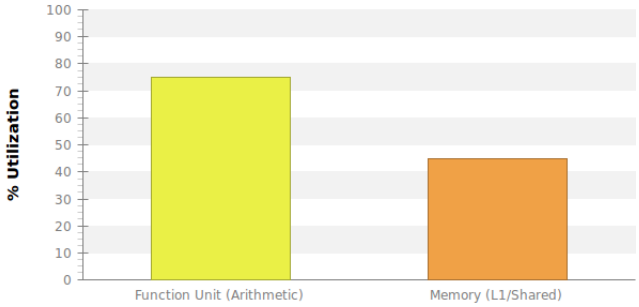


Fig. 4: Revision 1.0 kernel performance limiter.

K-means is a compute bound problem. The majority of a kernels time should be occupied by ALU operations. However memory utilisation occupies 45% of a kernels time. In addition the algorithm suffers from mid store/load times due to global memory load/store speeds. The ALUs utilisation is reduced as it waits for data to be fetched/stored between instructions.

The visual profiler issued warnings regarding memcpy/compute overlap, streams and memcpy throughput. The host to device memcpy throughput can be improved using pinned memory as explained earlier. However K-means requires the output from one stage before proceeding to the next. K-means' sequential nature prevents the use of streams and asynchronous transfers.

B. Revision 1.1

This revision utilises constant memory to reduce latency. Centroids [X Y] positions will be stored in constant memory for the discussed reasons. It also uses pinned memory to improve the DMA throughput. As explained in the parallel design, host memory is allocated on the pinned buffers using cudaMallocHost.

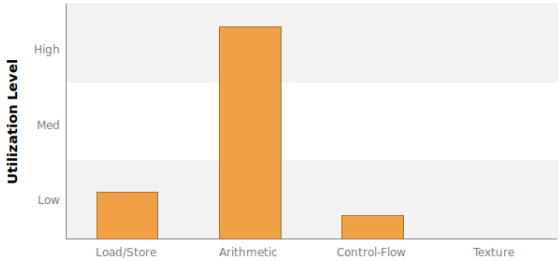


Fig. 5: Revision 1.1 kernel performance limiter.

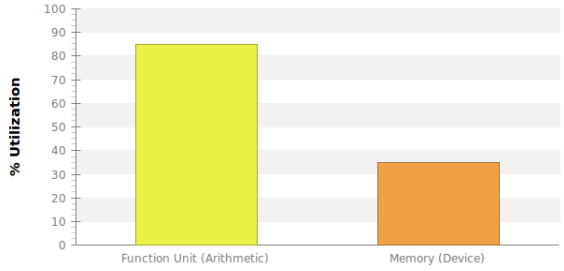


Fig. 6: Revision 1.1 kernel performance limiter.

The results shows the memory utilisation dropped by a further 10% and a low load/store utilisation was achieved. Furthermore the kernel execution time is faster than using global memory. The lower low/store and memory utilisation is a result of broadcast access patterns in the constant memory. Broadcasting has the same read speeds as registers. Without broadcasting the read speed is as slow as global memory. Therefore the load/store utilisation would have remained constant. The reduction in memory latency allows kernels to spend more time executing ALU instructions. Host memory was then allocated on the pinned buffers instead of the disk. Fig. 7 shows pinned memory doubles the DMA throughput. In addition the memory bandwidth warning was removed implying the device-to-host throughput is maximised.

Total Bytes	664.001 MB	564.001 MB
Avg. Throughput	4.58 GB/s	11.418 GB/s

Fig. 7: Revision 1.1 DMA throughput.

C. Revision 1.2

This revision uses blocking techniques to move feature data from global memory to shared memory. The first attempt uses blocking with the default cache configuration (48kB). Threads are synchronised after shared memory space is allocated to prevent out of bounds access. A synchronisation point is not required when the kernel completes as memcpy does this implicitly.

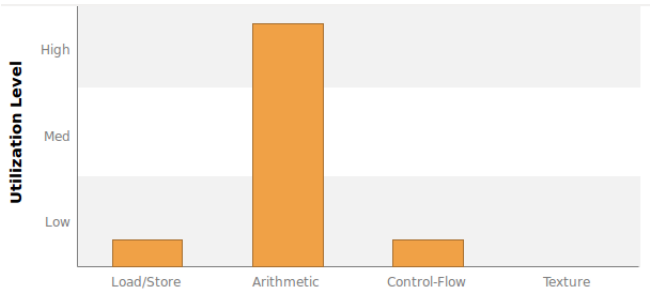


Fig. 8: Revision 1.2 compute analysis.

The memory utilisation dropped by 20% and kernel speed increased by 27.5%. Completely removing global memory accesses from the distance metric minimised the load/store times. This is due to two reasons. Shared and constant memory access speeds are faster than global. Each thread uses 4 constant and 4 shared reads for $N \times K$ iterations instead of 8 global reads. Secondly threads in half-warps will be at the same point in the kernel. The access patterns improved latency. Blocking allows half-warps to transfer 32 parts of memory (from 32 banks) between global and shared memory simultaneously, then request the same centroid data in parallel.

The cache partition was configured to favour L1 (16kB:48kB). 18kB of shared space was required. This would allow the majority of shared memory to be cached, which may improve read/write speeds. In reality the biased L1 partition was slower. The partition has 2kB of data overflow. Overflowed data is stored in the L2 cache. Accessing overflowed data from the L2 cache spent more time than the read speeds gained by caching 16kB of shared memory. An equal partition was used.

D. Revision 1.3

The ALU wastes time computing array indexes multiple times. Furthermore using 2*Threadidx.x indexes loses time to multiplications and member accessing. The kernel created a local copy of Threadidx.x and feature point indexes. Referencing reduced the register overhead for temporary variables and the ALU operations. As a result the memory and arithmetic utilisation dropped by 10%. However the execution speed increased. Using local memory meant less registers were available for threads. The compiler reduces the number of concurrent threads to ensure enough registers are distributed to fulfil the kernel requirements. Moving array indexes to shared memory would require an unjustifiable 16kB of space from the L1 partition.

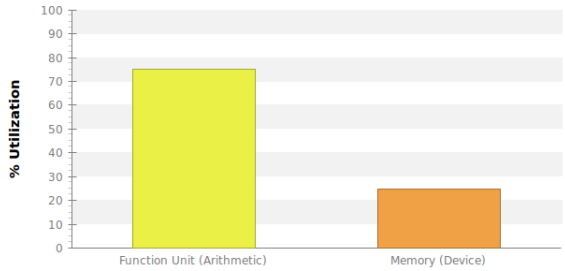


Fig. 9: Revision 1.3 kernel performance analysis.

Data was rearranged into a $[X_1X_2$ ... $X_nY_1Y_2$ ... $Y_n]$ format. This changes 2*ThreadIdx.x indexes to ThreadIdx.x. This saves 8 multiplications per distance metric calculation. Removing $8 \times N \times K$ multiplications per thread saved 10% of the ALUs time. Furthermore the memory utilisation dropped by 10%. However the kernels execution time deteriorated. The register/threads increased to 21 and the occupancy halved implying less threads were running concurrently due to a lack of register space. ThreadIdx.x + 1024 requires 2 bytes of local space per thread, while Threadidx.x*2 requires 1 byte of local space for 255 threads. To save local memory the two kernel arguments, CentroidSize and DataSize were stored in constant memory. This saved 8 bytes per thread. The register/thread became 20 for a practical occupany of 96.1%. The additional register space was used to store a local copy of threadIdx.x. The improvements have a 1.5% speed up compared to revision 1.2.

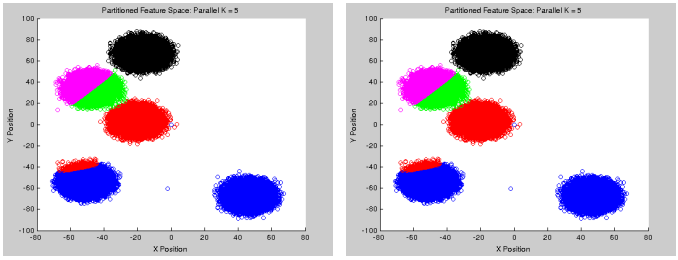
E. CPU Profiling: Next Step

A CPU profiler, gprof, identified whether other function would benefit from parallisation. The execution time consumed by the distance metric was split 45.61%:9.11% between classifying the feature points and converging centroids respectively. Less than 5% of the execution time is lost from calculating the new centroid positions, and 12.23% from converging centroids. An ideal speed up from 13x to 14x is not enough to justify the development time needed to parallise these functions.

V. Results

A. Verification on the Tesla C2075

This section evaluates the speed up achieved using the Tesla C2075. The first test verifies the parallel solutions operation. The serial and parallel algorithms partitioned the feature points into 5 clusters. The clusters starting position was constant. The results for the serial and parallel tests are shown in fig. 10 respectively. Colours represent clusters. The serial and parallel partition is identical. The algorithm performs correctly independent of the CPU and GPUs different rounding behaviours. This may be due to using double data types.



(a) Serial K = 5. (b) Parallel K = 5.

Fig. 10: Partitioned Feature Space

B. Testing on the Tesla C2075

Two tests were conducted. The first partitions a N feature points into 100 clusters. The number of feature points per cluster varies from 10 to 100k. The second test partitions 500k feature points into K partitions. The number of partitions varies. The results are shown in fig. 11 and fig. 12 respectively.

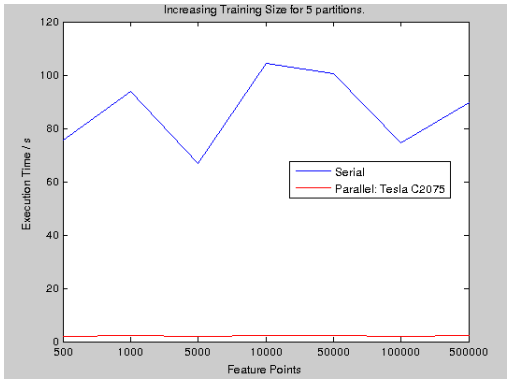


Fig. 11: Increasing Points per Cluster for 100 partitions.

The improvement of the former test case has a 40x speed increase. The algorithm has a growth of O(1). The practical growth has a ripple caused by the random starting positions of centroids. Different positions require different iterations until convergence. The measured complexity occurs due to the nature of the dataset. The synthetic dataset generated 5 uncluttered classes of equal distribution. Therefore changing the number of feature points per classes does not change the number of iterations until convergence. The iteration behaviour will change with different cluster distributions and centroid positions.

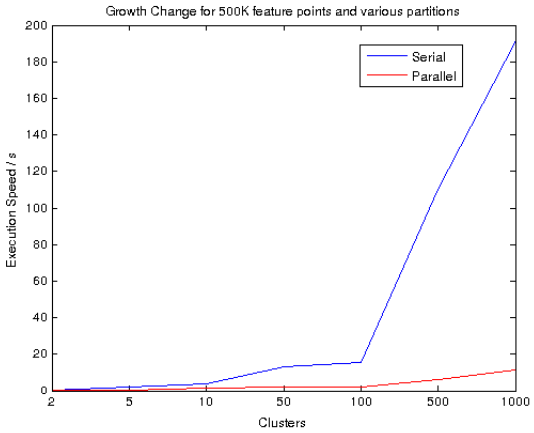


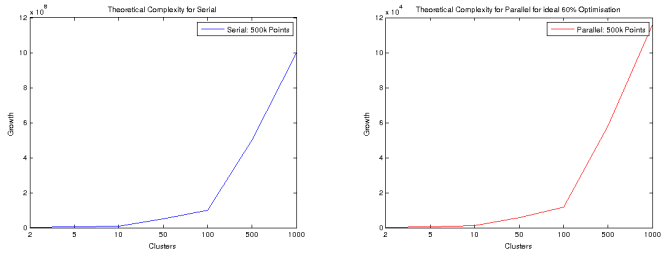
Fig. 12: Partitioned Feature Space

The parallel solution is 13x faster than the serial solution. The theoretical complexity is $O(N^k)$. As p concurrent threads are running N/p points in parallel, the practical complexity is $O((\frac{N}{13})^k)$. Thus the speed up is more prominent for a higher number of partitions.

C. Potential Speedup

The parallel implementation scales depending on the architecture. Theoretically it will be $O(f(n))/p$ times faster than the serial algorithm. The Fermi architecture on the Tesla supports up to 21476 threads (1534 threads per MP), equivalent to 48 cores per MP. However the Tesla only supports 32 cores per MP. The practical speed up is limited by the hardware which supports 14336 concurrent threads. In addition only 64% of the serial algorithm is improved by the parallel solution. The parallel speed up is $14336 * 0.64 = 8796$ times faster than the serial solution. The theoretical growth of the serial and parallel solutions are shown in fig. 13 respectively.

Revision 1.3 has a practical occupancy of 90%. This reduces the best speed up to 879.6x. Furthermore the algorithm reads the same memory address multiple times when computing distance calculations as a drawback to limited register space. Memory latency slows the execution speed further. The profiler showed a memory limitation of 25%, having a speed up of 35.88x faster. Finally 10% of time is taken up by load/store times, reducing the speed up to 18x. Finally the algorithm is



(a) Serial Growth. (b) Parallel Growth.

Fig. 13: Theoretical growth for varying partitions.

limited by the memory throughput and frequency of data transfers to/from the device, producing the speed up measured in 12.

The Teslas device limitations can be derived by comparing its performance to a similar device, the GT 520. The speed and memory differences are shown in fig. 14.

	Tesla C2075	GT 520
Cores per MP	32	48
GPU Clock Rate	1.15 GHz	1.62GHz
Memory Clock Rate	1.57 GHz	897 MHz
Bus Width	384-bit	64-bit

Fig. 14: Differences between the GT 520 and C2075.

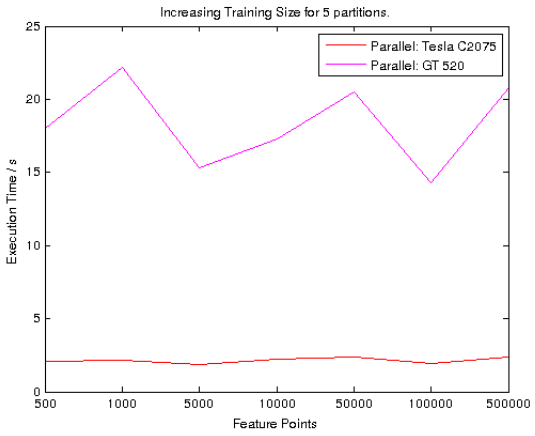


Fig. 15: Practical growth for various partitions.

Fig. 15 shows the C2075 has a practical speed up of 7x. The C2075s Fermi architecture supports 14x more concurrent threads than the 520. However the hardware is limited to 32 warps per MP. Therefore the C2075 only supports 9.3x more (14334) threads. In addition the Tesla has 1/3 the GPU clock rate of the GT 520, further reducing the speed up to 6.1x. The increased memory clock rate and bus size increases the speed up to 7x. The memory clock has less influence than the GPU clock as K-means is compute bound.

D. Practical Utilisation

Fig. 16 shows the kernels execution speed is limited by ALU memory requests. Register space was sacrificed to improve thread concurrency. As a result the Euclidean calculation re-reads the memory address of 4 variables twice from constant/shared memory instead of storing the value locally. The extra time wasted from memory requests/reads causes latency. As a consequence the ALU waits between instruction cycles for data to be fetched. This is shown in the kernel latency analysis in fig. 16 through data requests and execution dependencies.

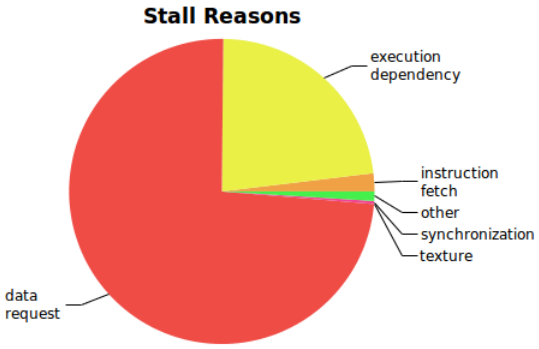


Fig. 16: Revision 1.3 kernel latency analysis.

Constant/shared memory and coalescence/broadcast access patterns were used to minimise latency. The access speed for constant memory is equal to register speeds. However like register space, constant memory space is limited. Storing index references cannot be done without restricting the number of clusters supported. As a result kernels computed array indexes N*K times.

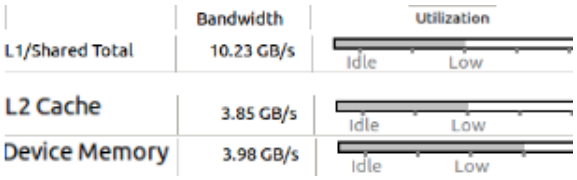


Fig. 17: Revision 1.3 memory usage.

Fig. 17 shows the memory hierarchy is utilised well, emphasising the arithmetic time is limited by the latency constraints. A practical occupany of 96.1% indicates there is enough resources to achieve maximum concurrency. Finally each iteration formats host/GPU data and copies it to/from the device. Unfortunately this design decision was compulsory due to the sequential nature of the algorithm. Although pinned memory improved memcpy throughput, it is the frequency of these transfers which limit the parallel speed up.