Computational Science Technical Note **CSTN-117**

GP-GPU and Multi-Core Architectures for Computing Clustering Coefficients of Irregular Graphs

A. Leist and K. A. Hawick and D. P. Playne

18-21 2011

Network science makes heavy use of simulation models and calculations based upon graph-oriented data structures that are intrinsically highly irregular in nature. The key to efficient use of data-parallel and multi-core parallelism on graphical processing units (GPUs) and CPUs is often to optimise the data layout and to exploit distributed memory locality with processing elements. We describe work using hybrid multi-core and many-core devices and architectures for implementing and optimising applications based upon irregular graph and network algorithms.

Keywords: multi-core; accelerators; GPU; CUDA; Cell; data parallelism

BiBTeX reference:

```
@INPROCEEDINGS{CSTN-117,
        author = {A. Leist and K. A. Hawick and D. P. Playne},
        title = {GP-GPU and Multi-Core Architectures for Computing Clustering Coefficients
                of Irregular Graphs},
        booktitle = {Proc. International Conference on Scientific Computing (CSC'11)},
        year = {18-21 2011},
        number = {CSC2720},
        pages = {3-9},
        address = {Las Vegas, USA},
        month = {18-21 July},
        publisher = {CSREA},
        keywords = {multi-core; accelerators; GPU; CUDA; Cell; data parallelism},
        owner = {kahawick},
        timestamp = {2011.05.16}
}
```

Technical Report CSTN-117

GP-GPU and Multi-Core Architectures for Computing Clustering Coefficients of Irregular Graphs

A. Leist, K.A. Hawick and D.P. Playne
Computer Science, Massey University
Albany, North Shore 102-904, Auckland, New Zealand
{ *a.leist, k.a.hawick, d.p.playne* } *@massey.ac.nz*

Abstract

Network science makes heavy use of simulation models and calculations based upon graph-oriented data structures that are intrinsically highly irregular in nature. The key to efficient use of data-parallel and multi-core parallelism on graphical processing units (GPUs) and CPUs is often to optimise the data layout and to exploit distributed memory locality with processing elements. We describe work using hybrid multi-core and many-core devices and architectures for implementing and optimising applications based upon irregular graph and network algorithms.

Index Terms

multi-core; accelerators; GPU; CUDA; Cell; data parallelism

1. Introduction

The recent introduction of the Graph 500 benchmarks [1] highlights the increasing importance of data intensive, graph-based algorithms in high performance computing. While traditional supercomputers have to yield more and more of the top spots in the TOP 500 [2] to hybrid systems featuring graphics processing units (GPUs) as compute accelerators, which provide the bulk of the processing power in those systems, irregular graph structures pose a challenge for general purpose computation on GPUs (GPGPU).

We implement the clustering coefficient as defined by Newman et. al. [3], a graph metric that is commonly used when analysing social networks, on a number of multi-core CPU and GPU architectures. We use this metric to compare the performance and scaling behaviour of a graph-based, bandwidth limited algorithm on these heterogeneous devices. Code fragments and pseudo-code are used to show how the implementations were optimised.

The clustering coefficient is a graph metric that is based on the concept of clustering in social networks, sometimes also called network transitivity, introduced by Watts and Strogatz [4]. It is often used when analysing networks with small-world characteristics [5], [4], [6]. Newman et. al. define the clustering coefficient C as follows:

$$C = \frac{3 \times (\text{number of triangles on the graph})}{\text{number of connected triples of vertices}}$$

Here, triangles are elementary circuits of length three, that is, three distinct vertices connected by three arcs creating a cycle. A connected triple is a path of length two that connects three distinct vertices.

In Section 2 we describe in detail how the clustering coefficient is computed for an arbitrary graph. We give algorithm fragments showing how we implement this on: single-core CPUs (Section 2.1), multi-core CPUs using the POSIX threads (PThreads) (Section 2.2) and threading building blocks (TBB) (Section 2.3) multi-threading libraries, NVIDIA's compute unified device architecture (CUDA) for both single-GPU (Section 2.4) and multi-GPU (Section 2.5) systems, as well as on the Cell Broadband Engine (CellBE) (Section 2.6). For details on these parallel hardware architectures, see the unpublished technical note [7].

We use two commonly found graph structures, small-world and scale-free, to compare the performance of the clustering coefficient algorithm on these platforms in Section 3. We discuss the outcomes and draw some conclusions in Section 4.

2. The Clustering Coefficient

Each one of the different hardware architectures described in this paper—x86 multi-core CPU, GPU and

CellBE—uses a very different approach to parallelism and thus requires an algorithm that is specifically tailored for its architecture to achieve peak performance. This section describes the implementations of the clustering coefficient algorithm along with architecture specific performance optimisations.

2.1. CPU - Sequential Algorithm

For reference purposes and to better explain the algorithm we use, we give a serial CPU code implementation of the clustering coefficient calculation in Algorithm 1.

Algorithm 1 Pseudo-code for the sequential CPU implementation of the clustering coefficient.

```
function CLUSTERING($G$)
    Input parameters: The graph $G := (V, A)$ is an array of
    adjacency-lists, one for every vertex $v \in V$. The arc set $A_i \subseteq A$
    of a vertex $v_i$ is stored in position $V[i]$. $|V|$ is the number of
    vertices in $G$ and $|A_i|$ is the number of neighbours of $v_i$ (i.e. its
    degree).
    R $\leftarrow$ determine reverse adjacency-list lengths
    declare $t$ //triangle counter
    declare $p$ //paths counter
    for all $v_i \in V$ do
        for all $v_j \in A_i$ do
            if $v_j = v_i$ then
                $p \leftarrow p - R[v_i]$ //correct for self-arcs
                continue with next neighbour $v_{j+1}$
            end if
            $p \leftarrow p + |A_j|$
            if $v_i > v_j$ then
                for all $v_k \in A_j$ do
                    if $v_k = v_i$ then
                        $p \leftarrow p - 2$ //correct for cycles of length 2 for both
                        $v_i$ and $v_j$
                        continue with next neighbour $v_{k+1}$
                    end if
                    if $v_k \neq v_j$ AND $v_i > v_k$ then
                        for all $v_l \in A_k$ do
                            if $v_l = v_i$ then
                                $t \leftarrow t + 1$
                            end if
                        end for
                    end if
                end for
            end if
        end for
    end for
    return $(3t)/p$ //the clustering coefficient
```

2.2. Multi-Core: POSIX Threads

The outermost loop of the sequential implementation executes once for every vertex $v_i \in V$. The iterations do not interfere with each other and can thus be executed in parallel. It is merely necessary to sum up the numbers of triangles and paths found in each of the parallel iterations to get the total counts for the graph before the clustering coefficient can be calculated. Algorithms 2 and 3 describe an implementation that uses POSIX Threads (PThreads) to achieve parallelism.

Algorithm 2 Pseudo-code for the multi-core CPU implementation of the clustering coefficient using PThreads. See Alg. 1 for a description of input parameter G and the triangle and paths counting algorithm.

```
declare $v_{curr}$ //the current vertex
declare mutex $m$ //mutual exclusion for $v_{curr}$
function CLUSTERING($G$)
    $R \leftarrow$ determine reverse adjacency-list lengths
    $v_{curr} \leftarrow 0$ //initialise current vertex $v_{curr}$
    $n \leftarrow$ number of CPU cores
    do in parallel using $n$ threads: call PROCESS($G$, $R$)
    wait for all threads to finish processing
    declare $t \leftarrow$ sum of triangles found by threads
    declare $p \leftarrow$ sum of paths found by threads
    return $(3t)/p$ //the clustering coefficient
```

Algorithm 3 Algorithm 2 continued. The function *PROCESS* is executed in parallel.

```
function PROCESS($G$, $R$)
    declare $t$ //local triangle counter
    declare $p$ //local paths counter
    declare $v_s$ //start vertex
    declare $v_e$ //end vertex
    repeat
        acquire lock on mutex $m$
        $v_s \leftarrow v_{curr}$
        $v_e \leftarrow v_s +$ work block size //$v_e$ must not exceed $|V|$
        $v_{curr} \leftarrow v_e$
        release lock on mutex $m$
        for all $v_i \in V_i \equiv \{v_s, \ldots, v_e\} \subseteq V$ do
            count triangles and paths as described in the sequential CPU
            implementation
        end for
    until $v_s \geq |V|$
    return $\{t, p\}$
```

2.3. Multi-Core: Threading Building Blocks

The TBB implementation, like the PThreads version explained before, applies the parallelism to the outermost loop. TBB's `parallel_reduce` can be used to do this parallel reduction without having to explicitly specify the chunk size and number of threads or having to worry about keeping all the threads busy.

Algorithm 4 shows how the full iteration range is defined and passed to `parallel_reduce`. TBB recursively splits the iteration range into sub-ranges until a certain threshold is reached. Then TBB uses available worker threads to execute `PROCESS_TASK` (Algorithm 5) in parallel. When the two halves of a range have been processed, then TBB invokes function `JOIN` (Algorithm 6) to combine the results. Eventually, all sub-ranges have been processed and the results have been joined into the root of the task tree. TBB returns and the results can be extracted from this root object.

Algorithm 4 Pseudo-code for the multi-core CPU implementation of the clustering coefficient using TBB. See Algorithm 1 for a description of the input parameter G and the triangle and paths counting algorithm.

```
function CLUSTERING($G$)
    $R \leftarrow$ determine reverse adjacency-list lengths
    declare blocked_range $br(0, |V|)$
    call parallel_reduce($br$, PROCESS_TASK)
    retrieve results and calculate clustering coefficient
```

Algorithm 5 TBB executes `PROCESS_TASK` in parallel if worker threads are available.

```
declare $t$ //task local triangle counter
declare $p$ //task local paths counter
function PROCESS_TASK($br, G, R$)
    declare $v_s \leftarrow br.begin()$ //start vertex
    declare $v_e \leftarrow br.end()$ //end vertex
    for all $v_i \in V_i \equiv \{v_s, \ldots, v_e\} \subseteq V$ do
        count triangles and paths as described in the sequential CPU
        implementation
    end for
```

Algorithm 6 TBB calls `JOIN` to combine the results of the two halves of a range.

```
function JOIN($x, y$)
    Input parameters: $x$ and $y$ are task objects.
    $x.t \leftarrow x.t + y.t$
    $x.p \leftarrow x.p + y.p$
```

2.4. GPU - CUDA

The CUDA implementation is much more complex due to the different hardware architecture and lower level performance tuning necessary to achieve high performance on the GPU.

Arbitrary graphs, like small-world networks, where the structure is not known beforehand, can be represented in different ways in memory. For CUDA applications, the data representation and resulting data accesses often have a major impact on the performance and have to be chosen carefully. Figure 1 illustrates the data structure used to represent a graph in device memory.

Another issue when processing arbitrary graphs with CUDA is that the adjacency-lists differ in length, and often it is necessary to iterate over such a list of neighbours. But since in CUDA's single-instruction, multiple-thread (SIMT) architecture all 32 threads of warp are issued the same instruction, iterating over the neighbours-lists of 32 vertices can cause warp divergence if these lists are not all of the same length. In the case of warp divergence, all threads of the warp have to execute all execution paths, which in this case means they all have to do x iterations, where x is the longest of the 32 adjacency-lists. And as described in the CPU implementation of the clustering coefficient algorithm, this particular case requires nested loops, which make the problem even worse.

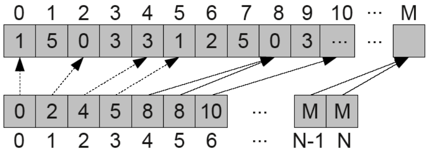

Figure 1: The data structure used to represent the graph in graphics card device memory. It shows the vertex set V (bottom) and the arc set A (top). Every vertex $v_i \in V$ stores the start index of its adjacency-list A_i at index i of the vertex array. The adjacency-list length $|A_i|$ can be calculated by looking at the adjacency-list start index of v_{i+1} ($V[i+1] - V[i]$). The vertex array contains $|V| + 1$ elements so that this works for the last vertex too.

However, the outermost loop can be avoided when the implementation iterates over the arc set A instead of the vertex set V. This improves the performance of the CUDA kernel considerably and also changes the total number of threads from $|V|$ to $|A|$. $|A|$ is usually much larger than $|V|$, giving CUDA more threads to work with, which it can use to hide memory latencies and which also means that the implementation should scale better to future graphics cards with more processing units. The implementation is described in Algorithm 7.

As the performance results section shows, this implementation performs well for graphs with only slightly varying vertex degrees, like Watts-Strogatz small-world networks [4]. If the vertex degrees of the input graph vary considerably, as it is typical for scale-free graphs with power-law degree distributions [8], [9], [10], a small variation of this implementation performs considerably better. In this second approach, the input array S not only references the source vertex of an arc, but also uses a second integer to store the end vertex. Furthermore, the host sorts this array by the degree of the arc end vertices before passing it to the CUDA kernel. Even though this means that the end vertex of each arc is stored twice, once in S and once in A, it makes it possible to process the arcs based on the vertex degrees of their end vertices,

Algorithm 7 Pseudo-code for the CUDA implementation of the clustering coefficient. It operates on the arc set A, executing one thread for every arc $a_i \in A$ for a total of $|A|$ threads. Self-arcs are filtered out by the host as they never contribute to a valid triangle or path. The host program prepares and manages the device kernel execution.

```
function CLUSTERING($V, A, S$)
  Input parameters: The vertex set $V$ and the arc set $A$ describe the
  structure of a graph $G := (V, A)$. Every vertex $v_i \in V$ stores
  the index into the arc set at which its adjacency-list $A_i$ begins in
  $V[i]$. The vertex degree $|A_i|$ is calculated from the adjacency-list
  start index of vertex $v_{i+1}$ ($V[i+1] - V[i]$). In order for this to
  work for the last vertex $v_N \in V$, the vertex array contains one
  additional element $V[N+1]$. $|A|$ is the number of arcs in $G$.
  $S[i]$ stores the source vertex of arc $a_i$.
  declare $V_d[|V|+1], A_d[|A|], S_d[|A|]$ in device memory
  copy $V_d \leftarrow V$
  copy $A_d \leftarrow A$
  copy $S_d \leftarrow S$
  declare $t_d, p_d \leftarrow 0$ in device memory //triangle and path counters
  do in parallel on the device using $|A|$ threads:
        call KERNEL($V_d, A_d, S_d, t_d, p_d$)
  declare $t, p$
  copy $t \leftarrow t_d$
  copy $p \leftarrow p_d$
  return  $(3t)/p$ //the clustering coefficient
```

Algorithm 8 Algorithm 7 continued. The device kernel is the piece of code that executes on the GPU.

```
function KERNEL($V, A, S, t, p$)
  declare $i \leftarrow$ thread ID queried from CUDA runtime
  declare $v_i \leftarrow S[i]$ //arc source
  declare $v_j \leftarrow A[i]$ //arc end
  $p \leftarrow p + |A_j|$
  if $v_i > v_j$ then
    for all $v_k \in A_j$ do
      if $v_k = v_i$ then
        $p \leftarrow p - 2$ //correct for cycles of length 2 for both $v_i$
        and $v_j$
        continue with next neighbour $v_{k+1}$
      end if
      if $v_i > v_k$ then
        for all $v_l \in A_k$ do
          if $v_l = v_i$ then
            $t \leftarrow t + 1$
          end if
        end for
      end if
    end for
  end if
```

which determine the number of iterations done by the outer one of the two loops in the CUDA kernel. This means that threads of the same warp can process arcs with similar end vertex degrees, thus reducing warp divergence considerably. The sorting is done by the host using TBB's `parallel_sort`, which utilises all available CPU cores.

Further CUDA specific optimisations applied to both versions of the clustering kernel are shown in

Algorithm 9 CUDA performance optimisations.

```
// shared memory counters
__shared__ unsigned int nTrianglesShared;
__shared__ unsigned int nPaths2Shared;
if (threadIdx.x == 0) {
  nTrianglesShared = 0;
  nPaths2Shared = 0;
}
__syncthreads();

// each thread uses registers to count
unsigned int nTriangles = 0;
int nPaths2 = 0;
...
// NOTE: this explicit caching can be
// counter-productive on Fermi devices!
const int prefetchCount = 7;
__shared__ int nbr2Prefetch[prefetchCount*
                            BLOCK_SIZE];
if (srcVertex > nbr1) {
  for (int nbr2Idx = 0; nbr2Idx < nArcsNbr1;
      ++nbr2Idx) {
    //pre-fetch nbr2 to shared mem. to take
    //advantage of the locality in
    // texture fetches
    int nbr2;
    int prefetchIdx=nbr2Idx%(prefetchCount+1);
    if (prefetchIdx == 0) {//global mem. read
      nbr2 = tex1Dfetch(arcsTexRef,
                        nbr1ArcsBegin+nbr2Idx);
      for (int i=0; i < prefetchCount; ++i) {
       nbr2Prefetch[i*blockDim.x+threadIdx.x]=
          tex1Dfetch(arcsTexRef,
                     nbr1ArcsBegin+nbr2Idx+i+1);
      }
    } else { // read from shared memory
      nbr2 = nbr2Prefetch[(prefetchIdx-1) *
                          blockDim.x+threadIdx.x];
    }
    ...
    for (int nbr3Idx=0; nbr3Idx < nArcsNbr2;
        ++nbr3Idx) {
      nTriangles += tex1Dfetch(arcsTexRef,
                      nbr2ArcsBegin+nbr3Idx)
                    == srcVertex ? 1 : 0;
    }
  }

  // write local counters to shared memory
  atomicAdd(&nTrianglesShared, nTriangles);
  atomicAdd(&nPaths2Shared,
            (unsigned int)nPaths2);
}

// write to global mem (once per thread block)
__syncthreads();
if (threadIdx.x == 0) {
  atomicAdd(nTotalTriangles,
    (unsigned long long int)nTrianglesShared);
  atomicAdd(nTotalPaths2,
    (unsigned long long int)nPaths2Shared);
}
```

Algorithm 9. They include counting the triangles and paths found by each individual thread in its registers, before writing them to shared memory, where the total counts for a thread block are accumulated, which are eventually written to global memory with a single atomic transaction per counter and thread block. Furthermore, texture fetches are used when iterating over the adjacency-lists of vertices, taking advantage of data locality. And because the caching done when fetching the neighbours v_k of vertex v_j may be overwritten by the inner loop, a constant number of arc end vertices are pre-fetched and written to shared memory. This pre-fetching is only done for older devices like the GTX295, as the latest generation of Fermi-based NVIDIA GPUs, which includes the GTX480, provides automatic caching in L2 (shared by all multiprocessors) and L1 (on each multiprocessor) caches. The overhead of manually caching the data in shared memory decreases the performance on these devices.

2.5. Multi-GPU - CUDA & POSIX Threads

When multiple GPUs are available in the same host system, then it may be desirable to utilise all of them to further reduce the execution time of the algorithm. And because the iterations of the outermost loop are independent from each other with no need for synchronisation, the work can be distributed over the available GPUs in the same way as multiple CPU cores are utilised by threads (See Section 2.2). One PThread is created for every GPU and controls the execution of all CUDA related functions on this particular GPU. The data structure of the graph is replicated on all graphics devices and instead of executing $|A|$ CUDA threads to count all triangles and paths with just one kernel call, a work block of N arcs $\{a_i, \ldots, a_{i+N-1}\} \subseteq A$ is processed during each kernel call. A new work block is determined in the same way as it is done when using PThreads to execute on multiple CPU cores. The work block size depends on the available graphics hardware and the size of the thread blocks in the CUDA execution grid: $N =$ (number of threads per block)$\times$(blocks per streaming multiprocessor)$\times$(number of streaming multiprocessors). The goal is to make it large enough to allow CUDA to fully utilise the hardware and small enough to keep all available GPUs busy for roughly the same amount of time.

2.6. Cell Processor - PS3

Like the CUDA implementation, the implementation for the Cell Broadband Engine (BE) requires a lot of architecture specific tuning to achieve good performance. The memory layout used is similar to the one used by the CUDA kernels, using one array for the vertices and one array for the arcs. However, the requirement that the direct memory accesses (DMA) used to transfer data from main memory to the local store of a Synergistic Processor Element (SPE) are aligned on 16-byte boundaries makes some changes necessary. See Figure 2 for an illustration and description of the memory layout.

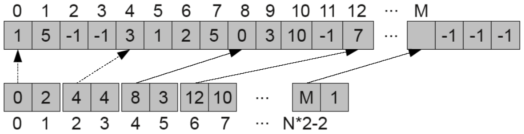
0 1 2 3 4 5 6 7 8 9 10 11 12 ··· M
1 5 -1 -1 3 1 2 5 0 3 10 -1 7 ··· -1 -1 -1
0 2 4 4 8 3 12 10 ··· M 1
0 1 2 3 4 5 6 7 ··· N*2-2

Figure 2: The data structure used to represent the graph in system memory of the Cell BE. It shows the vertex set V (bottom) and the arc set A (top). Every vertex $v_i \in V$ stores the start index of its adjacency-list A_i at index $i \times 2$ of the vertex array. The adjacency-list length $|A_i|$ is stored at the next index. The vertex array contains $|V| \times 2$ elements. Every adjacency-list in the arcs array is padded to the next multiple of 16-bytes (4-bytes per value) in order to conform with the memory alignment requirements. The padding elements have the value -1, which is an invalid vertex ID.

The main task of the Cell's PowerPC Processor Element (PPE) is to manage the Synergistic Processor Elements (SPEs) as illustrated in Algorithm 10. It is used to load the graph and store it in system memory using the memory layout described before. Then it initialises the SPEs, which do most of the actual computation (See Algorithms 11 and 12). However, the PPE would not be fully utilised if providing the SPEs with further work was all it did. Therefore, it performs some of the same computational tasks in its spare time, further improving the overall performance. The implementation of the triangle and paths counting algorithm on the PPE is basically the same as the single-threaded CPU implementation described in Algorithm 1, except that it uses the PPE's vector unit in the same way as the SPE implementation does in its innermost loop. These vector operations are described in Algorithm 13.

Traversing an arbitrary graph as it is done by the triangle and path counting algorithms requires many reads from unpredictable memory addresses. And since the local store of the SPEs with its 256KB capacity is relatively small, much too small to hold the entire graph structure of anything but very small graphs, it is necessary to load the required parts of the graph

Algorithm 10 Pseudo-code for the Cell BE implementation of the clustering coefficient. This algorithm describes the tasks of the PowerPC Processor Element. It operates on the vertex set V, issuing blocks of vertices to the the Synergistic Processor Elements for processing, as well as processing small work chunks itself when it has nothing else to do. Self-arcs are filtered out beforehand, as they never contribute to a valid triangle or path.

```
function CLUSTERING($V, A$)
  Input parameters: The vertex set $V$ and the arc set $A$ describe the structure of a graph $G := (V, A)$. Every vertex $v_i \in V$ stores the index into the arc set at which its adjacency-list $A_i$ begins in $V[i \times 2]$ and its degree in $V[i \times 2 + 1]$. $|V|$ is the number of vertices in $G$. $SPE = \{spe_0, spe_1, \ldots, spe_5\}$ is the set of SPEs.
  for all $spe_i \in SPE$ do
    initialise $spe_i$ and start processing a block of vertices
  end for
  while more vertices to process do
    for all $spe_i \in SPE$ do
      if inbound mailbox of $spe_i$ is empty then
        write the start and end vertices of the next work block to the mailbox
      end if
    end for
    process a small work block on the PPE
  end while
  for all $spe_i \in SPE$ do
    send interrupt signal and wait until $spe_i$ finishes processing
  end for
  aggregate results and calculate clustering coefficient
```

from system memory into local memory when needed. For example, when processing a certain vertex, then its adjacency-list has to be copied into the local store. This is done by issuing a DMA request from the Synergistic Processor Unit (SPU) to its Memory Flow Controller (MFC) (every SPE has one SPU and one MFC). However, the performance of the implementation would not be good if the SPU stalled until the requested data becomes available. Instead, the implementation for the SPE is split into phases (See Figure 3 and Algorithm 12). A phase ends after a DMA request has been issued and the following phase, which uses the requested data, is not executed until the data is available. This implementation of multi-buffering uses 16 independent buffers to process the work block issued to the SPE. Whenever a buffer is waiting for data, the implementation switches to another buffer that is ready to continue with the next phase.

The Cell PPE and SPE units all have their own vector units and 128-bit wide vector registers. This allows them to load four 32-bit words into a single register and, for example, add them to four other words stored in a different register in a single operation. A program for the Cell BE should be vectorised where possible to fully utilise the available processing power. Algorithm 13 describes how the innermost loop of the PPE and SPE implementations use of the vector units.

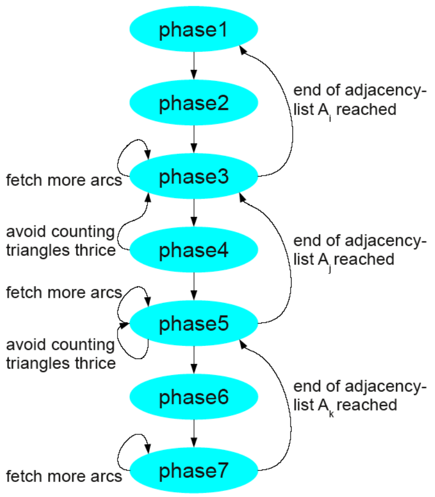
phase1
phase2
phase3
phase4
phase5
phase6
phase7
end of adjacency-list A_i reached
end of adjacency-list A_j reached
end of adjacency-list A_k reached
fetch more arcs
avoid counting triangles thrice
fetch more arcs
avoid counting triangles thrice
fetch more arcs

Figure 3: The phases of the SPE implementation and how they are connected to each other. The progression from phase x to phase $x + 1$ is always due to phase x issuing a DMA request to copy data from system memory into local memory, which is needed for phase $x + 1$ to execute. Phases with an odd number end after they issue a request to fetch the start index and length information about a particular adjacency-list, whereas phases with an even number end after they issue a request to fetch the actual adjacency-list data for a particular vertex. The figure illustrates under which conditions a phase is repeated or the execution path goes back up towards *phase1*. See Algorithm 12 for the pseudo-code of the phases implementation.

It turns out that the performance gain from using both the PPE and the SPEs to process the data is smaller than expected compared to using either only the PPE or only the SPEs to do the actual data crunching. It appears that the memory system is the bottleneck when using all of the available processing units on the Cell processor on a data-intensive problem like the one at hand.

3. Performance Results

This section compares the performance of the different clustering coefficient implementations. Table 1 lists the platforms used for the performance experiments. System (a) was used for the single-core and multi-core CPU measurements, both systems (a) and (b)

Table 1: The platforms used for the performance measurements. Note that only 6 of the total 8 SPEs on the CellBE are available to the developer (one is disabled and one reserved by the operating system).

ID	CPU	GPU	RAM	Operating System
(a)	Intel Core i7 970 @3.2 GHz (6 cores)	4× NVIDIA GTX480 (4 GPUs)	12 GB	Ubuntu 10.10 64-bit
(b)	Intel Core 2 Quad @2.66 GHz (4 cores)	NVIDIA GTX295 (2 GPUs)	4 GB	Ubuntu 10.10 64-bit
(c)	PlayStation 3 CellBE @3.2 GHz (1 PPE & 6 SPEs)	NVIDIA RSX	256 MB	Yellow Dog Linux 6.1

Algorithm 11 The pseudo-code for the SPE implementation of the clustering coefficient on the Cell BE. See Algorithm 10 for the PPE implementation and Algorithm 12 for the different execution phases.

```
function CLUSTERING($v_s, v_e$)
  Input parameters: Each SPE receives an initial work block
  $[v_s, \dots, v_e] \subseteq V$ of source vertices to process.
  copy init. data from system mem. to the local store
  initialise buffers $B = \{b_0, b_1, \dots, b_{15}\}$
  repeat
    $v_{curr} \leftarrow v_s$ //initialise current vertex $v_{curr}$
    for all $b_i \in B$ do
      $b_i.phase \leftarrow phase1$ //set the next phase of $b_i$
      mark buffer as "ready"
    end for
    //process the current work block
    set all buffers as active
    while at least one buffer is active do
      $b \leftarrow$ any "ready" buffer
      call $b.phase$ //execute the next phase of $b$
    end while
    //check if there is more work to do
    $v_s \leftarrow$ read next value from inbound mailbox
    if no interrupt signal recieved ($v_s \neq -1$) then
      $v_e \leftarrow$ read next value from inbound mailbox
    end if
  until interrupt signal received
  copy the results back to system memory
```

for the GPU results and system (c) for the CellBE implementation.

Two different graphs models are used as input to the algorithms. The Watts-Strogatz network model [4] generates small-world graphs, where every vertex is initially connected to its k nearest neighbours. These edges are then randomly rewired with a probability p. The graphs generated for the performance measurements ($k = 50$ and $p = 0.1$, see Figure 4) have a high clustering coefficient of ≈ 0.53. The vertex degrees do not deviate much from k.

The Barabási-Albert scale-free network model [9] generates graphs with a power-law degree distribution for which the probability of a node having k links follows $P(k) \sim k^{-\gamma}$. Typically, the exponent γ lies between 2 and 3 [8], [10]. The vertex degrees in the resulting graph vary considerably. The graphs generated for the performance measurements ($k \approx 50$, see Figure 5) have a clustering coefficient of ~ 0.01.

The timing results show that the type of graph used as input to the algorithms has a big effect on the execution times. The scale-free graphs take much longer to process than the small-world graphs, because even though only few vertices have a degree that is much higher than the average, most vertices are connected to one of these hub nodes and the algorithms therefore often have to iterate over the large adjacency-lists of these few vertices.

Table 2 gives an overview of the performance measurements and compares the results with each other. It shows that the multi-threading implementations using PThreads (12 threads) and TBB (automatic task management) are both $\approx 7\times$ faster than the sequential implementation for the largest measured instances of the small-world and scale-free graphs. Even though the processor only has 6 physical cores, Intel's hyper-threading technology effectively doubles this to 12 logical cores, which enables it to better utilise the physical cores. This makes it possible to scale beyond the actual number of cores. The TBB implementation performs almost exactly the same as the PThreads implementation. Its ease of development and automatic scaling to different system configurations thus makes it a powerful alternative to the more low-level multi-threading with PThreads.

As mentioned in Section 2.4, we have two CUDA kernels that differ in one aspect. The CUDA threads in kernel 1 access the array of arcs A in the given order, whereas kernel 2 uses a second array of arcs which is sorted by the degrees of the arc end vertices to determine which arc is processed by each thread. This second kernel uses $|A|\times$ (size of integer) more space and introduces some processing overhead, which shows in the lower performance when processing the small-world graphs. However, the reduced warp divergence gained through this overhead pays off when processing scale-free graphs. Here the scenario is reversed and kernel 2 clearly outperforms kernel 1 by a considerable margin. This shows once again [11], [12] that the performance of graph algorithms running on the GPU in many cases depends on the graph structure and that there is not one best implementation for all cases. If the graph structure is not known beforehand, then it may be worthwhile to attempt to automatically

Algorithm 12 Algorithm 11 continued. The phases of the SPE implementation execute on a buffer b. Each phase models a step in the process of counting the triangles t and paths p. A phase ends after a DMA request to load data into local storage has been issued or when the end of a loop is reached.

```
function phase1(b)
  $b.v_i \leftarrow v_{curr}$ //set the source vertex for this buffer
  $v_{curr} \leftarrow v_{curr} + 1$
  if $b.v_i \geq v_e$ then
    set buffer as inactive //end of work block reached
  else
    copy_async $b.v_i.dat \leftarrow$ load adjacency-list info
    $b.phase \leftarrow phase2$
  end if
function phase2(b)
  copy_async $b.A_i \leftarrow$ use $b.v_i.dat$ to load $A_i \subset A$
  $b.phase \leftarrow phase3$
function phase3(b)
  if end of adjacency-list $b.A_i$ reached then
    $b.phase \leftarrow phase1$ //loop condition not fulfilled
  else
    $b.v_j \leftarrow$ next value in $b.A_i$
    copy_async $b.v_j.dat \leftarrow$ load adjacency-list info
    $b.phase \leftarrow phase4$
  end if
function phase4(b)
  $b.p \leftarrow b.p + |A_j|$
  if $b.v_j > b.v_i$ then
    $b.phase \leftarrow phase3$ //do not count triangle thrice
  else
    copy_async $b.A_j \leftarrow$ use $b.v_j.dat$ to load $A_j \subset A$
    $b.phase \leftarrow phase5$
  end if
function phase5(b)
  if end of adjacency-list $b.A_j$ reached then
    $b.phase \leftarrow phase3$ //loop condition not fulfilled
  else
    $b.v_k \leftarrow$ next value in $b.A_j$
    if $b.v_k = b.v_i$ then
      $b.p \leftarrow b.p - 2$ //correct for cycles of length 2
      $b.phase \leftarrow phase5$
    else if $v_k > v_i$ then
      $b.phase \leftarrow phase5$ //don't count triangle thrice
    else
      copy_async $b.v_k.dat \leftarrow$ load adj.-list info
      $b.phase \leftarrow phase6$
    end if
  end if
function phase6(b)
  copy_async $b.A_k \leftarrow$ use $b.v_k.dat$ to load $A_k \subset A$
  $b.phase \leftarrow phase7$
function phase7(b)
  for all $b.v_l \in b.A_k$ do
    if $b.v_l = b.v_i$ then
      $b.t \leftarrow b.t + 1$ //triangle found
    end if
  end for
  $b.phase \leftarrow phase5$
```

Algorithm 13 Vector operations are used to speed-up the execution of the innermost loop (phase7) of the Cell BE PPE and SPE implementations. The comparison of vertex ID v_i with $v_l, v_{l+1}, v_{l+2}, v_{l+3}$ is done concurrently using the 128-bit vector unit. As the vector unit executes instructions in SIMD fashion, it is necessary to eliminate the branch. Several intrinsic instructions can be used to get the same effect as the if-condition: *spu_cmpeq* compares two vectors for equality and returns a bit-mask which represents true and false results; *spu_sel* selects one of two values (0 if the vertex IDs are not equal and 1 if a triangle has been found) based on this bit-mask; and *spu_add* adds the selected values to a vector that is used to count the number of triangles.

```
vec_uint4 case0 = spu_splats((uint32)0);
vec_uint4 case1 = spu_splats((uint32)1);
for (int nbr3Idx=0; <loop condition>;
    nbr3Idx+=4) {
  buf.nTrianglesVec =
    spu_add(buf.nTrianglesVec,
            spu_sel(case0,
                    case1,
                    spu_cmpeq(
   *((vec_int4*)&buf.arcsBuf3[nbr3Idx]),
                              buf.vertexId
                                )
                    )
            );
}
```

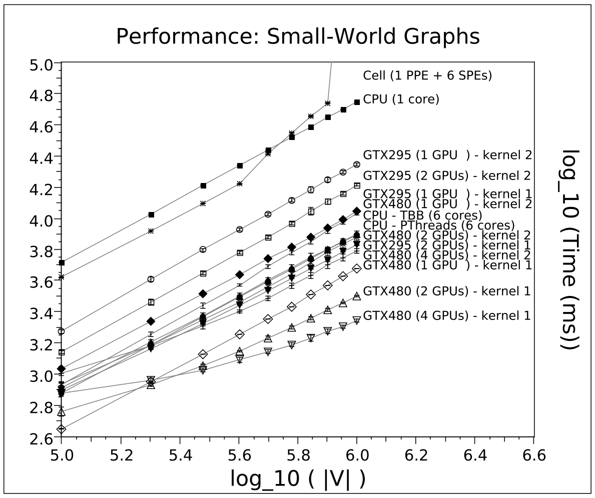

Figure 4: The timing results in milliseconds for Watts-Strogatz small-world graphs with rewiring probability $p = 0.1$ and degree $k = 50$. The number of vertices V ranges from $100,000 - 1,000,000$. All data points are the mean values of 20 measurements. Error bars show the standard deviations.

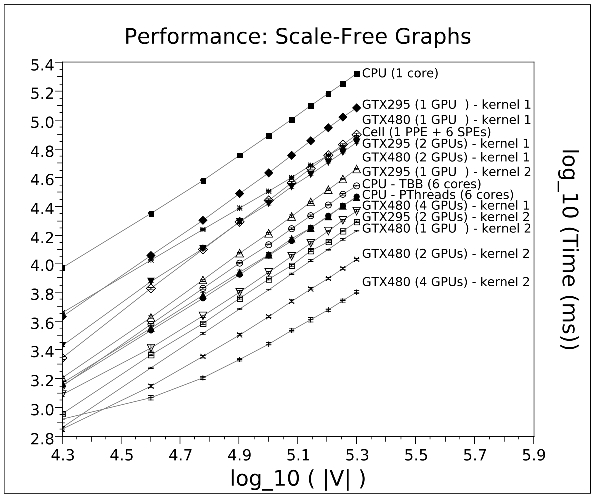

Figure 5: The timing results in milliseconds for Barabási scale-free graphs with a degree $k \approx 50$. The number of vertices V ranges from $20,000 - 200,000$. All data points are the mean values of 20 measurements. Error bars show the standard deviations.

determine the type of graph in order to be able to choose the optimal CUDA implementation.

The multi-GPU implementations using both GPUs of the GeForce GTX295 or up to four GeForce GTX480s perform best when the graph size is large enough to keep all processing units of the devices busy. Even the largest measured graph instances are not large enough to allow the GTX480s to scale particularly well. The slopes given in the table highlight this especially for the small-world graphs. The single-GPU implementation even outperforms the multi-GPU implementation for the smallest measured graph instances due to the overhead introduced by using multiple GPUs. The first three data points were filtered out when the fitted slopes were calculated so that these small graphs do not distort the scaling of the multi-GPU measurements.

The Cell implementation positions itself between the single- and multi-threaded CPU implementations when processing the scale-free graphs or the smaller instances of the small-world graphs. The timing results of the small-world graphs suddenly increase at the $V = 400,000$ mark and even more considerably at the $V = 800,000$ mark. This is caused by the minimalistic 256 MB of main memory available in the PlayStation 3, which forces the system to start paging memory to the hard drive. We therefore filter the results above $V = 400,000$ out when calculating the slope for these graphs to report the true scaling of the CellBE and compare the performance of the CellBE using graph instances of size $V = 400,000$.

Table 2: Performance comparison. The speed-up values are for the largest measured graph instance. An exception are the small-world network measurements on the Cell processor as mentioned in the main text. Speed-up **S1** is relative to the single-core CPU implementation, whereas speed-up **S2** compares the multi-core/GPU implementations to the respective single-core/GPU implementations. The slopes of the least square linear fits show how well the algorithms scale with increasing graph size. Values are rounded to 3 significant digits.

Compute Device	**S1**	**S2**	**Slope**
Small-world			
Core i7 970 (1 core)	1.00	1.00	1.02
Core i7 970: PThreads (6 cores)	7.06	7.06	1.01
Core i7 970: TBB (6 cores)	7.02	7.02	1.00
CellBE (1 PPE & 6 SPEs)	1.31	N/A	1.00
GTX295: kernel 1 (1 GPU)	5.02	1.00	1.02
GTX295: kernel 1 (2 GPUs)	8.24	1.64	0.98
GTX480: kernel 1 (1 GPU)	11.8	1.00	1.07
GTX480: kernel 1 (2 GPUs)	17.6	1.50	0.88
GTX480: kernel 1 (4 GPUs)	25.3	2.15	0.65
Scale-free			
Core i7 970 (1 core)	1.00	1.00	1.42
Core i7 970: PThreads (6 cores)	7.14	7.14	1.37
Core i7 970: TBB (6 cores)	7.23	7.23	1.32
CellBE (1 PPE & 6 SPEs)	2.80	N/A	1.21
GTX295: kernel 2 (1 GPU)	5.96	1.00	1.36
GTX295: kernel 2 (2 GPUs)	10.7	1.79	1.33
GTX480: kernel 2 (1 GPU)	12.2	1.00	1.38
GTX480: kernel 2 (2 GPUs)	19.7	1.61	1.30
GTX480: kernel 2 (4 GPUs)	33.1	2.71	1.17

4. Discussion & Conclusions

Power consumption and physical size constraints have led to a “slowing down of Moore’s Law” [13], [14] for processing devices at least in terms of conventional approaches using uniform and monolithic core designs. The consequence is that parallel computing techniques—such as incorporating multiple processing cores and other acceleration technologies—have become increasingly important [15].

Following Moore’s Law, the number of transistors on a GPU roughly doubled from the GT200 to the GT400 series graphics cards, which have become available within just under 2 years from each other. But more importantly, the effective performance achieved in our experiments has roughly doubled too. The many-core architecture of today’s GPUs has been shown [16] to significantly outperform traditional multi-core CPU architectures for algorithms that can be adapted to the specific requirements of the CUDA programming model.

Considering its age at the time of writing, the results of the Cell Broadband Engine are still quite impressive and show the potential of this hybrid CPU architecture compared to an architecture with fewer full-fledged cores. However, it requires considerably more effort to achieve good results when developing for the Cell than it does for an x86-based multi-core CPU.

In summary, we have implemented the clustering coefficient on a number of popular parallel architectures and discussed the design decisions necessary to achieve good performance and scaling on these platforms. We have used code fragments to highlight the differences in the implementations and explained architecture specific optimisation strategies. We have compared the runtime performance and scalability using both small-world and scale-free graphs.

We found that developing for multi-core CPUs using PThreads or TBB is much easier than developing for GPUs or the CellBE. This is partly due to the fact that compilers for x86-based processors have been around for much longer and are expected to perform most of the necessary low-level optimisations automatically, and partly due to the processors themselves having very sophisticated caching, pre-fetching and branch-prediction logic. The developer has to tackle a lot of these challenges explicitly when programming for the GPU or CellBE.

These efforts can yield very worthwhile performance enhancements. The GPUs dominated the benchmarks even though this algorithm is not particularly SIMD friendly, showing that even some graph-based, bandwidth limited algorithms can be implemented efficiently on their many-core architecture.

Recent GPU developments, like the automatic caching on NVIDIA's Fermi devices, have somewhat relaxed the demands placed on the developer and this trend is likely to continue with ever improving hardware and software. A feature recently introduced in CUDA toolkit 3.2 [17] even enables the allocation of dynamic memory in kernel code, which we intend to exploit in future work to generate graphs directly in graphics device memory.

References

[1] Graph500.org, "The Graph 500 List," http://www.graph500.org/, last accessed November 2010.

[2] H. Meuer, E. Strohmaier, H. Simon, and J. Dongarra, "36th list of top 500 supercomputer sites," www.top500.org/lists/2010/11/press-release, November 2010.

[3] M. E. J. Newman, S. H. Strogatz, and D. J. Watts, "Random graphs with arbitrary degree distributions and their applications," *Physical Review E*, vol. 64, no. 2, p. 026118, July 2001.

[4] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, June 1998.

[5] S. Milgram, "The Small-World Problem," *Psychology Today*, vol. 1, pp. 61–67, 1967.

[6] M. E. J. Newman, "The Structure and Function of Complex Networks," *SIAM Review*, vol. 45, no. 2, pp. 167–256, June 2003.

[7] K. Hawick, A.Leist, and D.P.Playne, "Mixing Multi-Core CPUs and GPUs for Scientific Simulation Software," *Res. Lett. Inf. Math. Sci.*, vol. 14, no. ISSN 1175-2777, pp. 25–77, 2010. [Online]. Available: http://www.massey.ac.nz/massey/learning/departments/iims/research/research-letters/

[8] D. J. d. Price, "Networks of Scientific Papers," *Science*, vol. 149, no. 3683, pp. 510–515, July 1965.

[9] A.-L. Barabasi and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, October 1999.

[10] R. Albert, H. Jeong, and A.-L. Barabasi, "The diameter of the world wide web," *Nature*, vol. 401, pp. 130–131, 1999.

[11] A. Leist, D. Playne, and K. Hawick, "Exploiting Graphical Processing Units for Data-Parallel Scientific Applications," *Concurrency and Computation: Practice and Experience*, vol. 21, pp. 2400–2437, December 2009, CSTN-065.

[12] K. A. Hawick, A. Leist, and D. P. Playne, "Parallel Graph Component Labelling with GPUs and CUDA," *Parallel Computing*, vol. 36, pp. 655–678, 2010. [Online]. Available: www.elsevier.com/locate/parco

[13] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. April, p. 4, 1965.

[14] S. K. Moore, "Multicore is bad news for supercomputers," *IEEE Spectrum*, vol. 45, no. 11, p. 11, 2008.

[15] G. Goth, "Entering a parallel universe," *Communications of the ACM*, vol. 52, no. 9, pp. 15–17, September 2009.

[16] K. Hawick, A. Leist, and D. Playne, "Regular Lattice and Small-World Spin Model Simulations using CUDA and GPUs," Computer Science, Massey University, Tech. Rep. CSTN-093, 2009, to appear in Int. J. Parallel Programming (2010).

[17] *NVIDIA CUDA^{TM} C Programming Guide Version 3.2*, NVIDIA® Corporation, 2010, last accessed December 2010. [Online]. Available: http://www.nvidia.com/