SDP Memo 095: Convolutional Gridding Routine: GPU port

Document number……SDP Memo 095
Document Type……...MEMO
Revision……..……1
Author……..……..S.F. Antao
Release Date……..……..2018-10-31
Document Classification…….. Unrestricted

Lead Author:
Samuel F. Antao, IBM Research UK

Released by:

Name	Designation	Affiliation
Jeremy Coles	SDP Project Manager	University of Cambridge
Signature & Date: Jeremy Coles Jeremy Coles (Oct 31, 2018)		

SDP Memo Disclaimer

The SDP memos are designed to allow the quick recording of investigations and research done by members of the SDP. They are also designed to raise questions about parts of the SDP design or SDP process. The contents of a memo may be the opinion of the author, not the whole of the SDP.

Convolutional Gridding Routine: GPU port

Samuel F. Antao, IBM Research, UK

October 23, 2017

1 Executive Summary

We received a serial implementation of a Convolutional Gridding Algorithm (CGA) [1] using the C language provided by Ania Brown from Oxford e-Research Centre. We were also provided with an existing GPU port in `https://github.com/OxfordSKA/GPU-gridding` already developed by the Numerical Algorithms Group (NAG) along with a report describing the changes [2]. Given that our main goal is to evaluate the performance improvements enabled by NVIDIA P100 GPU and NVLINK, we based our work on the version developed by NAG, which already contains extensive optimisation of the target algorithm for the same GPU.

The following summarises actions taken and obtained results:

- We were not provided new input data sets with the NAG code, therefore we employed the same provided with the serial version as they use the same naming convention in [2] - see Table 1.
- We employed an IBM Minsky server - dual-socket machine with 10 Power8 cores in each socket and a NVLINK bus connecting each socket to two NVIDIA P100 GPUs - in our experiments.
- The NAG code performed 10% slower out of the box, when compared with results obtained for an x86 server connected to a NVIDIA P100 GPU over PCIe reported in [2].
- Profiling showed that the performance downgrade was due to unoptimised data transfers to/from the GPU, which were not employing pinned memory (as opposed to pageable memory) for the host storage. This is not an issue for Power9-based servers as they support a shared address space and GPU memory is pageable.
- Switching to pinned memory for all the host data copied to/from the GPU resulted in a performance improvement in the range of 14.1-19.6% for all three input data sets. This is due to a performance increase of up to 3.66x in the data movements between host and GPU.

Table 1: Data sets used in the experiments.

Data set	**# Visibilities**	**# W-planes**	**Max. w_support**
D1: Synthetic	2152800	714	36
D2: SKA-LOW EL56-82	31395840	601	72
D3: SKA-LOW EL82-70	31395840	339	44

Table 2: Results summary for the different data sets.

	D1	**D2**	**D3**
Runtime (ms) x86+PCIe+NVIDIA P100 [2]	74	781	553
Runtime (ms) IBM Minsky	61.3	662	444
Runtime improvement	+17.1%	+15.2%	+19.6%
Estimated data-movement improvement	2.09x	3.66x	3.64x

- Limiting the number of registers per thread to 64, resulted in slightly higher GPU occupancy and in 1% performance increase.
- Final results summarised in Table 2.
- Man-effort to apply all changes to the provided code is less than 1 day.

1.1 Future work

The GPU code used in our experiments is written in CUDA. This means that it needed extensive rewrite and has limited portability. If code portability and code-base disruption is a major concern, novel programming models like OpenMP 4.5 can be employed instead [3].

2 Algorithm Overview

The algorithm is thoroughly described in [1]. In a nutshell, its goal is to project a visibility (sample) into a grid by convolving its value with a set of input kernels. This code is representative of some of the astronomy workloads that will support the Square Kilometre Array (SKA) currently being designed. Representative inputs were also provided with the code, along with expected outputs for verification of the results.

The target kernel is using single-precision floating point arithmetic. There is complex single-precision floating point arithmetic but it is implemented explicitly in the code, i.e. the programmer expanded the complex operations as a function of its real and imaginary parts without using any complex built-in types. Complex numbers are stored in memory with 8 consecutive bytes, being the first 4 bytes the real part and the last 4 bytes the imaginary part. The relevant kernel is presented in Listing 1.

The goal of this work is to evaluate and provide an optimised implementation of the CGA kernel for NVIDIA GPUs, in particular assess the contribution of NVLINK to the performance. The target machine for our experiments was a IBM Minsky server, which is a dual-socket machine with 10 Power8 cores in each socket and a NVLINK bus connecting each socket to two NVIDIA P100 GPUs. A GPU port has been made available to us [2], which consists of a major rewrite of the code, developed in CUDA. Given that this code was already highly

```
for (i = 0; i < num_points; ++i)
{
  /* ... */
  const float conv_conj = (ww_i > 0.0f) ? -1.0f : 1.0f;
  const size_t grid_w = (size_t)roundf(sqrtf(fabsf(ww_i * w_scale))
      );
  const int grid_u = (int)roundf(pos_u) + grid_centre;
  const int grid_v = (int)roundf(pos_v) + grid_centre;

  /* Get visibility data. */
  const float weight_i = weight[i];
  const float v_re = weight_i * vis[2 * i];
  const float v_im = weight_i * vis[2 * i + 1];

  /* Scaled distance from nearest grid point. */
  const int off_u = (int)roundf((roundf(pos_u) - pos_u) *
      oversample);
  const int off_v = (int)roundf((roundf(pos_v) - pos_v) *
      oversample);

  /* Get kernel support size and start offset. */
  const int w_support = grid_w < num_w_planes ?
      support[grid_w] : support[num_w_planes - 1];
  const size_t kernel_start = grid_w < num_w_planes ?
      grid_w * kernel_dim : (num_w_planes - 1) * kernel_dim;

  /* Catch points that would lie outside the grid. */
  if (grid_u + w_support >= grid_size || grid_u - w_support < 0 ||
      grid_v + w_support >= grid_size || grid_v - w_support < 0)
  {
    *num_skipped += 1;
    continue;
  }

  /* Convolve this point onto the grid. */
  for (j = -w_support; j <= w_support; ++j)
  {
    size_t p1, t1;
    p1 = grid_v + j;
    p1 *= grid_size; /* Tested to avoid int overflow. */
    p1 += grid_u;
    t1 = abs(off_v + j * oversample);
    t1 *= conv_size_half;
    t1 += kernel_start;
    for (k = -w_support; k <= w_support; ++k)
    {
      size_t p = (t1 + abs(off_u + k * oversample)) << 1;
      const float c_re = conv_func[p];
      const float c_im = conv_func[p + 1] * conv_conj;
      p = (p1 + k) << 1;
      grid[p]     += (v_re * c_re - v_im * c_im);
      grid[p + 1] += (v_im * c_re + v_re * c_im);
      sum += c_re; /* Real part only. */
    }
  }
  *norm += sum * weight_i;
}
```

Listing 1: Original serial code.

Table 3: Convolutional gridding results for other target platforms [2] - runtime in ms.

Architecture	**D1**	**D2**	**D3**
Intel E2670 (serial)	4980	37843	18109
Intel E2670 (24 threads + OpenMP atomics)	1040	14369	6685
Intel E2670 (24 threads + tilling)	198	2927	2211
Xeon Phi 7250 (136 threads + tilling)	560	4948	3579
Unknown x86 CPU + PCIe + P100 GPU + tilling	74	781	553

optimised for our target GPU (NVIDIA P100) we end up working on this code instead of the original serial code. We kept using the same input data sets provided with the original serial implementation.

3 Results verification

A set of input and expected-output files were provided with the original code. The result verification in place was prepared to allow any output that differ by less that ϵ, with ϵ the smallest number that added to 1.0 yields a result different than 1.0 in the target machine.

A lot of the performance improvement opportunities of the provided code have to do with parallelisation/threading as well as compile-time optimisations, which will break the order of the floating-point operations and therefore introduce rounding errors that will result in differences much larger than ϵ. Therefore, we assume it is acceptable to relax the maximum error of the results and moved to monitor the outputs by comparing with the results of optimised serial implementations of the original code.

We were not provided compatible input files for the CUDA implementation provided by NAG, therefore we adapted the code to be compatible with the inputs we were given originally. We used the verification strategy mentioned above for this code, otherwise it would have failed verification due to rounding errors as well.

The inputs and outputs provided refer to the data sets in Table 1.

We were also provided a set of performance results for three different platforms that we can use to assess the performance of the obtained implementations. These results are replicated in Table 3.

4 CUDA acceleration

We were provided an optimised version of the code using the CUDA language for NVIDIA GPUs. Full details of this code are provided in [2]. In a nutshell, the CUDA implementation divides the grid in tiles and evaluates which visibilities contribute to updates in that tile. As it can be seen in Listing 1, lines 48 and 49, the grid array is updated $2\texttt{w\_support}^2$ times for each visibility, therefore is important to explore locality when this array is updated. To decrease the number of global memory accesses in the GPU, the updates in a tile are initially done on shared and local memory before being stored back to the grid array. Shared and local memory are very efficiently accessed from the GPU stream processor, so this strategy provides a significant performance boost.

Table 4: Vanilla CUDA implementation performance for Minsky - runtime in ms.

	D1	**D2**	**D3**
Minsky (CUDA as in [2])	82	857	613
Improvement to Table 3 (%)	-11.8	-9.7	-10.8

We tested the vanilla version of the code that was provided on a Minsky server using CUDA 9RC. XL C++ was used has host compiler. The following compiler flags were used on Nvidia compiler, nvcc:

- `-ccbin xlc++_r -std=c++11 -arch=sm_60 -Xptxas -v -O3 -g -restrict -use_fast_math -D_FORCE_INLINES -D_DEBUG -lineinfo -Wno-deprecated-declarations`

We just had to insert small changes so that we could use the input files provided with the serial code: these changes consisted of adjusting the data types loaded from the input file to the expected size as the code was prepared to load 4-byte integers instead of 8-byte, and double-precision instead of single-precision values for some input quantities.

The results obtained are reported in Table 4 and compared with results obtained for P100 and x86 in Table 3. We can see there is a performance downgrade of about 10%.

4.1 Using pinned memory to store host data

Profiling shows that the downgrade is connected with low bandwidth that we are getting from CUDA memory copies, we can measure $\tilde{4}.5$ GB/s but we would expect $\tilde{3}2$ GB/s, which is the approximate bandwidth we would get from NVLINK after subtracting package payload overheads in the NVLINK protocol.

The GPU driver requires the host storage it copies data to/from to be in physical memory at all times, so that storage cannot be paged-out by the operating system during the transfer. Therefore, the associated pages have to be *pinned*. The default implementation is to have the CUDA runtime library copy the data from its pageable memory to a buffer in pinned memory and then have the driver copy it from there. This extra copy from pageable to pinned memory adds overhead to the whole data movement.

There are two ways typically used to improve performance in these cases: *a)* chunk the data to copy and use threading to do the copy of the different chunks (threading will improve the bandwidth of the pageable-pinned data copy), and *b)* allocate pinned memory on the host in the first place to store data that will eventually be moved to the GPU (this avoids the copy completely). In our experiments we used *b)* given that all the data sets, which total less than 3 GB, can easily fit the system's physical memory (512 GB). This strategy would have to be revisited for data sets whose size starts to be close to the system's total memory, as having many pinned pages will decrease the efficiency of the operating systems' page management.

In order to use pinned memory we replaced all the calls to `calloc` and `free` with the calls to the two definitions in Listing 2, respectively. In addition to that, we created an allocator to use with the vector declarations from the C++ STL library whose storage is also copied to the GPU. Listing 3 shows the definition

```
extern "C" void *mycalloc(size_t elems, size_t size) {
  void *ptr = nullptr;
  CHECK( cudaMallocHost(&ptr, elems*size));
  assert(ptr);
  memset(ptr,0,elems*size);
  return ptr;
}

extern "C" void myfree(void *ptr) {
  CHECK( cudaFreeHost(ptr) );
}
```

Listing 2: Definitions that allocate and release pinned memory. These definitions have the same signature as `calloc` and `free`.

Table 5: CUDA implementation timmings for Minsky when pinned memory is used - runtime in ms.

	D1	**D2**	**D3**
Minsky (CUDA with pinned memory)	61.8	671	444
Improvement to Table 3 (%)	+16.5	+14.1	+19.6

of the allocator and how it was used in the vector declarations. The timings and performance improvements for the version of the code that uses pinned memory are reported in Table 5. Profiling of this version shows that we can consistently get approximately 32 GB/s bandwidth during the data movements. In order to assess the effect of NVLINK in the data movements, we obtained the ratio of compute/data-movements for the three data sets. This allows us to estimate the difference in bandwidth we are getting compared to the reference results in Table 3. By assuming that the whole difference in performance is due to improved data movements (the GPU is the same) we can deduct the differences in timing of the several runs from the data-movement component and with that estimate the improvement in the data-movements due to NVLINK in the Minsky server. Results are summarised in Table 6. We can see that for D2 and D3 - the larger data sets - we get a consistent improvement of 3.6x.

We have no indication on whether the results in [2] used pinned memory, or the exact configuration of the GPU in the system as well as the version of the PCIe bus. However we estimate a PCIe maximum bandwidth of about 12 GB/s which is the same we observe in Power8 with K80 GPUs connected with PCIe. This would yield about 2.6x more bandwidth with NVLINK. Its possible the remaining 1.0x is due to the pageable-pinned memory copies in the x86 + P100 machine.

Note that all the performance implications related to data movements to/from pageable/pinned memory won't be relevant anymore for Power9 platforms, as the hardware will support pageable memory for the GPU as well.

Table 6: Improvement in data movement.

Metric	**D1**	**D2**	**D3**
% Compute time	84.9	94.7	92.6
% Data-movement time	15.1	5.3	7.4
Estimated improvement of data-movement compared to Table 3	2.09x	3.66x	3.64x

```
namespace
{
  template <typename T>
  class myallocator: public std::allocator<T>
  {
  public:
    typedef size_t size_type;
    typedef T* pointer;
    typedef const T* const_pointer;

    template<typename _Tp1>
    struct rebind
    {
      typedef myallocator<_Tp1> other;
    };

    pointer allocate(size_type n, const void *hint=0)
    {
      return (pointer)mycalloc(n, sizeof(T));
    }

    void deallocate(pointer p, size_type n)
    {
      myfree(p);
    }

    myallocator() throw(): std::allocator<T>() { }
    template <class U>
    myallocator(const myallocator<U> &a) throw(): std::allocator<T
        >(a) { }
    ~myallocator() throw() { }
  };
}

/* ... */

std::vector<float2, myallocator<float2>> compacted_wkernels;
```

Listing 3: Allocator that uses pinned memory and a STL vector instantiation using this allocator.

Table 7: CUDA implementation timmings for Minsky when pinned memory is used and registers limited to 64 - runtime in ms.

	D1	**D2**	**D3**
Minsky (CUDA with pinned memory and 64 registers per thread)	61.3	662	444
Improvement to Table 3 (%)	+17.1	+15.2	+19.6

4.2 Limit number of register

Using nvcc V9.0.151, the main CUDA kernel, `oskar_process_tiles_excluding_box`, is implemented with 72 registers per thread. Like other resources in the stream multiprocessor, registers affect the amount of CUDA threads (and therefore blocks), that can run at a given time. If each thread uses 64 or less registers, registers will no longer affect the occupancy, as there will be enough registers to run the maximum number of threads the hardware can allow. Therefore we used the flag `--maxrregcount=64` with nvcc to limit the number of registers per thread. The results are summarised in Table 7. We got a performance improvement of about 1% for D2 - the data set that requires more compute time. This is due to an improvement of 3% in occupancy.

References

[1] Cornwell, Golap & Bhatnagar (2008), *The Noncoplanar Baselines Effect in Radio Interferometry: The W-Projection Algorithm*. IEEE Journal of Selected Topics in Signal Processing, Vol. 2, Issue 5, p.647

[2] Dingle, du Toit & Hopkins, *Convolution Gridding on CPU, GPU and KNL*. The Numerical Algorithms Group, Manchester, May 4, 2017

[3] OpenMP 4.5 specification, `http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`.