TECHNICAL MEMORANDUM

805

An OpenACC GPU adaptation of the IFS cloud microphysics scheme

Huadong Xiao[1], Michail Diamantakis and Sami Saarinen

Research Department

[1]Visiting Scientist, National Meteorological Information Center, China Meteorological Administration (CMA)

June 2017

European Centre for Medium-Range Weather Forecasts
Europäisches Zentrum für mittelfristige Wettervorhersage
Centre européen pour les prévisions météorologiques à moyen

Series: ECMWF Technical Memoranda

A full list of ECMWF Publications can be found on our web site under:
http://www.ecmwf.int/en/research/publications

Contact: library@ecmwf.int

European Centre for Medium Range Weather Forecasts
Shinfield Park, Reading, Berkshire RG2 9AX, England

Abstract

The ECMWF Integrated Forecast System (IFS) cloud microphysics scheme has been adapted for a GPU architecture. Hybrid OpenMP and OpenACC within a single node, hybrid MPI and OpenACC over multiple nodes as well as different algorithmic and code optimization methods were employed to study the performance impact. The roofline model was used to conduct a performance analysis and the CLAW compiler has been explored as a tool for automatic code adaptation. For a very large number of grid columns, the double precision performance of the 4 GK210 GPUs of a single node was slightly better than the performance of two 12-core CPUs (contained in the node) in terms of the total run rime. However, without taking into account the GPU data transfer and other overheads, the actual calculation time for the same large size problem was reduced to approximately one quarter of the CPU time giving a speed up factor of 4. Comparing the performance of a single GPU with a single CPU, the obtained speed up factor is approximately 2. A further 40% gain can be achieved with single precision. The obtained GPU speed up factor depends a lot on the workload given to a GPU; for small or moderate size problems (number of grid columns) the above mentioned speed up factors cannot be achieved.

1 Introduction

The cloud microphysics scheme is one of the most computationally expensive parts in the IFS model (Saarinen, 2014, 2015). Computation of the cloud scheme (CLOUDSC) is dependent only in the vertical direction (column mode). When a high resolution model runs on a parallel supercomputer a very large number of grid point columns, allocated in groups of separate MPI tasks, can be processed simultaneously. GPUs can be used in order to accelerate further this calculation. GPUs are attractive due to their low energy to performance ratio being one of the most energy efficient architectures as indicated by “The Green 500” supercomputer list: https://www.top500.org/green500/. The question is if this can be exploited in a time-critical application such as the global weather prediction system of ECMWF.

The CLOUDSC routine was ported and tested on the LXG GPU test cluster of ECMWF which runs on SUSE Linux Enterprise Server 11 SP4 and consists of 34 nodes. Each compute node of this cluster contains two Intel Xeon E5-2690v3 (Haswell) CPU sockets with 12 cores per socket and 2 NVIDIA K80 cards with 2 GK210 GPUs per card. The configuration of each compute node is shown in Table 1. The compiler and its flags used were given in Table 2.

Different GPU adaptation approaches have been tried for CLOUDSC. A combination of OpenMP with OpenACC programming was first tested followed by a hybrid MPI and OpenACC programming approach. At the same time, different optimization methods were employed to improve computing performance, for example:

1. Hoisting local arrays with global ones.
2. Algorithm adjustment to utilize batched CUBLAS library.
3. Changes of water species sorting method.
4. Other GPU-related tuning methods such as loop interchange.
5. Single precision computation.

Table 1. *Configuration of a node in LXG cluster*

Type	Sockets (SM)	Cores/CPU	Cache(MB) Mem(GB)	Processor type
CPU(Haswell)	2	12	30 128	Intel Xeon E5-2690 v3 @ 2.60GHz AVX2 (Haswell)
GPU(2×K80)	(26)	4992 SP 1664 DP	3.0 24	Tesla K80 3.7 @ 823 MHz (Mem 2505MHz) CUDA 8.0

Table 2. *Compiler and its flags used in the experiment*

Node	Type	Compiler	Compiler flags used
Dell	CPU	PGI 16.7	-O3 -fast -Minfo -mp=bind,allcores,numa -Ktrap=fp -Mbyteswapio -Kieee -Mdaz -Mfprelaxed
	GPU	PGI 16.7	-O3 –fast -acc -Minfo -ta=nvidia:7.5,kepler -tp=haswell -Mvect=simd:256 -mp=bind, -mp=bind,allcores,numa -Mbyteswapio -Ktrap=fp -Kieee -Mdaz

In section 2, we give a brief introduction to IFS cloud scheme. In section 3, we summarize performance results of CLOUDSC for CPUs compared with GPUs. In section 4 the OpenACC parallelization method is described. In section 5 detailed results and comparisons are given and in section 6 we report our conclusions. Three appendices are included in the report showing: (i) detailed results with respect to the sensitivity with blocking size NPROMA (ii) summarising preliminary experience with the code adaptation tool CLAW and (iii) finally performance analysis of a roofline model.

2 The IFS cloud scheme

The IFS cloud scheme is a parametrization of cloud processes for prognostic cloud. It is a multi-species prognostic microphysics scheme, with 5 prognostic equations for water vapour, cloud liquid water, rain, cloud ice and snow. The equation governing each prognostic cloud variable within the cloud scheme is

$$\frac{\partial q_x}{\partial t} = A_x + \frac{1}{\rho}\frac{\partial}{\partial z}(\rho V_x q_x) \quad (1)$$

where, q_x is the specific water content for category x (so x = 1 represents cloud liquid, x = 2 for rain, and so on), A_x is the net source or sink of q_x through microphysical processes, and the last term

represents the sedimentation of q_x with fall speed V_x (for details see Forbes et al., 2011; Forbes and Ahlgrimm, 2012). A schematic of the IFS cloud scheme is given Figure 1.

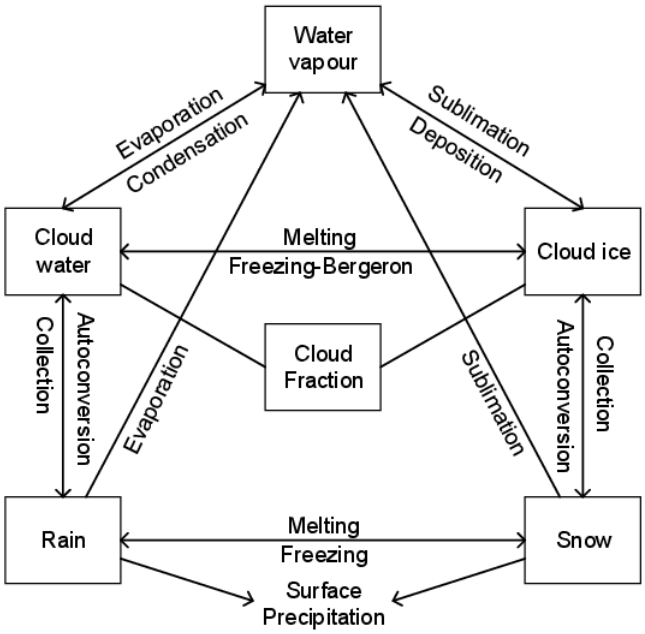

Figure 1. Schematic of the IFS cloud scheme (from Forbes et al, 2011).

The five prognostic equations for the individual species are solved using a simple forward-in-time, implicit solver approach. The solution to this set of equations uses the upstream approach, which utilizes the forward difference quotient in time and the backward difference quotient in space (ECMWF, 2017). In the scheme, after calculating initial input profiles and initial setup, the sources and sinks variables are calculated, followed by the precipitation processes and numerical solvers, then tendencies of all the thermodynamic quantities updated, and finally the calculated flux for diagnostics and budget terms.

3 Performance comparison: CPU versus GPU

Testing showed that the total run time of a compute node with 4 GPUs is 11% less than the total run time of a compute node with 2 CPUs (with 24 cores each) for a large size problem in double precision. Furthermore, without taking into account the GPU data transfer and other overheads, the actual calculation time, for the same large size problem, is reduced to approximately a quarter of the CPU time i.e. a speed up factor of 4. Comparing the performance of a single GPU with a single CPU, the obtained speed up factor is approximately 2. The detailed results of computation with double precision for 160,000 and 320,000 grid columns using OpenMP and OpenACC directives are given in Table 3. The chosen NPROMA corresponds to the optimal setting for the given number of threads and GPUs. Assigning 320,000 grid columns per GPU node gives a performance near the peak. The highest number of grid columns that can be used in a single LXG node is 1,280,000 – our tests indicated that performance peaked at this number of columns (not shown here). Although this performance may be optimal for a GPU, using such a large number of columns is unrealistic for a NWP forecast which should be completed in a relatively short time of the order of 1 hour. Therefore, we will not expand further on these results and we will use the more modest but rather large set up of 160,000 grid columns.

Table 3. *Time and Gflops/s on CPU and GPU platform for different number of OpenMP threads, GPUs, NPROMA and grid columns with double precision (here speedup is defined as the ratio of the* CPU total *run time of a single CPU with 12 cores to that of two CPUs with 24 cores or the GPU actual calculation time).*

Grid columns	OpenMP Threads	GPUs	NPROMA	Time(ms)			Gflops/s	Speedup
				Calculation	Overhead	Total		
160000	12	-	48			2105	9.48	1
	24	-	12			1286	15.53	1.63
	1	1	80000	1037	2507	3545	5.63	2.02
	2	2	80000	555	1529	2086	9.57	3.79
	4	4	40000	376	799	1174	17.00	5.59
320000	12	-	48			4127	9.68	1
	24	-	12			2470	16.17	1.67
	1	1	80000	2056	4999	7057	5.66	2.00
	2	2	80000	1064	3126	4192	9.54	3.87
	4	4	80000	641	1560	2193	18.20	6.43

It is worth mentioning that when the number of NPROMA-block was increased, the GPU calculation time reduced remarkably. Generally, the GPU calculation time is the shortest when NPROMA-block is set up to 80,000 which is the number giving the best fitting in GPU's memory for NVIDIA K80 GPU. The computational performance of CLOUDSC is also sensitive to the number of NPROMA-blocks. In most cases, the computational performance of CLOUDSC on CPUs with 12 or 24 OpenMP threads is optimum when NPROMA is between 12 and 128 (see Appendix A).

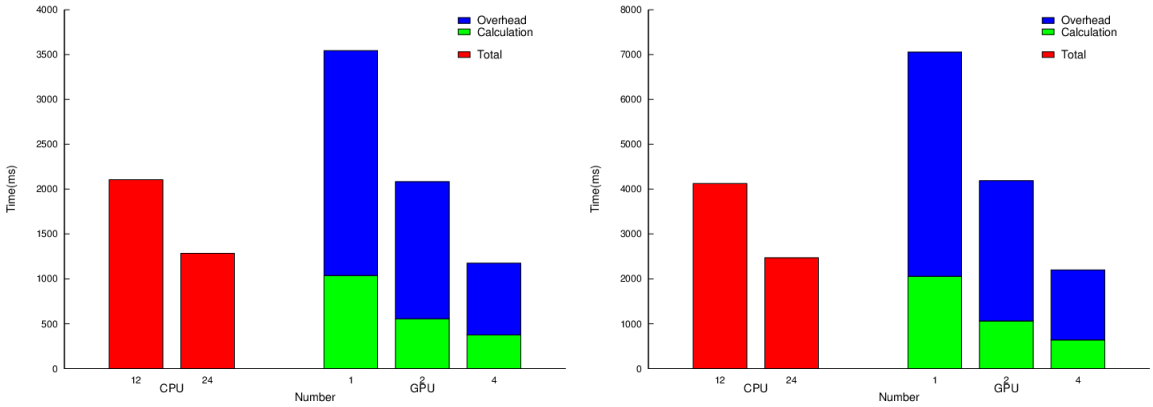

Figure 2. *Comparison of CPU total run time, GPU calculation time, and GPU overhead time for grid columns of 160000(left) and 320000(right)*

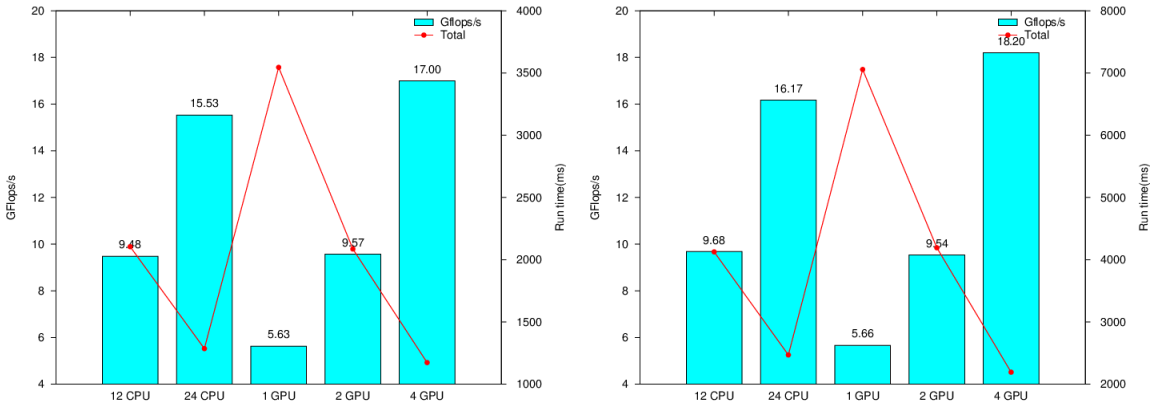

Figure 3. *Comparison of performance and total run time on CPUs and GPUs for grid columns of 160000(left) and 320000(right)*

4 Parallelization with OpenACC directives

4.1 Implementation

A simple shell script was written to insert OpenACC directives like ACC KERNELS or ACC PARALLEL in the front of code loops (see Listing 1). Furthermore, ACC COPY or ACC PRESENT clauses have been manually added in the beginning of real computation. Through these two steps an initial GPU based accelerated version can be obtained. Furthermore, setting a bigger NPROMA size and runtime environment variables like buffer-size, using memory binding, and other methods like asynchronous computing and data movement have been attempted to attain an optimized performance.

Because the current version of PGI Accelerator Compiler does not support derived type of allocated array and pointer association operation efficiently (Norman et al., 2015), the codes using derived type and pointer operations should be rewritten first. Following that, 38 input arrays were requested to be copied from CPU memory to GPU memory using ACC DATA COPYIN clause, and 22 output arrays were requested to be copied from GPU memory to CPU memory through using ACC DATA COPYOUT clause. The array variables could be obtained through the "intent" attribute from array definition in the

subroutine of "cloudsc". The automatic arrays in the subroutine of "cloudsc" were assigned to GPUs and released after the "cloudsc" call via additional ACC DATA CREATE layer.

```
#!/bin/bash
file=$1
sed -e 's/SUBROUTINE CLOUDSC/SUBROUTINE CLOUDSC_ACC/g' \
    -e 's/tendency_\([a-zA-Z]\+\)%\([a-zA-Z]\+\)/pstate_\2_\1/g' \
    -e '1,/START OF VERTICAL LOOP/ {
        s/^DO /\n!$ACC KERNELS\n!$acc loop\n&/g
        s/^[ \t]\+DO /\n!$acc loop\n&/g
        s/^ENDDO/&\n!$ACC END KERNELS\n/g
        }' \
    -e '/END OF VERTICAL LOOP/,$ {
        s/^DO /\n!$ACC KERNELS\n!$acc loop\n&/g
        s/^[ \t]\+DO /\n!$acc loop\n&/g
        s/^ENDDO/&\n!$ACC END KERNELS\n/g
        }' \
    -e '/START OF VERTICAL LOOP/,/END OF VERTICAL LOOP/ {
        s/^  DO /\n!$ACC KERNELS\n!$acc loop\n&/g
        s/^  [ \t]\+DO /\n!$acc loop\n#endif\n&/g
        s/^  ENDDO/&\n!$ACC END KERNELS\n/g
        }' \
$file
```

Listing 1. *Scripts to insert* OpenACC *directives*

One of the most computationally expensive subroutines in the IFS model is "cuadjq" also called by the subroutine of "cloudsc". Fortunately just few lines of code were used and these were inlined to "cloudsc" in order to obtain better performance.

Regarding the accuracy of results, the relative error was in the range of tolerance of machine precision (double precision).

4.2 Optimization

In order to analyse and improve performance, the NVIDIA profiler tool of "nvprof" was used for hotspot and bottle-neck analysis of the routine. Interchanging the JL-outer loop with the inner JM-loop index (see Listing 2, 3 of Appendix B) reduced calculation time. In most cases, performance could be further improved by increasing the compiler's default vector size to improve GPU occupancy. Replacing temporary arrays by scalar, especially 3D arrays, was also an effective way to boost performance both in terms of calculation time and overhead. Other methods such as loop fusion to reduce computation kernels contribute to further improvement of performance. More specifically, based on the original implementation for implicit solver, by only changing the vector size from default 128 to 512 could reduce by 10% the calculation time for the Gaussian elimination. Furthermore replacing a 3D array with a scalar (see Listing 2), an additional 5% gain can be achieved.

```
  ! Non pivoting recursive factorization
  DO JN = 1, NCLV-1  ! number of steps
     DO JM = JN+1,NCLV ! row index
 ZQLHS(KIDIA:KFDIA,JM,JN)=ZQLHS(KIDIA:KFDIA,JM,JN)
       &                  / ZQLHS(KIDIA:KFDIA,JN,JN)
      DO IK=JN+1,NCLV ! column index
        DO JL=KIDIA,KFDIA
          ZQLHS(JL,JM,IK)=ZQLHS(JL,JM,IK) -
&                   ZQLHS(JL,JM,JN)*ZQLHS(JL,JN,IK)
        ENDDO
      ENDDO
    ENDDO
  ENDDO
  ! Backsubstitution step 1
  DO JN=2,NCLV
    DO JM = 1,JN-1
      ZQXN(KIDIA:KFDIA,JN)=ZQXN(KIDIA:KFDIA,JN)   -
&     ZQLHS(KIDIA:KFDIA,JN,JM)*ZQXN(KIDIA:KFDIA,JM)
    ENDDO
  ENDDO
```

```
! Non pivoting recursive factorization
DO JN = 1, NCLV-1  ! number of steps
  DO JM = JN+1,NCLV ! row index
    xmult =ZQLHS(JL,JM,JN)
    &     / ZQLHS(KIDIA:KFDIA,JN,JN)
    DO IK=JN+1,NCLV ! column index
      ZQLHS(JL,JM,IK)=ZQLHS(JL,JM,IK) -
       &             xmult *ZQLHS(JL,JN,IK)
    ENDDO
    ! Backsubstitution
    ZQXN(JL,JN)=ZQXN(JL,JN)- xmult * ZQXN(JL,JM)
  ENDDO
ENDDO
```

Listing 2. *Code segment for replacing a 3D array with a scalar (left: original, right: modified)*

The solver for the microphysics consumes most of the time of the GPU CLOUDSC except for the overhead part which includes the data movement between host and device (see Figure 4). Data movement is the biggest contributor in the overhead. An implicit solver for many small linear systems with Gaussian elimination method is used in the original implementation. For a matrix, LU (Lower and Upper triangular Matrix) factorization is first done, followed by back substitution, then the final result is obtained.

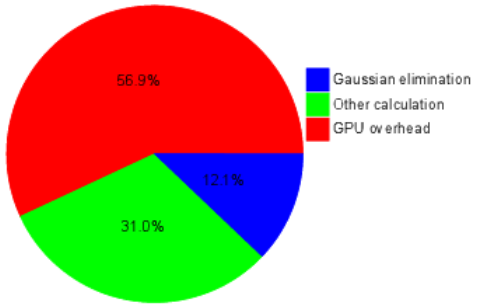

Figure 4. *The computational cost distribution of GPU accelerated CLOUDSC on a single GPU with 80,000 of NPROMA-block for 160,000 grid columns.*

There are a large number of small linear equation systems (5×5) that need to be solved in CLOUDSC. Calling a batched Gaussian elimination routine from CUBLAS library was attempted to find if any performance increase could be gained. Currently, some CUDA Fortran features (like type definition attribute with “device” and “c_devptr”, and function of “c_devloc”) need to be used to call batched

CUBLAS routine successfully. But the performance result is much worse than without calling CUBLAS routine.

5 Detailed performance results and comparisons

In this section we expand the results presented in section 3, and we provide detailed comparisons for each attempted optimization.

The results were obtained by averaging 5 separate runs using 160,000 grid columns (NGPTOT). For the double precision version of the program, a complete list of experiments were performed, while for the single precision version of program, two selected experiments were performed as a simple comparison to indicate the impact. In general, the conclusion derived from computations with various grid columns holds independently of the number of grid columns.

5.1 Results of the original CLOUDSC on CPU host

The CPU version of CLOUDSC, has been tested using different number of OpenMP threads and NPROMA-blocks. The total run time and sustained performance (Gflops/s) for different number of CPU threads and an optimum NPROMA-block for the number of threads are shown in table 4.

Table 4. *Total run time, Gflops/s of the original CLOUDSC computation on an LXG cluster node for different number of OpenMP threads and 160,000 grid columns. For each number of OpenMP threads listed, the corresponding NPROMA is the one that gives the best performance.*

OpenMP threads	NPROMA	Total time(ms)	Gflops/s
1	128	19869	1.00
4	64	5305	3.76
8	64	2954	6.76
16	48	1711	11.67
24	12	1285	15.53

Irrespective to the change of number of NPROMA-blocks, the total run time was decreased when the number of OpenMP threads was increased. Good parallel scaling efficiency is achieved which decreases gradually as the number of threads increases.

5.2 Results of accelerated CLOUDSC on GPU

5.2.1 Hybrid OpenMP and OpenACC directives

In this section results from the version of accelerated CLOUDSC on GPU using hybrid OpenMP and OpenACC directives are first shown, followed by another version where the original Gaussian elimination method is replaced by calling CUBLAS library. The implementation using hybrid OpenMP and OpenACC directives with calling CUBLAS library depends on CUDA Fortran features of device

type definition attribute of “device”, “c_devptr”, and functions of “c_devloc”, “cudaDeviceSynchronize”, “cublasDgetrfBatched”, “cublasDgetrsBatched”.

Table 5. *Time and Gflops/s of the GPU accelerated CLOUDSC computation for different number of OpenMP threads, GPUs, NPROMA and 160,000 grid columns. The listed NPROMA is the one that gives the shortest calculation time and biggest number of Gflops/s. Results repeated using CUBLAS library.*

GPUs	OpenMP threads	NPROMA	Time(ms)			Gflops/s
			Calculation	Overhead	Total	
1	1	80000	1037	2507	3545	5.63
	2	40000	1631	2092	3485	5.73
	4	20000	1805	1981	3535	5.65
2	2	80000	555	1529	2086	9.57
	4	40000	955	1242	2104	9.49
	8	20000	995	1354	2182	9.15
4	4	40000	376	799	1174	17.00
	4	20000	432	822	1254	15.92
CUBLAS LIBRARY RESULTS						
1	1	80000	1976	2509	4486	4.45
	2	40000	2535	2091	4347	4.59
	4	20000	3487	1479	4577	4.36
2	2	80000	1120	1528	2648	7.54
	4	40000	1522	1308	2647	7.54
	8	20000	1883	1124	2821	7.08
4	4	40000	949	794	1738	11.49
	4	20000	1060	810	1864	10.71

For both implementations, different number of GPUs, number of OpenMP threads and NPROMA-blocks were chosen. Computation on each GPU is driven by one or more OpenMP threads. The GPU calculation time, overhead time, total run time and sustained performance (Gflops/s) for different combination of settings are shown in Table 5.

The shortest GPU calculation time is achieved when one OpenMP thread controls one GPU for the optimal setting of 80,000 NPROMA for a specified GPU number. When using time-sharing GPU mode (i.e. more than one OpenMP threads drives one GPU), the GPU overhead time including data transfer overhead is reduced, but the actual GPU calculation time is increased. And there is little difference in the total time between using or not using GPU time-sharing. Only considering the GPU calculation time, the performance scales very well with the increasing number of GPU.

The hybrid OpenMP and OpenACC version with calling CUBLAS library shows an apparent degradation of performance especially in terms of GPU calculation time, when compared with the equivalent version without CUBLAS (see Table 5). The overhead time is almost the same given that codes changes are only applied in the computation of Gaussian elimination method.

In some of the entries of Table 5, there is a small difference between the total time and the sum of the GPU calculation time and the GPU overhead time. This occurs when more than one thread is used to control a GPU. The reason is that the reported time is the minimum time for each thread. Each thread time is the average of five runs.

5.2.2 Hybrid MPI and OpenACC directives

In this section, results from the GPU adaptation with hybrid MPI and OpenACC directives is shown first, followed by another version with hoisting of local arrays. Hoisting of local arrays means moving the local definition of arrays in a subroutine to another module file including statements of allocatable array definitions, allocating memory and deallocating memory. This may improve performance, especially when a subroutine with a large number of local arrays is called several times at each execution.

Computation in each GPU is controlled by an MPI task. The GPU calculation time, time of hoisting local arrays, overhead time, total time and sustained performance (Gflops/s) for different combination numbers of GPUS, MPI tasks, nodes and NPROMA-blocks are demonstrated in table 6 and table 7, respectively.

Table 6. *Time and Gflops/s for the GPU CLOUDSC for 160,000 grid columns with different number of MPI tasks, GPUs per node, nodes and an optimum NPROMA for each configuration.*

MPI tasks	GPUs per node	Nodes	NPROMA	Time(ms)			Gflops/s
				Calculation	Overhead	Total	
1	1	1	80000	1102	2485	3588	5.57
2	1	2	80000	562	1250	1813	11.02
	2	1	80000	568	1537	2106	9.48
4	1	4	40000	319	657	977	20.47
	2	2	40000	314	766	1080	18.47
	4	1	40000	342	806	1143	17.49

As we can see in Tables 6, 7 when multiple MPI tasks are employed on the same node, the overhead time is longer than the time when MPI tasks are distributed over different nodes with the same

combination of MPI tasks, GPUs and NPROMA. The possible reasons lie in two aspects. One is that each MPI task could only use part of the whole node's memory when several MPI tasks are executed on the same node. The other is that the system overhead would be less when fewer MPI tasks per node are used. The performance of hybrid MPI and OpenACC version of CLOUDSC also scales very well, much like that of hybrid OpenMP and OpenACC version.

The impact of hoisting local arrays is small (see Table 6). With the available hardware and PGI compiler environment, hoisting seems not an effectively optimization method for the GPU computation of CLOUDSC.

Table 7. *Time and Gflops/s of the GPU accelerated CLOUDSC computation with hoisting local arrays.*

MPI tasks	GPUs per node	Nodes	NPROMA	Time(ms)				Gflops/s
				Calculation	Hoisting	Overhead	Total	
1	1	1	80000	1090	218	2498	3808	5.24
2	1	2	80000	557	232	1364	2151	9.29
	2	1	80000	556	580	1640	2751	7.27
4	1	4	40000	338	215	693	1245	16.04
	2	2	40000	338	562	873	1706	11.71
	4	1	40000	348	1305	1042	2651	7.56

5.3 Single precision results

The output atmospheric parameters in single precision differ from that in double precision. The size of the difference for some variables can be up to 1 percent with respect to their magnitude in double precision. Larger differences can be noticed between double precision CPU results and single precision GPU results. Two are the possible reasons. The first is that the input data used in single precision are transformed directly from the input data provided in double precision that produced rounding off errors. The second and perhaps more important reason is that the accelerators available today support most of the IEEE floating-point standard but they do not support all the rounding modes and some operations. However, the performance results in single precision could be referred.

5.3.1 Results of the original CLOUDSC on CPU

The total run time and sustained performance (Gflops/s) for different number of CPU threads and an optimum NPROMA-block are shown in Table 8 and can be compared with Table 4. A similar conclusion to double precision can be drawn: regardless what the number of NPROMA-blocks was, the total time was decreased when the number of OpenMP threads was increased. Furthermore, the calculation time was approximately reduced by one third compared with double precision.

Table 8. *Total run time, Gflops/s of original CLOUDSC computation with the best NPROMA for each OpenMP thread setting on an LXG cluster node for 160,000 grid columns.*

OpenMP threads	NPROMA	Total time(ms)	Gflops/s
1	100	13552	1.47
4	128	3499	5.71
8	128	1959	10.19
16	64	1102	18.10
24	64	864	23.10

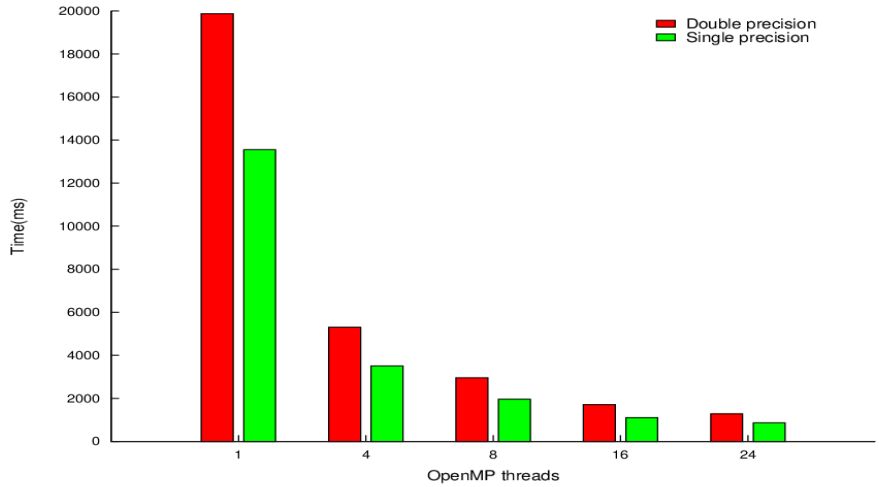

Figure 5. C*omparison of total run time for CLOUDSC computation on a CPU node with single precision and double precision for 160000 grid columns.*

5.3.2 Results of GPU accelerated CLOUDSC with hybrid OpenMP and OpenACC

Computation on each GPU is controlled by one or more OpenMP threads. The GPU calculation time, overhead time, total run time and sustained performance (Gflops/s) for different combination number of GPUs, OpenMP threads and NPROMA blocks are shown in Table 9.

Table 9. *Time and Gflops/s of the GPU CLOUDSC with different number of GPUs, OpenMP threads, NPROMA and 160,000 grid columns (NPROMA=160,000 is the maximum number can be used in the GPU version of CLOUDSC due to the memory limit of NVIDIA K80 card).*

GPUs	OpenMP threads	NPROMA	Time(ms)			Gflops/s
			Calculation	Overhead	Total	
1	1	160000	618	1255	1875	10.65
	2	80000	962	888	1851	10.79
2	2	80000	367	772	1138	17.54
4	4	40000	284	422	701	28.50

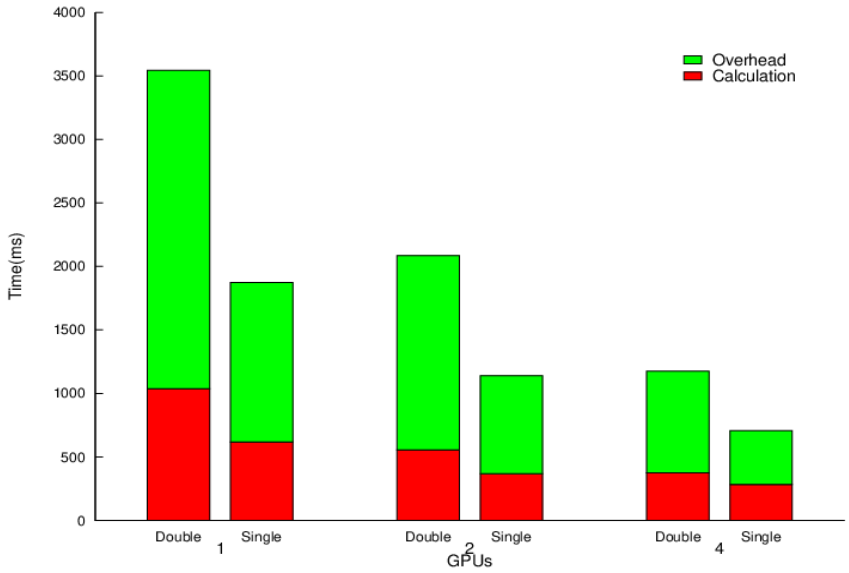

Figure 6. *Comparison of time for hybrid OpenMP and OpenACC of CLOUDSC computation on a GPU node with single precision and double precision for 160,000 grid columns.*

Compared with the time in double precision, if the work load of GPUs is saturated fully, both the time of GPU calculation and overhead in single precision reduced as the number of GPU was increased, and the time reduced about 40%. The amplitude of GPU calculation time reduction in the computation of GPU accelerated CLOUDSC between single precision and double precision for the grid column of 160,000 in the right of Figure 5 is not as big as that in the left one. Since NVIDIA K80 GPU in singe precision can fit at most 160,000 of NPROMA-block once, computation for the 160,000 grid column with more than one GPU cannot be fully exploited the power of GPUs.

6 Conclusions

The key conclusions of this study are:

(1) Computational performance for CLOUDSC on both CPU and GPU platforms scales well with the number of CPU cores and GPU devices respectively.

(2) The performance is sensitive to the size of NPROMA-block. The CPU version favours a relative small size of NPROMA-block between 12 and 128 while the GPU version favours a large size of NPROMA-block over 10,000.

(3) Interchanging the loops with the largest number of iterations is the most effective way to improve computational performance on GPU.

(4) Optimizations such as hoisting local arrays and calling batched CUBLAS library to perform a large number of small matrix operations did not show any benefit in the context of a stand-alone code where the cloud scheme is called only once. However, such optimizations may be beneficial in a realistic simulation where multiple time steps are performed. In this case data arrays would been placed into Fortran modules and there is only one allocation to be done for the entire run. This would remove a big fraction of the overhead.

(5) Increasing the workload given to a GPU improves performance.

(6) As a comparison, for CLOUDSC, the total run time spent on a single GPU is usually more than the time spent on a CPU with 12 cores. However, if the total run time is separated into GPU calculation time and overhead time the GPU calculation time is about half of the CPU total run time. This means that in the adapted CLOUDSC most of the time is spent on the data movement between CPU and GPU.

(7) Compared with double precision, the single precision reduced the total run time by approximately 40% if the work load was saturated.

A question that arises from the above analysis is how big a GPU cluster running CLOUDSC should be to deliver an operational forecast within an hour as normally required by ECMWF operations. The cloud scheme consumes approximately over 7% of the total time of an operational forecast in the current CRAY XC40 supercomputer, hence, the total time spent on this routine should be approximately 250s. In Tco1279 horizontal resolution there are $2 \times \sum_{n=1}^{(1279+1)}(4 \times n + 16)$=6599680 grid columns and the model performs 1920 time steps so which gives the rate of 130 ms per step on CLOUDSC. If a system such as LXG was to be used at peak performance, saturating each GPU's work load, then the NPROMA size and the number of grid columns assigned on each GPU should be equal to 80,000. The estimated number of GPUs to perform the computation would be about $\lceil 6599680/80000 \rceil$=83 which corresponds to $\lceil 83/4 \rceil$=21 GPU nodes (an estimate for higher resolutions is given in Table 10). In this case the calculation time for a CLOUDSC call would be approximately 500 ms but the total time is much higher given that includes overheads. Unfortunately, this setup would be too slow to deliver the forecast in the required 1 hour time. When the number of columns is reduced to 10,000 then the desired rate of 130 ms per step can be achieved for the calculation time (without including overheads). If the Meteo Swiss approach was followed, in which the entire model was adapted on GPUs, the overhead costs would be very small as data transfer would be required only during an I/O step. With 0 overhead, it would have been possible to complete the forecast within 1 hour with approximately 165 GPU nodes which is lower than the number of CPUs used operationally (360). But if GPUs are only used as an accelerator for CLOUDSC then, because of large data transfer overhead costs, the number of GPU nodes needed for a timely delivery of a forecast would likely exceed the number of CPU nodes currently used.

Table 10. *Estimated minimum number of GPUs required to run GPU CLOUDSC at peak performance at different resolutions based on LXG cluster.*

Resolution of IFS	Grid columns	Estimated number of GPUs	Estimated number of LXG computing nodes
$T_{CO}1279$	6599680	83	21
$T_{CO}1999$	16072000	201	51
$T_{CO}3999$	64144000	802	201

Acknowledgements

This work is supported by the bilateral cooperation agreement between ECMWF and the China Meteorological Administration (CMA) and the decision support project of responding to climate change. We would also like to thank Valentin Clément from ETH Zurich for his help and guidance in using the CLAW package.

References

Saarinen S., D. Salmond, R. Forbes (2014): *Preparation of IFS physics for future architectures*. The 16th ECMWF Workshop on High Performance Computing in Meteorology, Reading. UK, October 2014.

Saarinen S. (2015): *Using OpenACC in IFS Physics' Cloud Scheme (CLOUDSC)*.

Forbes R., A. M. Tompkins, A. Untch (2011): *A new prognostic bulk microphysics scheme for the IFS.* ECMWF Tech. Memo. No. 649.

Forbes R., M. Ahlgrimm (2012): *Representing cloud and precipitation in the ECMWF global model*. ECMWF Workshop on Parametrization of Clouds and Precipitation. Reading, UK. November 2012.

ECMWF, R-D. (2017): *IFS Documentation. Cy43r1 Operational implementation PART IV: Physical processes*.
https://www.ecmwf.int/sites/default/files/elibrary/2016/17117-part-iv-physical-processes.pdf.

Norman M., J. Larkin, A. Vose, K. Evans. (2015): *A case study of CUDA Fortran and OpenACC for an atmospheric climate kernel*. Journal of Computational Science. 9, 1-6.

Clement V. (2017): *CLAW Fortran Compiler Documentation*. https://github.com/C2SM-RCM/claw-compiler.

Williams S., A. Waterman, D. Patterson. (2009): *Roofline: an insightful visual performance model for multicore architectures*. Communications of the ACM, Vol.52 DOI: 10.1145/1498765.1498785.

Appendix A: Performance sensitivity with respect to NPROMA size.

The following results are obtained from CLOUDSC computation with double precision.

1) CPU

Memory binding is used to make sure that one CPU core corresponds to one OpenMP thread. In brief, the total time and sustained performance obtained only from all the physical cores with different number of NPROMA-blocks used are shown in table 11.

Table 11. *Total time, Gflops/s of original CLOUDSC computation on an LXG cluster node*

OpenMP threads	NPROMA	Time(ms) 12/24	Gflops/s 12/24
12/24	2	4857/3383	4.11/5.90
	4	3337/1978	5.98/10.10
	8	2450/1447	8.15/13.82
	10	2414/1351	8.28/14.78
	12	2242/1286	8.91/15.53
	16	2228/1321	8.96/15.11
	24	2288/1371	8.78/14.57
	32	2147/1358	9.30/14.72
	48	2106/1433	9.48/13.94
	64	2228/1379	8.97/14.48
	100	2157/1553	9.26/12.85
	128	2184/1508	9.16/13.26

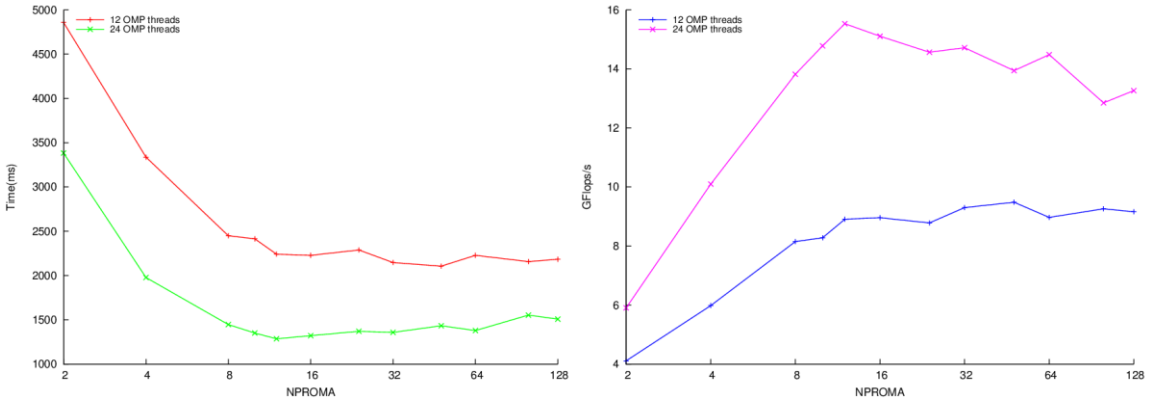

Figure 7. *Time and sustained performance of CLOUDSC computation for 160,000 grid column on 12 and 24 OpenMP threads with different number of NPROMA*

2) GPU

The mode that one OpenMP thread controls the computation of one GPU is used. Time and sustained performance with variable number of GPUs and NPROMA-blocks are shown in Figure 8 and Table 12.

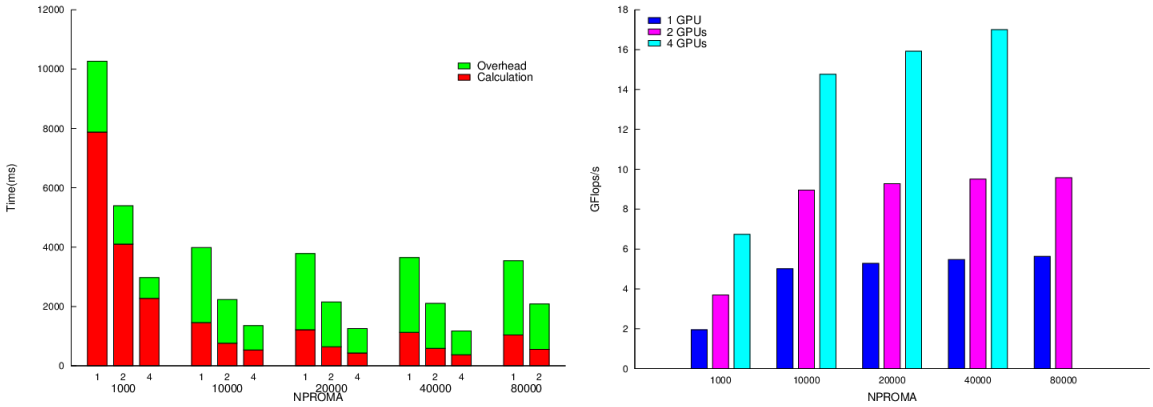

Figure 8. *Time and sustained performance of GPU accelerated CLOUDSC computation for 160,000 grid column on a GPU node with different number of GPUs and NPROMA-blocks*

Table 12. *Time and Gflops/s of the GPU accelerated CLOUDSC computation with different GPU number, OpenMP threads, NPROMA size settings.*

GPUs	OpenMP threads	NPROMA	Time(ms)			Gflops/s
			Calculation	Overhead	Total	
1	1	100	69271	3710	72982	0.27
		1000	7878	2381	10261	1.95
		10000	1454	2530	3986	5.01
		20000	1218	2566	3785	5.28
		40000	1123	2525	3650	5.47
		80000	1037	2507	3545	5.63
2	2	100	35017	1936	36940	0.54
		1000	4098	1296	5392	3.70
		10000	760	1473	2231	8.95
		20000	640	1514	2153	9.27
		40000	589	1514	2100	9.51
		80000	555	1529	2086	9.57
4	4	100	17754	1010	18754	1.06
		1000	2273	700	2963	6.74
		10000	536	817	1352	14.77
		20000	432	822	1254	15.92
		40000	376	799	1174	17.00

GPU Performance relative with the number of OpenMP threads

Time-sharing GPU across multiple OpenMP threads is used to enable concurrently executed kernels through OpenMP. For GPU computation of CLOUDSC on NVIDIA K80 GPU, the product of the number of threads and NPROMA-blocks must not be greater than the maximum NPROMA of 80000

for each logical GPU due to the limit of GPU memory. The time and sustained performance are shown in Table 13.

Table 13. *Time, Gflops/s of GPU accelerated CLOUDSC computation with an optimum combination of settings of variable GPUs, OpenMP threads, and NPROMA*

GPUs	OpenMP threads	NPROMA	Time(ms)			Gflops/s
			Calculation	Overhead	Total	
1	1	80000	1037	2507	3545	5.63
	2	40000	1631	2092	3485	5.73
	4	20000	1805	1981	3535	5.65
	8	10000	2676	1589	3674	5.43
2	2	80000	555	1529	2086	9.57
	4	40000	955	1242	2104	9.49
	8	20000	995	1354	2182	9.15
	16	10000	1470	1368	2420	8.25
4	4	40000	376	799	1174	17.00
	8	20000	615	869	1314	15.30
	16	10000	903	903	1519	13.14

Sometimes the total time in GPU accelerated CLOUDSC may be reduced a little by GPU time-sharing. In most cases time-sharing could reduce the overhead time, but increase the actual GPU calculation time. If taking into account the total time, time-sharing GPU seems to make little difference with that not using the time-sharing method.

Appendix B: CLAW Compiler.

CLAW is a high level source to source compiler based on OMNI compiler (Clement, 2017). CLAW uses its own Fortran directive language for adapting weather and climate models to different computer architectures. For example, it automatically re-organizes a scientific code performing loop extraction and fusion, loop re-ordering, loop hoisting and generating appropriate OpenACC or OpenMP directives. Due to its continuous development and improvement it is becoming an effective tool to maintain a unified source code for different computer architecture such as multicore, many-core, and GPU platform.

In order to illustrate the functions of CLAW a code segment from CLOUDSC is taken as an example, the original and transformed code are in Listing 2 and Listing 3 respectively, which includes directives

of OpenACC directives, loop-hoist, loop-interchange, and reshape. By executing the command "*clawfc –o scratch_trans.f90 scratch.f90 –d=openacc –t=gpu*", the code generated after transformation is shown in Listing 3.

```
!$claw ACC KERNELS
!$claw acc loop
!$claw loop hoist(k) interchange reshape(ZLFINALSUM(0))
DO JM=1,NCLV
  IF (.NOT.LLFALL(JM).AND.IPHASE(JM)>0) THEN
!$claw acc loop
    DO JL=KIDIA,KFDIA
      ZLFINAL=MAX(0.0_JPRB,ZLCUST(JL,JM)-ZDQS(JL)) !lim to zero
      ! no supersaturation allowed incloud ---V
      ZEVAP=MIN((ZLCUST(JL,JM)-ZLFINAL),ZEVAPLIMMIX(JL))
!       ZEVAP=0.0_JPRB
     ZLFINAL=ZLCUST(JL,JM)-ZEVAP
      ZLFINALSUM(JL)=ZLFINALSUM(JL)+ZLFINAL ! sum
      ZSOLQA(JL,JM,JM)      = ZSOLQA(JL,JM,JM)+ZLCUST(JL,JM) ! whole sum
      ZSOLQA(JL,NCLDQV,JM) = ZSOLQA(JL,NCLDQV,JM)+ZEVAP
      ZSOLQA(JL,JM,NCLDQV) = ZSOLQA(JL,JM,NCLDQV)-ZEVAP
      ! Store cloud liquid diagnostic if required
      IF (LLCLDBUDL.AND.JM == NCLDQL) ZBUDL(JL,4)=ZLCUST(JL,JM)*ZQTMST
      IF (LLCLDBUDI.AND.JM == NCLDQI) ZBUDI(JL,4)=ZLCUST(JL,JM)*ZQTMST
      IF (LLCLDBUDL.AND.JM == NCLDQL) ZBUDL(JL,5)=-ZEVAP*ZQTMST
      IF (LLCLDBUDI.AND.JM == NCLDQI) ZBUDI(JL,5)=-ZEVAP*ZQTMST
    ENDDO
  ENDIF
ENDDO
!$claw ACC END KERNELS
```

Listing 2. *Original codes with CLAW directives (scratch.f90)*

```
!$ACC KERNELS
!$acc loop
DO JL=KIDIA,KFDIA
  ZLFINALSUM = 0.0_JPRB
!$acc loop
  DO JM=1,NCLV
    IF (.NOT.LLFALL(JM).AND.IPHASE(JM)>0) THEN
      ZLFINAL=MAX(0.0_JPRB,ZLCUST(JL,JM)-ZDQS(JL)) !lim to zero
      ! no supersaturation allowed incloud ---V
      ZEVAP=MIN((ZLCUST(JL,JM)-ZLFINAL),ZEVAPLIMMIX(JL))
!       ZEVAP=0.0_JPRB
      ZLFINAL=ZLCUST(JL,JM)-ZEVAP
      ZLFINALSUM = ZLFINALSUM + ZLFINAL ! sum
      ZSOLQA(JL,JM,JM)      = ZSOLQA(JL,JM,JM)+ZLCUST(JL,JM) ! whole sum
```

```
      ZSOLQA(JL,NCLDQV,JM) = ZSOLQA(JL,NCLDQV,JM)+ZEVAP
      ZSOLQA(JL,JM,NCLDQV) = ZSOLQA(JL,JM,NCLDQV)-ZEVAP
      ! Store cloud liquid diagnostic if required
      IF (LLCLDBUDL.AND.JM == NCLDQL) ZBUDL(JL,4)=ZLCUST(JL,JM)*ZQTMST
      IF (LLCLDBUDI.AND.JM == NCLDQI) ZBUDI(JL,4)=ZLCUST(JL,JM)*ZQTMST
      IF (LLCLDBUDL.AND.JM == NCLDQL) ZBUDL(JL,5)=-ZEVAP*ZQTMST
      IF (LLCLDBUDI.AND.JM == NCLDQI) ZBUDI(JL,5)=-ZEVAP*ZQTMST
    ENDIF
  ENDDO
ENDDO
!$ACC END KERNELS
```

Listing 3. *The transformed codes (scratch_trans.f90)*

Appendix C: Roofline Model.

The roofline model is a useful approach for finding out how close the program is compared to sustainable peak performance on a particular GPU (Williams, 2009). The roofline model defines the peak performance of an architecture by looking at the memory bandwidth (for memory-bound kernels) and on the theoretical peak Glops/s (for compute-bound kernels). The operational intensity [Flops/Byte] is given by the algorithm and thereby defines the performance limit.

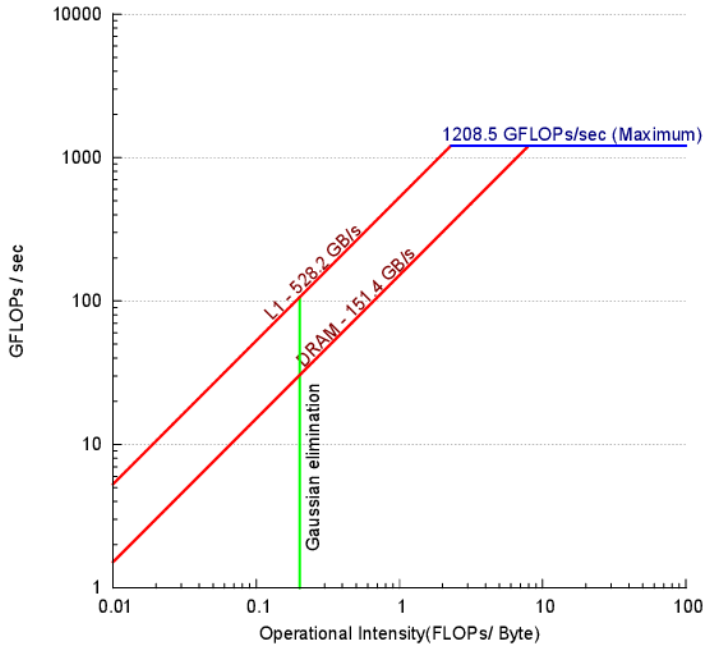

Figure 8. *Performance roofline for a half NVIDIA K80 (GK210) node for Gaussian elimination.*

The performance roofline for a half NVIDIA K80 (GK210) is shown in Figure 8. The kernel's operational intensity here is determined by approximating the corresponding values by using the NVIDIA Profiler. The kernel of Gaussian elimination, the most time-consuming port of CLOUDSC, is taken as an example.

Table 14. *Measurement of Gaussian elimination kernel through NVDIA Profiler (Execution of CLOUDSC is based on grid columns of 160,000, NPROMA of 80,000 with 1 GPU and 1 OpenMP thread)*

Kernel	FLOPS of double precision	DRAM read transactions	DRAM write transactions	Duration (ms)
Gaussian Elimination	31600000	2205001	2692383	1.314

The number of bytes in the formula of operational intensity should multiply the number of transaction to/from the device memory by 32 since each transaction takes place in 32 Byte chunks.

$$OI = FLOP/Byte = 31600000\ FLOPs / ((2205001 + 2692383) \times 32Byte) = 0.2016\ FLOPs/Byte$$

where, OI is the operational intensity.

The above value for OI implies that Gaussian elimination kernel is memory bound. The maximum theoretical performance $Perf_m$ is defined by the following formula and computed:

$$Perf_m = \min (OI \times PSMB, PDPFPP) = \min(0.2016\ [FLOPs/Byte] \times 151.3\ [GB/s], 1200.4\ [GFLOPs/s])$$

$$= 30.5021\ GFLOPs/s$$

where, PSMB is the "peak sustained memory bandwidth", PDPFPP is the "peak double precision floating point performance",

The reached performance for Gaussian elimination kernel ($Perf_r$) is computed as follows:

$$Perf_r = FLOP/time = 31600000\ FLOPs/1.314\ ms = 24.0487\ GFLOPs/s$$

The efficiency (E) is the ratio of reached performance over maximum performance which is a metric of the utilization of the resources.

$$E = Perf_r/Perf_m = 24.0487/30.5021 = 78.8\%$$