Science & Technology Facilities Council

Technical Report
DL-TR-2012-001

Final report for the gNEMO project: Porting the oceanographic model NEMO to run on many-core devices

AR Porter, SM Pickles, M Ashworth

March 2012

Chadwick Library
Science and Technology Facilities Council
Daresbury Laboratory
Daresbury Science and Innovation Campus
Warrington
WA4 4AD

Tel: +44(0)1925 603397
Fax: +44(0)1925 603779
email: librarydl@stfc.ac.uk

Science and Technology Facilities Council reports are available online at: http://epubs.stfc.ac.uk

ISSN 1362-0207

Final Report for the gNEMO Project: Porting the Oceanographic Model NEMO to run on Many-Core Devices

A. R. Porter, S. M. Pickles and M. Ashworth,
STFC Daresbury Laboratory, UK

January 2012

Executive Summary

This report describes work done in the gNEMO project investigating whether general-purpose graphics processing units (GPUs) can be used to improve the performance of an ocean-modelling code, NEMO. The porting is performed using accelerator directives with the aim of minimising changes to the original Fortran source code. Although speed-up factors of between two and five were obtained for five out of the six subroutines tackled, this is only in comparison to a single CPU core. Once the routines were modified so as to make use of all of the cores on a CPU, the GPU speed-up was only ever a factor of two at most. In current systems, GPUs and CPUs are separate units with a relatively slow connection between them. The time spent transferring data over this connection can be considerable and must be minimised. The work required to do this resulted in complex code consisting of double the number of lines in the original — a far cry from minimising changes to the NEMO source. Finally, with the GPU-enabled routines merged back into NEMO, a speed-up of 25% was measured when a GPU was used to augment a single CPU core. We conclude that environmental codes such as NEMO that do not have a single bottleneck and require that data be frequently transferred between CPU and GPU are not well suited for making use of current GPU technology.

Contents

1 Introduction

Many-core devices are widely acknowledged as being a key building block of future supercomputers. In fact, four of the top ten machines in the June 2011 Top500 List of Supercomputers [7] are already using some sort of many-core accelerator. Three of those four machines are using NVIDIA Graphical Processing Units (GPUs) which are currently the most popular accelerator option for scientific computing. These devices consist of hundreds of lightweight cores, groups of which execute together in lock-step (Same Instruction Multiple Data [SIMD]) fashion. The theoretical peak performance of GPUs is very attractive; the NVIDIA Tesla M2090 for instance has 512 cores capable of 665 GFlops (double precision) and a memory bandwidth of 177 GBytes/s (with ECC switched off). However, the lightweight nature of the cores combined with their SIMD operation can make getting such performance for a real-world code difficult.

NEMO [5] is a widely-used European oceanographic model. It is designed to be highly-portable and is written in Fortran90 and parallelised using MPI with a regular domain decomposition in latitude/longitude. At about 20 years old, NEMO has already seen many computer architectures come and go and therefore retaining its portability is crucial. At the same time, the code is very much memory-bandwidth bound and therefore the high peak memory bandwidth offered by GPUs is an attractive prospect.

This report describes work done in porting parts of NEMO to make use of GPUs while attempting to retain the original Fortran90 code base. We discuss the details of

using accelerator directives, the performance achieved and the lessons learned along the way.

1.1 NEMO Configurations

For the most part we have used the ORCA2_LIM configuration; one of the standard NEMO configurations. It is a two-degree-resolution global ocean model with 31 vertical levels coupled with the Louvain-la-Neuve sea-ice model (LIM) [3, 1] in the Arctic and Antarctic regions. The low resolution gives a grid of $182 \times 149 \times 31$ which is very manageable – the whole model executes in about 1GB of RAM. This makes it straightforward to read and write arrays from disk to allow the answers produced by a kernel running on a GPU to be checked against those obtained from the full model running on a CPU.

In order to investigate the scaling performance of the kernels we have also used a grid of $1442 \times 1021 \times 46$ to represent the ORCA025 configuration ($\frac{1}{4}^\circ$ resolution) and a grid of $362 \times 292 \times 46$ to represent the ORCA1 configuration.

1.2 Machines Used

We have made use of the following machines in this project:

- *cseht* is a linux cluster run by the DiSCO group at STFC Daresbury Laboratory. In this work we made use of the Nehalem-based compute nodes and associated NVIDIA Tesla S1070 GPU servers (which contain four M1060 cards). Each compute node has two, four-core Nehalem (Intel Xeon E5540) chips clocked at 2.53GHz and is connected to four M1060 cards;

- *SiD* is an IBM iDataPlex system hosted at Daresbury. It has compute nodes consisting of two, six-core Intel Westmere X56 chips clocked at 2.67GHz and 24GB of memory (2GB/core). (The Westmere chip is the 32nm die shrink evolution of the Nehalem.) One of the compute nodes also includes two NVIDIA Fermi (M2050) GPUs;

- *HECToR* is the current national UK HPC service. When this work was performed it was at "Phase IIb" and was a Cray XE machine with two, twelve-core AMD Magny Cours chips (clocked at 2.1GHz) per compute node;

- *Power7* is an IBM Power 750 Express system run by the DiSCO group at Daresbury. Each of the four frames has 256GB of memory and four Power7 chips (32 processor cores in total) clocked at 3.55GHz. A processor core has 256kB of L2 cache and 4MB of L3 cache.

The Power7 machine was used for some benchmarking runs of the OpenMP versions of the ported routines and HECToR was used for access to the craypat profiling tool.

2 Choosing the Routines to Port

The choice of routines to port to GPU obviously depends on the profile of NEMO. An example for the ORCA2_LIM configuration is shown in Table 1. In common with many environmental-science codes, the profile is rather flat in that no one routine accounts for

a large percentage of the total run-time. We can exclude the *lib_mpp_mpp_lnk_3d* routine since it is a largely a wrapper for the organisation of halo swaps for 3D arrays. This means that the most significant routines are *lim_rhg* (rheology of the sea ice), *tra_ldf_iso* (lateral diffusion of tracers), *tra_adv_tvd* (tracer advection), *ldf_slp* (lateral diffusion; slopes of neutral surfaces) and *trazdf_imp* (vertical diffusion of tracers handled using an implicit scheme). It turns out that *traadv_tvd_nonosc* is actually a child of *traadv_tvd* so in porting the latter it too must be ported. Therefore, these are the routines that we have attempted to port to the GPU.

3 Assessment of the Porting Technologies

As listed in Table 2, there are currently several ways of writing code for execution on a GPU. Of these, three are specific to NVIDIA hardware and while NVIDIA currently has the edge in GPU hardware, both AMD and Intel are working to develop their offerings in this area. A second key issue is the fact that NEMO is written in Fortran while CUDA and OpenCL are C-based. The performance benefits of running on a GPU would have to be substantial to warrant the effort required to re-write tried, tested and trusted parts of NEMO in C.

As a consequence of these considerations we have considered only directives-based approaches in this work since, in principle, they allow the original Fortran code to remain unchanged and also have the potential to generate code for different GPUs. In fact, HMPP Workbench (a product from French company CAPS entreprise) can generate either CUDA or OpenCL code from the same source. Retaining a single code base is essential for a portable code like NEMO as to fork the code to support a single, specialist architecture would result in an unacceptable increase in the maintenance overhead. It is also likely that such a branch will fall into disrepair and disuse as the main trunk of the code continues to be developed.

Programming the GPU with high-level directives can potentially introduce a performance penalty compared to the flexibility available when using a lower level such as CUDA. This cost must be balanced against the portability of the resulting code. A comparison of the performance of CUDA and PGI Directives implementations of a routine from the Weather Research & Forecast model has been performed by Wolfe and Toepfer [9]. They found that there was no performance penalty associated with using directives in that case. Obviously this can't always be guaranteed but it seems likely that good performance can be obtained from directives provided they are used with an understanding of the underlying hardware.

3.1 Directives-based Approaches

As with any well-structured scientific code, NEMO makes heavy use of Fortran MODULEs to encapsulate data and routines related to different parts of the model. This was found to cause difficulties for both the PGI and HMPP directives as they have limitations on the scope of data that the region of code to be run on the GPU can access.

An HMPP 'codelet' (subroutine/kernel to be run on the GPU) for instance cannot access arrays from a module - they must either be local or passed as an argument to the codelet. One solution to this is to use a 'region' rather than a codelet. In this approach one inserts directives to delimit the section of code that is to be run on the GPU. So long as the necessary modules are in scope within the program unit containing the region then

% of total run-time	Imbalance %	Routine
74.4	-	USER
13.4	2.6	limrhg_lim_rhg
5.2	7.8	lib_mpp_mpp_lnk_3d
5.1	2.6	traldf_iso_tra_ldf_iso
4.3	3.9	traadv_tvd_tra_adv_tvd
4.2	3.9	ldfslp_ldf_slp
3.3	7.5	traadv_tvd_nonosc
3.3	2.1	trazdf_imp_tra_zdf_imp
3.2	4.6	zdftke_tke_tke
2.6	2.8	dynzdf_imp_dyn_zdf_imp
2.2	5.0	zdftke_tke_avn
2.1	7.9	field_bufferize_bufferize_field
1.9	6.8	mathelp_moycum
1.6	34.4	solpcg_sol_pcg
1.5	7.5	field_bufferize_init_field_bufferize
1.2	6.5	traadv_eiv_tra_adv_eiv
1.2	6.3	eosbn2_eos_bn2
1.1	7.7	eosbn2_eos_insitu_pot
1.1	8.3	traswp_tra_unswap
1.1	12.7	dynspg_flt_dyn_spg_flt
1.0	6.7	zdfddm_zdf_ddm
1.0	7.8	eosbn2_eos_insitu
15.2	-	MPI
6.1	23.4	mpi_allreduce
5.9	39.7	mpi_recv
2.5	60.5	mpi_allgather
10.4	-	ETC
3.5	7.5	_wordcopy_fwd_aligned
1.4	23.8	__c_mcopy8
1.0	28.8	__c_mzero8

Table 1: A profile of NEMO running the ORCA2_LIM configuration on 12 MPI processes on HECToR Phase IIb.

Approach	Notes	Fortran support
PGI Accelerator Directives	Currently NVIDIA specific	Yes
HMPP Workbench	Can generate CUDA and OpenCL code	Yes
PGI CUDA Fortran	NVIDIA specific	Yes
OpenCL	Portable, open standard	No
CUDA C	Widely used but NVIDIA specific	No

Table 2: The available options for programming a GPU.

they can be used within the region. We did however find that using a region rather than a codelet has some drawbacks. First, extra data transfers to the device are generated for integers holding the dimensions of the arrays, and a lot of these are not actually used in the code (as evidenced by the Fortran compiler warnings). Second, there are some features (notably asynchronous data transfer) available to HMPP codelets that are not available to regions.

An alternative solution that retains the use of a codelet is to write a new subroutine to take the place of the region. All of the module arrays that it uses can then be passed to it as arguments. In fact, HMPP can help with this process if a version that works with a region has been constructed. Adding "-d -k" to the command line options passed to HMPP causes it to generate a file with an "extracted.{c,f90}" extension. This file contains the codelet that has been automatically generated out of the region. This code can then be used to help with converting the code to the codelet approach.

A key issue with making use of current many-core devices is the transfer of data between the memory of the CPU and the memory of the device. With the potential future integration of many-core devices and CPUs onto the same die this issue should be resolved but currently it is easy to lose all compute speed-up because of the cost of getting data to and from the device. It is therefore essential to be able to overlap computation and data transfer in order to hide the cost of the latter. However, as Figure 1 shows, hiding the cost of data transfers comes at a price. For routines that do not involve any halo swaps (*traldf_iso* and *trazdf_imp*), the overhead of using directives to port them to GPU in terms of lines of code is approximately 30%. This increases dramatically to 100–150% when a routine contains halo swaps due to the associated data transfers and the need to break the routine into more codelets (because a codelet cannot contain a halo swap), each with a fairly lengthy interface description in both Fortran and HMPP directives.

At the time we did this work, the PGI Accelerator implementation had no support for asynchronous data transfers although it does have 'data regions' for keeping data on the device between calls to accelerated regions. It also allows for subroutine calls from within such regions. The use of regions rather than separate codelets would also significantly reduce the extra code required when halo swaps are present. HMPP does have support for asynchronous transfers and also, codelets can be grouped together with data retained on the device between calls to them.

Unless otherwise specified, we used version 11.3 of the PGI compiler, version 2.4.4 of CAPS HMPP Workbench and version 11.1 of the Intel compiler.

4 Comparing GPU and CPU Performance

One must be careful in comparing the performance obtained by an application on a GPU with that on a CPU. Inevitably an application must be optimised to get good performance from a GPU. Unless equal effort is put into optimising it for the CPU then the GPU has an unfair advantage. Just as one optimises code to make use of the many cores of a GPU, an application must also be optimised to make use of the multiple cores that all modern CPUs have. (Obviously there is little point in running the ported kernel on a single core of a GPU so why compare the performance of a parallel application on a GPU with that of a serial one on the CPU?)

Modifying an already parallel (MPI) program like NEMO to make use of GPUs makes

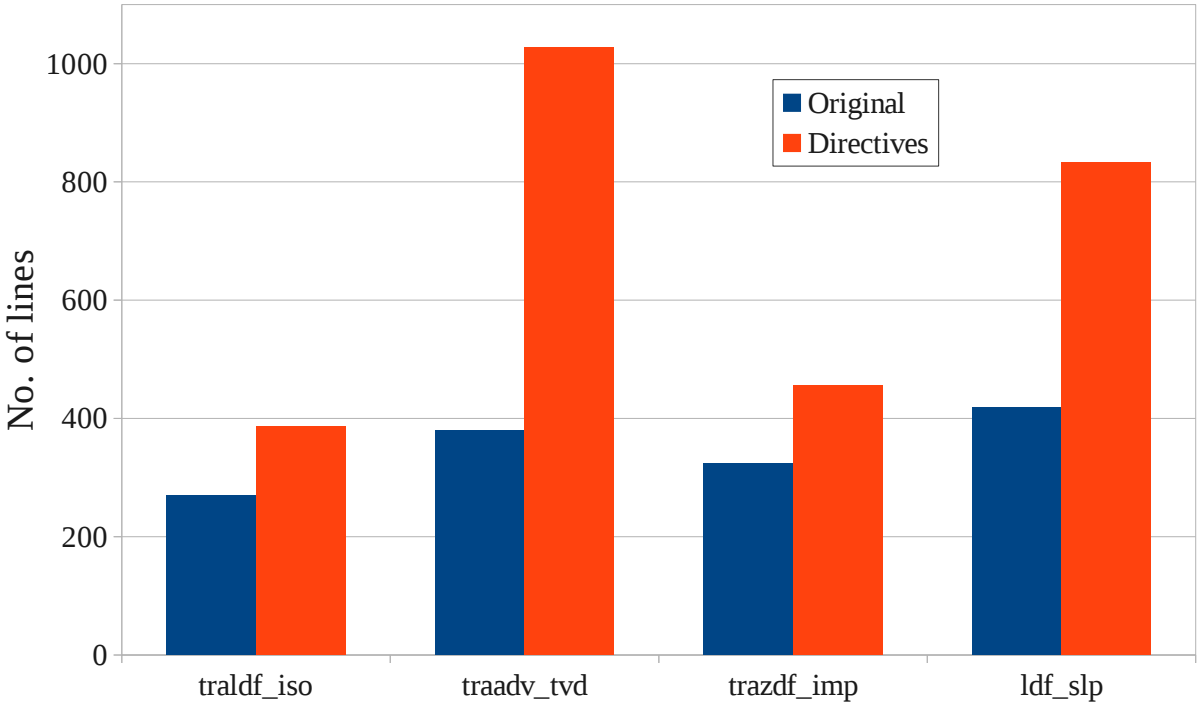

Figure 1: Plot of the number of lines of code for each routine before and after porting to GPU with HMPP directives. Note that both the *traadv_tvd* and *ldf_slp* routines involve halo-swaps while the other two do not.

the situation more complex. Normally, the number of GPUs available on a compute node is significantly fewer than the number of CPU cores on the node. This creates a problem for any parallel program wishing to make use of GPUs since only a subset of the program's processes will have access to a GPU if it uses all the cores of a compute node. (Once a GPU has been bound to a process it cannot be used by another process until it is released.)

There are two common solutions to this problem. The first is to under-populate the host CPUs so that there are only as many processes on a compute node as it has GPUs. However, this means that the performance gain obtained by using the GPU must be weighed against the lost performance of the unused CPU cores. For instance, cseht has four Tesla cards per compute node but each compute node consists of two Nehalem sockets and thus a total of eight CPU cores. Therefore, if we wish to keep a 1:1 mapping between CPU processes and GPUs then we must occupy only half of the available CPU cores. The second solution is more complex and involves attempting to use OpenMP (for example) to employ the CPU cores that do not have a GPU available in useful work.

5 Lateral diffusion of tracers (*traldf_iso*)

Before attempting to optimise the routine for the GPU, we measured its performance on a single core of an Intel Nehalem chip. Compiled with the Intel compiler with flags "-O3 -axAVX" and run on 1 Nehalem core the mean time/kernel call over 100 calls was 0.095 seconds. Following all of the optimisations done for the GPU, this time was reduced to 0.082 seconds.

Optimisation notes	Number of calls	Mean time per call (s)
First working *traldf_iso* on Tesla GPU	2	19.539
Made $zdk1t$ and $zdkt$ private so loop at 216 can be parallelised	2	1.087
Made $zdkt$ and $zdk1t$ depend on jk so that loops over jk can be parallelised. Moved timing loop down into kernel	100	0.038
Build with optimising flags on host	100	0.039
Put in explicit loops instead of (:,:) notation as allows loops to be fused	100	0.037
Removed jn index from $zd\{i,j\}t$ workspaces	100	0.033
Moved outer loop over tracers inside REGION but run on host with DO HOST	100	0.034
Added fastmath option to -ta=nvidia	100	0.033
Took tracer loop down inside all loops and manually unrolled it.	100	0.024
Added time option to -ta=nvidia	100	0.025
Removed erroneous DO HOST directive on first loop	100	0.024
Tweaked schedule (longer vector on innermost loop) on most expensive loop to improve performance	100	0.024
Permute loops from jk, jj, ji to jj, ji, jk to improve 'memory coalescing'.	100	0.021
Same code on Fermi GPU	100	0.018
Optimised code on single Nehalem core	100	0.082

Table 3: The key stages in optimising the *tra_ldf_iso* routine to run on the Tesla GPU using PGI accelerator directives. For comparison, the bottom row gives the performance of the final code when built with the Intel compiler and run on a single core of a Nehalem chip on cseht.

5.1 Porting *traldf_iso* with PGI directives

The *traldf_iso* routine presents a good starting point for porting NEMO to make use of GPUs due to its relative simplicity and complete lack of MPI calls. This enabled the entire body of the routine to be encapsulated in a single accelerated region using the PGI accelerator directives. The basic optimisation steps and timings of the resulting code are given in Table 3.

The current generation of many-core devices must be accessed over a PCI-Express bus which means that data transfer to and from the device can be a significant overhead. This must be borne in mind when timing the execution of code on a GPU in order to separate this cost from that of the computation. Since the whole routine being accelerated could be encapsulated in a single, accelerated region, it was possible to add an extra loop inside this region to repeat the kernel multiple times without requiring any additional data transfers.

One advantage of using the PGI Accelerator directives is that basic profile information

```
./traldf_iso_harness.F90
  kernel
    172: region entered 101 times
        time(us): total=2416730 init=21 region=2416709
                  kernels=2303948 data=43597
        w/o init: total=2416709 max=33362 min=23641 avg=23927
        194: kernel launched 101 times
            grid: [19x15]  block: [16x8x2]
            time(us): total=180467 max=1798 min=1780 avg=1786
        205: kernel launched 101 times
            grid: [12x10]  block: [16x16]
            time(us): total=7357 max=74 min=71 avg=72
        226: kernel launched 101 times
            grid: [75x4]  block: [16x2x8]
            time(us): total=139061 max=1380 min=1374 avg=1376
        257: kernel launched 101 times
            grid: [37x8]  block: [16x4x4]
            time(us): total=896716 max=8903 min=8842 avg=8878
        294: kernel launched 101 times
            grid: [37x8]  block: [16x4x4]
            time(us): total=298665 max=2962 min=2954 avg=2957
        373: kernel launched 101 times
            grid: [10x32]  block: [16x16]
            time(us): total=557276 max=5549 min=5490 avg=5517
        402: kernel launched 101 times
            grid: [74x4]  block: [16x2x8]
            time(us): total=208920 max=2072 min=2060 avg=2068
./GPU/traldf_iso_harness.F90
  kernel
    165: region entered 2 times
        time(us): total=6292127 init=3805949 region=2486178
                  data=56672
        w/o init: total=2486178 max=2425436 min=60742 avg=1243089
```

Figure 2: An example of the output of the built-in NVIDIA profiler for the *traldf_iso* kernel running on a Tesla GPU. Entries for some inexpensive kernels have been removed for clarity.

is readily obtained for the code running on the GPU, simply by adding the "time" option to the "-ta=nvidia" flag. An example of the output for *traldf_iso* is shown in Figure 2. From this we see that the region starting at line 172 in the source was executed 101 times (once for results verification, 100 further times for timing). The 'data' field tells us that out of a total time of 2.4 s, only 44 ms were lost to data transport for this region. Below this, timings are given for each of the 'kernels' (parallelised loops executed on the GPU) in the region. So we see for instance that the loop at line 205 only took 72 μs on average while that at line 257 took nearly 9 ms. It is therefore easy to identify which loops to attempt to optimise.

In order to check that the code was optimal, we made use of the PGI forum to ask for feedback from other users and developers. This led to the moving of the tracer loop (which only has a trip count of two – once for temperature and once for salinity) down inside all of the other loops and then to manually unrolling it.

5.2 Porting *traldf_iso* with HMPP

Once we could get no further speed-up from the PGI-directives version of the routine, we moved to using HMPP Workbench instead. As with the PGI directives, the most difficult task in porting the kernel was dealing with the scoping of the various arrays used in the computation; with the exception of integer parameters, all of the variables used in an accelerated region must be contained within the current program unit and cannot come from external modules.

We worked around this issue by enclosing the computational kernel (the body of a subroutine that USE'd several modules) within a 'region' pragma. The data usage patterns for the various arrays (*c.f.* INTENT(in) or INTENT(inout) in Fortran) are then specified as parameters to the region. The key steps in optimising the resulting kernel are listed in Table 4.

As with the PGI directives, the key step is, unsurprisingly, to ensure that the correct loops are being parallelised. The next largest improvement was gained by permuting loop indices from jk, jj, ji (*i.e.* levels, latitude, longitude) to jj, ji, jk. If left unpermuted, the nested loop is parallelised such that consecutive threads are working on array sections well separated in memory. Since threads on the GPU are divided up into groups which are then executed in SIMD (Same Instruction Multiple Data) fashion, best performance is obtained when a fetch from memory supplies data that can be used by all of the threads in a given group. If the threads aren't working on a contiguous section of memory then this will not happen. Permuting the loop indices ensures that parallelisation occurs over the indices in which an array is contiguous in memory and thus that neighbouring threads are working on contiguous parts of an array. Strangely, this seems to be more important with HMPP than it did with the PGI directives. The final result of 0.015 s per kernel call is some 20% faster than the time of 0.021 s achieved with the PGI directives.

A key advantage of HMPP at the time of writing is its support for asynchronous data transfer to the GPU. Although not strictly necessary for the *tra_ldf_iso* kernel, we experimented with this feature to overlap loading of data to the GPU with the reading of initial array values from disk. This revealed that in order to use asynchronous I/O with an array on an NVIDIA device, the array must be allocated in 'pinned' (page-locked) memory on the host. Further, the only way to achieve this is to use the CUDA routine *cudaMallocHost()*. Since this routine is in C, we used the F2003 standard mechanism for C/Fortran interoperability to generate a Fortran interface to it.

Optimisation notes	No. of calls	Mean time per call (s)
First working *traldf_iso* on GPU	10	32.238
Put *!$hmppcg parallel* for outer two loops of the most expensive triply-nested loop	10	16.920
Repeat above for *all* triply-nested loops	10	0.117
Move outer tracer loop inside and unroll	10	0.100
Put *io=in* condition on temporary arrays to prevent them being copied back to host	10	0.096
Simulate 3D gridification in 2D on most expensive loop	100	0.067
Permute indices jk, jj, ji to jj, ji, jk on second most expensive loop	100	0.053
Undo 3D gridification on most expensive loop and permute indices	100	0.022
Permute indices on all remaining loops	100	0.017
Removal of device allocation from within timing region	100	0.015
Optimised code on single Nehalem core	100	0.082

Table 4: The key stages in optimising *tra_ldf_iso* to run on the Tesla GPU using HMPP workbench. For comparison, the bottom row gives the performance of the final code when built with the Intel compiler and run on a single core of a Nehalem chip on cseht.

5.3 Porting *traldf_iso* with OpenMP

Finally, for a fair comparison of the GPU with the CPU we must create a version of the routine capable of using all of the cores on the CPU. The standard method for doing this is to use OpenMP to parallelise the various loops in the routine over the available number of threads/cores. In order to minimise the overhead of the creation and destruction of thread teams, the whole timing loop was enclosed within an OMP PARALLEL region. Within this, each computational loop was parallelised by simply specifying OMP DO. This means that all of the 3D loops were parallelised in the z/depth dimension. The few 2D loops, mainly dealing with the surface and ocean floor, were parallelised in the y dimension. The expensive loop at line 257 was also fused with the small loop immediately following it since both of the outer loops over z had identical limits.

In order to maintain good performance when running across more than one socket, the code had to be modified to ensure that memory is initialised by the thread that will access it, rather than just by the master thread - this ensures that it is allocated in close vicinity to the physical core on which it is executing. Care also must be taken in enforcing suitable affinity settings in the run-time environment. We set KMP_AFFINITY=none and used the *taskset* command on the linux-based systems and set PSC_OMP_AFFINITY=FALSE on HECToR. On the Westmere chip, the six- and four-thread jobs were fastest when the threads were shared evenly between the two sockets of a node. [1] On the older Nehalem chip, the same applied just to the four-thread job. We were unable to find any way of sharing threads evenly between sockets on the Power7 system.

[1] This demonstrates that four threads are sufficient to saturate the memory bandwidth to a single socket.

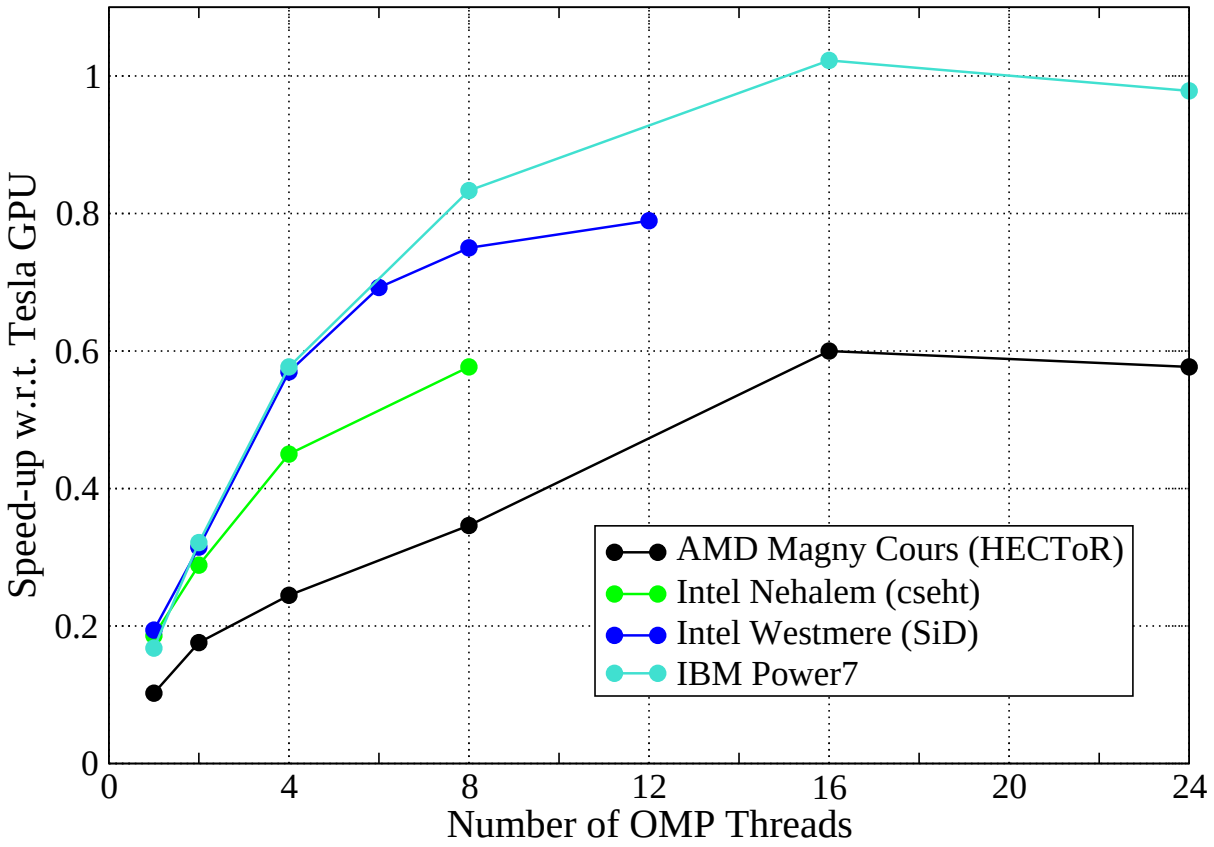

Figure 3: The speed-up of the OpenMP version of the *tra_ldf_iso* routine with respect to its performance on an NVIDIA Tesla GPU. In each case the number of cores utilized is the same as the number of OpenMP threads. Results are the averages of three runs for the ORCA2_LIM case and lines are guides to the eye.

Figure 3 shows the performance of this OpenMP version of the routine relative to the HMPP version running on the NVIDIA Tesla card. For a single thread/core, the Intel Nehalem and Westmere processors gave very similar performance and were, surprisingly, slightly quicker than the Power7. As the number of threads is increased, the Westmere initially matches the Power7 and both outperform the Nehalem, presumably due to their greater memory bandwidth which is key for the low computational-intensity loops at the heart of the routine. Despite the relatively good scaling on the Westmere, even using a full node of SiD (two, six-core Westmere chips) only gets us to 79% of the performance of the code on the Tesla GPU. Using a single socket (six cores) gets us 69%. In fact, only the Power7 system is able to match the performance of the GPU and it requires two sockets (16 cores) to do so. Note that the HECToR results could be improved upon by taking care to share threads evenly between sockets and/or dies (the Magny Cours chip is actually two, six core dies on a single socket) so as to make best use of available memory bandwidth.

6 The Rheology of the sea-ice component (*lim_rhg*)

In the standard ORCA2_LIM configuration, the sea-ice component (LIM2 or LIM3) couples to the ocean model only once in every ten time steps. Despite this, the ice rheology

Region	Call count	Nehalem		Tesla	
		Total (s)	Average (s)	Total (s)	Average (s)
Whole kernel	6	39.43	6.572	981.04	163.51
Alloc GPU	2	0.00	0.000	2.43	1.22
GPU store	3252	0.00	0.000	273.41	0.08
GPU load	2172	0.00	0.000	179.75	0.08
part1	6	0.22	0.037	4.43	0.74
part2	6	0.35	0.058	7.60	1.27
part3a	720	12.94	0.018	9.11	0.01
part3b	720	4.23	0.006	7.40	0.01
part3c	720	5.86	0.008	6.15	0.01
part3d_odd	360	2.63	0.007	129.92	0.36
part3d_even	360	2.73	0.008	130.74	0.36
part3e_even	360	2.65	0.007	133.89	0.37

Table 5: Comparison of the profile of the ported *lim_rhg* routine when run on a single Nehalem core and a Tesla GPU. Only codelet routines, data transfer and GPU initialisation costs are included. Timings are for the ORCA025 dataset.

routine accounts for approximately 13% of run-time when the model is run on 12 or 24 cores of Phase IIb of HECToR (Table 1). It therefore appears to be a prime target for optimisation/acceleration.

Unlike the tracer diffusion routine tackled in Section 5 however, the ice rheology routine is relatively long (776 lines), contains an iterative solver and performs many halo swaps. Building on the experience gained in porting the tracer-diffusion routine, only HMPP was used for this component (since it had performed slightly better than the PGI accelerator directives for *traldf_iso* and the latter do not support asynchronous data transfers at the present time).

As with *traldf_iso*, a completely serial test harness was constructed around the *lim_rhg* routine. However, the halo-swap calls were retained and always executed on the master thread running on the host CPU. This ensured that, for the harness to give correct results, the necessary data had to be available on the CPU prior to each halo-swap call. Again, following our findings when porting *traldf_iso*, each section of code suited to acceleration was moved into a distinct codelet subroutine.

All unnecessary data transfers to and from the GPU were eliminated by making the related variables 'resident' on the device. Required data transfers for these variables were explicitly managed via HMPP's advancedload/delegatedstore directives. Note that trying to declare Fortran allocatable arrays to be resident on the GPU revealed a bug in HMPP (version 2.4.4). For the purposes of the test harness therefore, these allocatable arrays were made static.

As with the majority of NEMO, the computational intensity of the loops in *lim_rhg* is actually rather low. In addition, the sea-ice model does not use an explicit discretisation of the thickness of the ice and as a result there is no z-dimension to the calculations. Hence all of the compute loops are only doubly nested.

The profile of the ported, optimised *lim_rhg* routine for an ORCA025–resolution test case is shown in Table 5 for both a single Nehalem core and a Tesla GPU. Clearly the average time taken per kernel call is much greater on the GPU (164 s) than it is on the

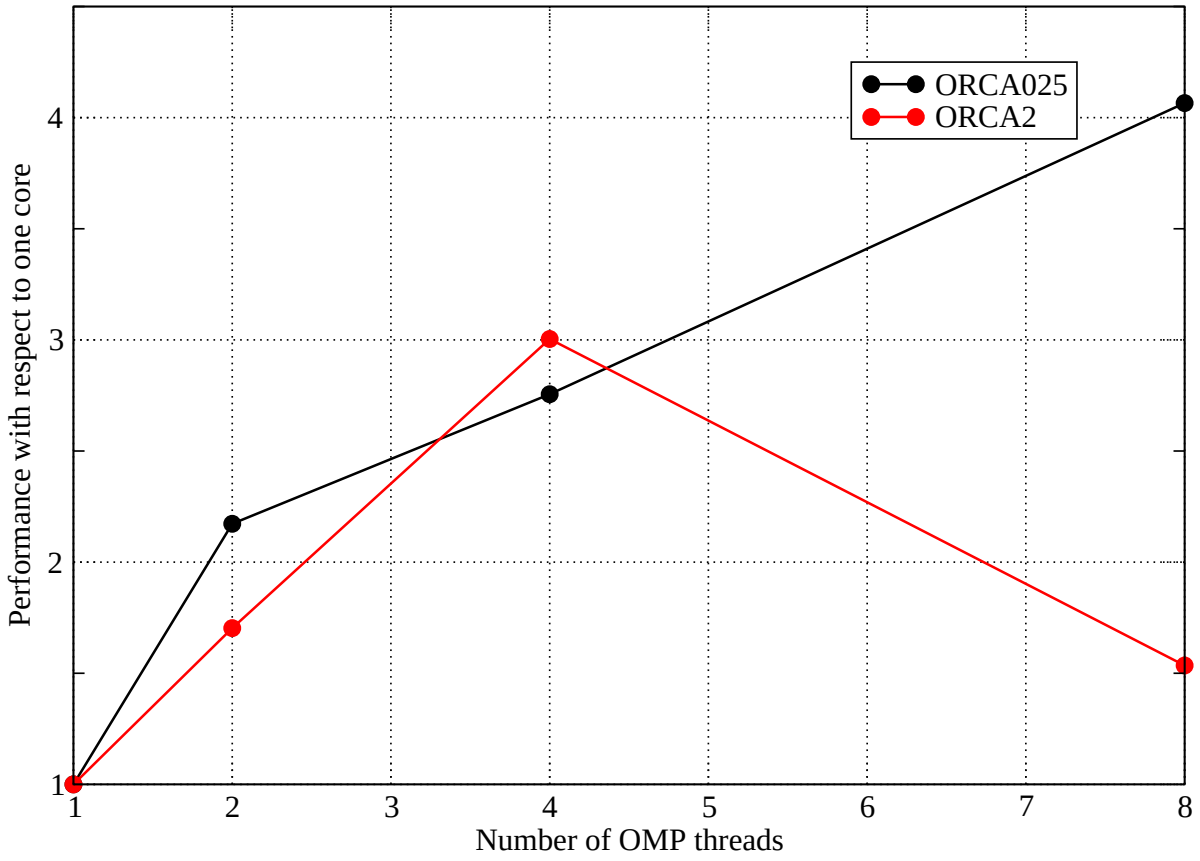

Figure 4: Scaling performance of the OpenMP version of the *lim_rhg* kernel on a Nehalem compute node for the ORCA2 and ORCA025 datasets.

Nehalem (7 s). However, this large difference is primarily due to data transport costs as can be seen by the entries for GPU store and GPU load (data downloaded from the GPU to CPU RAM and vice versa, respectively). The *part3d** and *part3e** kernels also include substantial data transfer costs because their codelet arguments include arrays that are transferred to/from the GPU upon every call. (They have not been optimised to the same extent as the other kernels in the table.) This emphasises the need to optimise data transport to/from the device in order to achieve good overall performance.

In this case however, the compute performance itself does not justify the effort required to optimise the data transport. Consider the performance of the *part3a–c* kernels which are particularly important due to their involvement in the iterative solver (note the high call counts). Only for *part3a* does the GPU out-perform the Nehalem core and then only by ∼30%; *part3b* is ∼75% slower on the GPU and *part3c* ∼5% slower. This is to be contrasted with the situation in *traldf_iso* where the kernel was a factor of four faster on the Tesla GPU and retained a factor of two speed-up, even when OpenMP was employed to use all four cores of a single Nehalem chip.

We can therefore conclude that given the low performance of the compute kernels and the frequency with which data must be transferred back to the CPU memory, this routine is not well suited to making good use of the Tesla GPU. To underline this conclusion, we consider the performance of this kernel when ported to use OpenMP. The plot in Figure 4 shows the scaling performance of the OpenMP version of the kernel on a single node (two Nehalem chips) of the cseht cluster. On a full Nehalem socket (four cores),

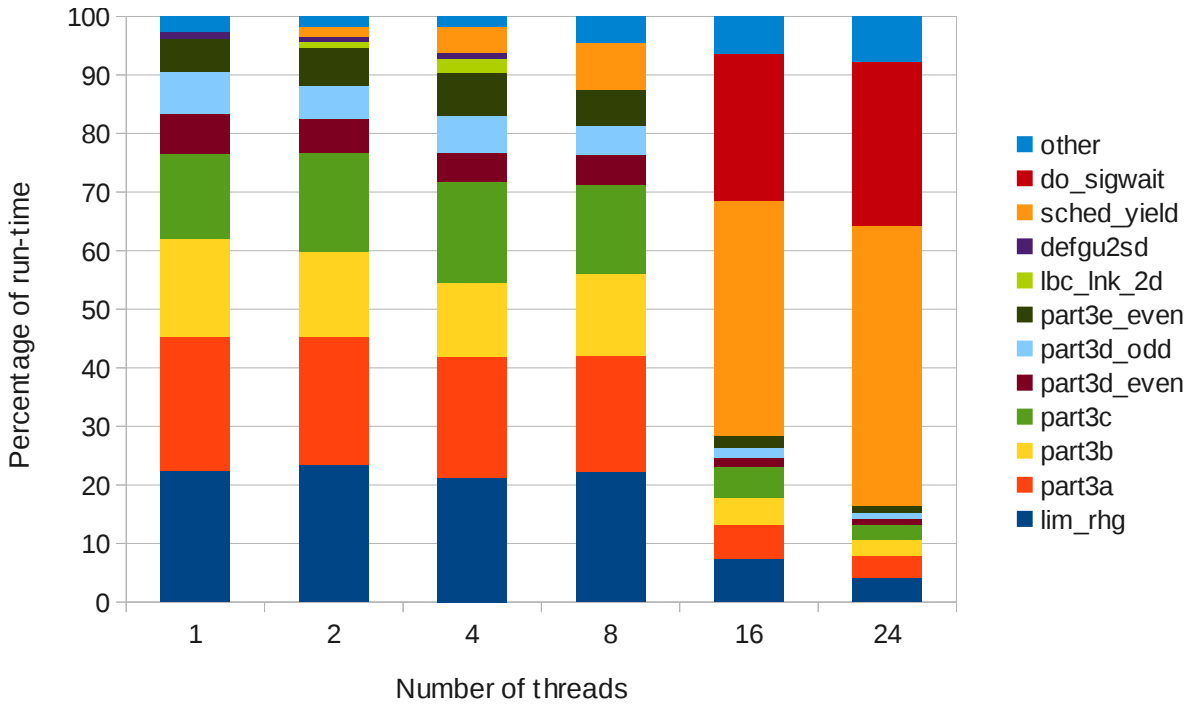

Figure 5: Profile of the OpenMP version of the *lim_rhg* kernel as the number of OpenMP threads is increased. Results are for the ORCA2 dataset run on HECToR IIb.

the OpenMP version achieves nearly a factor of three speed-up over the performance obtained on a single core for both the ORCA2 and ORCA025 datasets. The OpenMP version is therefore a significant improvement and emphasises the dominance of the CPU over the GPU for this kernel. That said, the scaling of the OpenMP implementation is poor, even for the relatively large ORCA025 dataset. Investigation of this aspect with profiling tools shows that it is the thread synchronisation required for the calls to the halo-swap routines that is the cause – see Figure 5.

7 Tracer Advection (*tra_adv_tvd*)

Back in Section 2 we concluded that the *tra_adv_tvd* routine was a potential candidate for porting to the GPU. Table 1 shows that this routine accounts for 4.3% of execution time for the ORCA2_LIM configuration when run on 12 MPI processes on HECToR IIb. The routine calls another subroutine, *nonosc*, but the two routines combined are only 374 lines in total. However, both *tra_adv_tvd* and *nonosc* contain several halo-swap calls and these present the major difficulty in porting these routines to the GPU.

As with the other routines, we first created a serial test harness for *tra_adv_tvd* which allows its results to be compared with those obtained from the original version within NEMO. The initial form of this harness with the original version of *tra_adv_tvd* demonstrated that it took 0.115 s/call on a single Westmere core and 0.124 s/call on a single Nehalem core (when compiled with the Intel compiler with "-O3 -axAVX").

We used HMPP Workbench to port the routine due to its support for asynchronous data movement. Since the calls to the halo-swap routines must be executed on the CPU, these naturally break the routine up into several sections, each of which was made into a separate codelet for execution on the GPU. The two calls to *nonosc* had to be inlined

Notes	Time per call (s)
First working port with kernel1 on GPU	0.401
Permute loops in kernel1	0.229
Move kernel2 to GPU	0.252
Permute loops in kernel2 and keep arrays on GPU between calls	0.214
Make kernel2 async. and overlap with halo swaps	0.194
Async. download for results of kernel2	0.177
In-line *nonosc* and convert into two codelets, nonosc1 and nonosc2	0.274
Permute loop indices in nonosc{1,2}	0.242
Make nonosc{1,2} work arrays local instead of args, overlap sending of zwx and zwy with halo swaps for zwy and zwz, respectively	0.205
Remove unnecessary data transport for nonosc{1,2}	0.196
Move kernel3 to GPU	0.181
Improve halo-swap performance by re-ordering indices on work arrays so that tracer index is slowest-varying	0.092
Move working-array initialisation into separate *kernel_init* so can overlap with data transfers which must happen upon every iteration of the timing loop (more realistic)	0.095
Re-ordered initial data loads and switched to have them sync. and *kernel_init* codelet async.	0.085
Original kernel on single Nehalem core	0.124

Table 6: Steps in the porting and optimisation of the *tra_adv_tvd* routine. Timings are on Nehalem and Tesla hardware for the ORCA2 dataset.

since code executing on a GPU cannot call subroutines within the HMPP model. In total the ported routine consists of six codelets for execution on the GPU. As usual, great care had to be taken to avoid unnecessary data transfers to/from the GPU. For this we made use of HMPP's ability to map an array from different codelets to the same piece of memory on the GPU and keep it there between calls. This achieves the same result as declaring an array to be device resident but is simpler to do in practice. We also succeeded in removing uploads/downloads of temporary arrays by declaring them as inputs to the codelet and then using the *noupdate* clause for them at the corresponding callsite.

The main steps in the porting and optimisation of the routine are listed in Table 6. After quite a lot of work, the final, ported version of the routine on a Tesla GPU is some 31% faster than the original version running on a Nehalem core when using the ORCA2_LIM dataset. Attempts to execute the code with the ORCA025 grid failed because of insufficient memory on the GPU and so we used the ORCA1 grid.

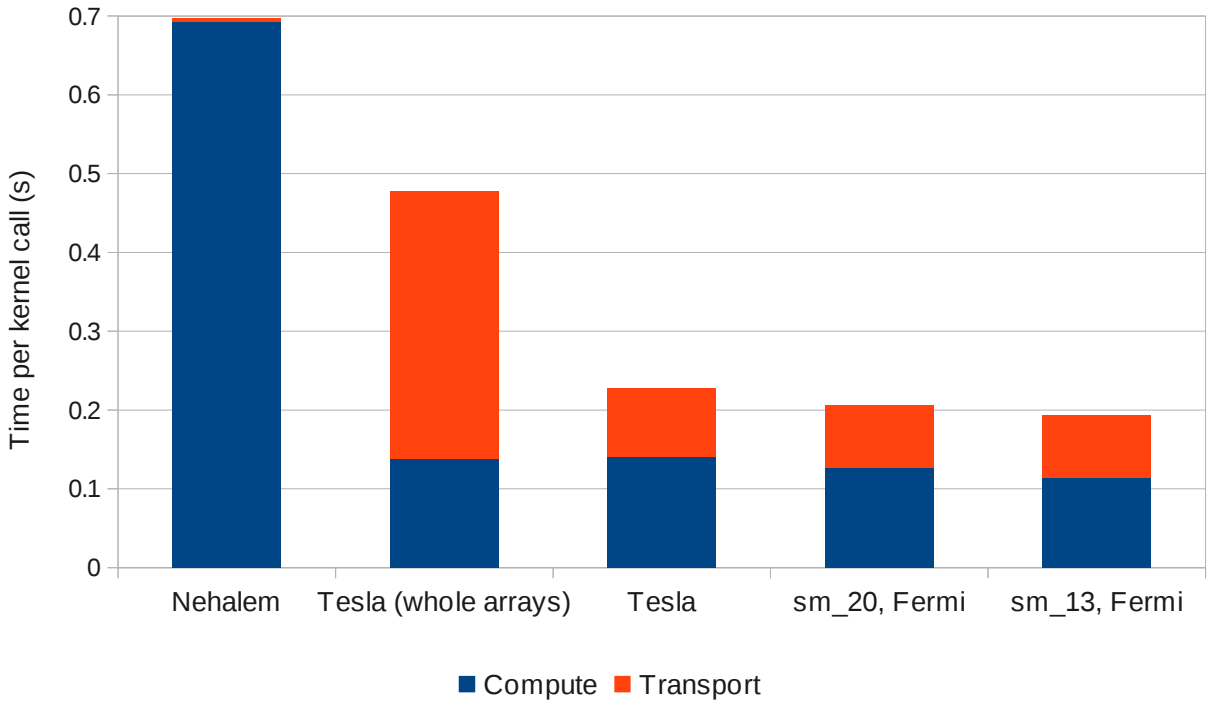

Figure 6: The time spent in compute and data transport in the *tra_adv_tvd* kernel and the ORCA1 grid when running on a single Nehalem (Westmere) core and a Tesla (Fermi) GPU. The "Tesla (whole arrays)" column shows the performance of the kernel before it was optimised to reduce data transport. The rightmost column shows the performance of the kernel on a Fermi GPU when compiled for a Tesla.

Figure 6 shows the breakdown of the kernel execution time in terms of compute and data transport (to and from the GPU). From a comparison of the first two columns, it is clear that data transport is the main performance bottleneck when the kernel is run on the GPU. However, the majority of the data transfers between the GPU and CPU are for the purposes of doing halo-swaps which obviously only involves the halo regions of each array. Therefore, we modified the code so that only the halo regions of an array are transferred between the GPU and CPU when doing a halo swap. Doing so reduced the time spent in transferring data from 0.34 s (per kernel call) to just 0.09 s when using the Tesla GPU on cseht (third column in Figure 6). Uploads (downloads) of halos to (from) the GPU were overlapped with the packing (unpacking) of halos on the CPU for any other arrays involved in a particular halo swap.

Finally, this version of the kernel was benchmarked on a Fermi GPU on SiD. As expected, the data transport cost remained similar at 0.08 s per call and the computational cost was slightly reduced from 0.14 s on the Tesla to 0.11 s on the Fermi. Strangely, this time was obtained when the NVIDIA CUDA compiler targeted the Tesla architecture ("sm_13"). If it targeted the Fermi ("sm_20") architecture then the computational cost of the resulting binary was 0.13 s per kernel call.

8 Vertical Tracer Diffusion (*tra_zdf_imp*)

As it is essential to minimise data transport to and from the GPU, it seemed sensible to continue porting the tracer-related routines that often make use of the same fields.

Table 1 of Section 2 shows that the *tra_zdf_imp* routine accounts for 3.3% of execution time for the ORCA2_LIM configuration when run on 12 MPI processes on HECToR IIb. This routine is only about 300 lines long and does not contain any halo-swap calls.

Following the same strategy used for the other ported routines, *tra_zdf_imp* was extracted from NEMO and integrated into a serial test harness which gave the same results as when the routine executed within NEMO. (For the first time it was found necessary to save the harness input data in binary form [Fortran unformatted] otherwise the loss of precision caused the harness to produce results that differed from the original by more than one part in 10^{10}). With the harness giving correct results when built with the Intel compiler and flags "-O3 -axAVX," it was used to time 50 calls of the routine on a single Nehalem core. The measured time per call was 0.039 s.

HMPP directives were again used to port this routine to the GPU. Like all of the other tracer-related routines dealt with so far, *tra_zdf_imp* contains an outer loop over tracers of trip-count two (temperature and salinity) which obviously has very limited scope for parallelisation. In this case however, this loop contains a large IF block that modifies the calculation depending on the tracer being dealt with. Therefore we unrolled this loop in contrast to previous routines where it was pushed down inside all of the other loops. This has the advantage of not increasing the memory-bandwidth requirements of the loops but does nearly double the number of distinct loops in the routine (to 19). Since each of these loops becomes a separate kernel that must be launched on the GPU this is not ideal. However, even in this form, once all of the loops had been given permute directives (k,j,i $\Rightarrow$ j,i,k) so as to improve memory coalescing, the performance on the Tesla GPU (including data transport costs) matched that of the Nehalem core.

At this point the code was analysed using the simple CUDA profiler (export CUDA_PROFILE=1; run application; timings written to cuda_profile_0.log). Using this, the more expensive loops were optimised. For some, precious memory bandwidth could be saved by storing intermediate results calculated for the first tracer for use in the same loop with the second tracer. Following the work of Pickles *et al.* [8] on changing the loop order in NEMO such that the depth index varies fastest, we found considerable benefit in manually permuting the order of each of the nested loops (rather than using the HMPP permute directive) as this then enabled many of them to be fused. This reduced the number of separate kernels launched on the GPU from 19 to just seven and in some cases effectively removed the cost of an individual kernel altogether. Overall, this process reduced the time spent in kernels on the GPU by approximately 30%. After this work, the GPU version took 0.015 s/call on the Tesla card (when data transport costs are excluded) – a factor of 2.6 times faster than on a single Nehalem core.

9 Slopes of Neutral Surfaces (*ldf_slp*)

Having ported all of the significant tracer-related routines, the next routine to tackle is *ldf_slp*. According to the profile in Table 1, this routine accounts for 4.2% of the execution time of the ORCA2_LIM configuration on 12 MPI processes of HECToR IIb. In contrast to *tra_zdf_imp* tackled in Section 8, *ldf_slp* contains several halo swaps as well as a call to a child routine (*ldf_slp_mxl*). For porting to the GPU the child routine had to be manually in-lined and then the subroutine broken into four kernels, separated by halo swaps. Using the test harness, this version of the code was found to take 0.079s/call on a single Nehalem core.

Optimisation of the four kernels and the halo exchanges was performed as for the previously-ported routines. HMPP's refusal to parallelise one loop led to the discovery of a bug in the original code (incorrect z-index used to access some arrays). With the correction of the bug, the loop parallelised as expected. In optimising some of the individual loops within the kernels it was found very beneficial (a 40% speed-up was obtained in one case) to remove division operations. This was possible because the triply-nested loops are parallelised over x and y with each CUDA thread then performing the loop over z. It was sometimes possible to hoist-out from that loop z-invariant quantities involving division, resulting in the saving of a considerable number of operations. The relatively unusual (for NEMO) compute-bound nature of some of the kernels was emphasised by a large speed-up (*e.g.* a factor of two) seen when moving from the Tesla to the Fermi GPU.

Once the individual kernels had been optimised, the time taken by the calculation as a whole was the same as when the original code was run on a single Nehalem core. However, the performance was dominated by the data transport to and from the CPU required for the halo swaps. As before, the quantity of data transported was reduced by altering the code so that only the halo regions of an array are transferred to and from the CPU. With this optimisation the whole calculation took 0.05s/call. However, 0.016s of this was taken up with the initial upload of data to the GPU. If we exclude this cost (on the basis that much of this data will already be on the GPU when running within NEMO) then the GPU version run on a Tesla card is a factor of two faster than the original on a single Nehalem core. The same code run on the Fermi GPU took 0.036s/call with 0.017s spent doing the initial data upload. Excluding this cost, the calculation is four times faster than it was on a single Nehalem core. This is the largest difference in performance between the two GPU cards seen for any of the subroutines tackled and emphasises the greater computational intensity of this case.

10 Merging the Accelerated Routines into NEMO

While it is interesting to see what speed-up may be obtained for a given kernel in isolation, it is not of much use unless it is possible to integrate the GPU version of the kernel back into NEMO without losing performance. HMPP's support for asynchronous data transfer is a crucial part of this work. We shall begin this section with a detailed description of the work involved in merging the accelerated version of *traldf_iso* into NEMO.

The first stage in this task is to alter the NEMO code so that the new version of the *traldf_iso* routine is called instead of the old. In order to minimise the changes to NEMO, we simply replaced the contents of the original routine with a call to the accelerated version. The latter was re-named *traldf_iso_hmpp* and contained within a new *traldf_iso_hmpp_mod* module. When using HMPP *traldf_iso* then contains the *callsite* for the codelet and also acts as a wrapper that brings the necessary module variables into scope ready for the call. Since the NEMO build system has a rule of one module per file with the filename matching the name of the module (for the purposes of dependency checking) it is necessary for *traldf_iso_hmpp_mod* to be contained in a separate file.

Having merged the new version of *traldf_iso* into NEMO so that it may be built and run in traditional form, the next task is to compile the GPU-enabled routine into the NEMO binary. In principle it should be possible to compile the whole of NEMO with the HMPP compiler since, in the absence of any HMPP directives, the code should just

	Time in tracers (s)	Time to step (s)
Nehalem	0.463	1.267
GPU sync. copy	0.361	1.134
GPU async. copy	0.359	1.129
GPU async. launch	0.369	1.174

Table 7: Performance of NEMO as a whole when run in ORCA2_LIM configuration. Each row gives the mean time of three runs where the time for each run is averaged over twenty time steps with the first and last steps excluded.

be compiled with the user-specified Fortran compiler. However, this approach resulted in a large number of errors relating to duplicate definitions of symbols. Therefore we adopted a strategy of building NEMO as usual (which works because the HMPP directives are standard Fortran comments), using HMPP to build the accelerated version of the subroutine and then re-linking this with the other object files to get a new NEMO binary.

We also added a new module, *tra_hmpp*, as a central location for adding *e.g.* the statements to map variables between different codelets. Since HMPP requires that all codelets of a given group be in a single file, and that all of the callsites, transfers, *etc.* also be in a single file we found it easiest to concatenate the *traldf_iso.F90* and *traldf_iso_hmpp_mod.F90* files with *tra_hmpp.F90* before compiling it with HMPP. A script to do all of this within the default NEMO directory structure (as of version 3.1 and later) is shown in Figure 7. This procedure is complicated slightly by the fact that any modules USE'd by a codelet must also have been compiled with HMPP - hence the need to re-compile the *par_kind.f90* module in Figure 7. Note also the need to link with the CUDA run-time library because of the use of the *cudaMalloc()* routine to access pinned memory.

Once we had a working version of NEMO with the *traldf_iso* routine running on a GPU and had verified the results, the overheads associated with device initialisation and data transfer had to be tackled. Unless a device is explicitly released at some point, HMPP defaults to releasing it after the first codelet call. Therefore, calls to allocate and release the device were added within separate subroutines in the *tra_hmpp* module; *hmpp_init* and *hmpp_final*. The former is called from within NEMO's *nemo_init* routine in *nemogcm.F90* and the latter from a new routine, *nemo_final* in the same file, once all of the time-stepping is complete. This guarantees that the device is only initiated once which is important since it is a relatively expensive blocking operation.

In order to deal with data transfer to the device we introduced another routine, *tra_ldf_prepare*, called from within the time-stepping routine, *stp*. Its purpose is to initiate asynchronous data transfers from the host memory to the GPU. As such, it is called as soon as the arrays involved in *tra_ldf_iso* are ready *i.e.* at the last point in *stp* beyond which they are not modified until the codelet itself is called. Fortunately the various tracer routines are such that this approach enables us to overlap the costly transfer of data to the GPU with the call to the *tra_dmp* routine.

Table 7 contains the results of timing how long it takes NEMO to do a single time step when running entirely on a Nehalem core and when running on a Nehalem core but with *tra_ldf_iso* running on the GPU. We see an overall speed-up of a single time-step of 7% which compares well with the 5.1% that the *tra_ldf_iso* routine was reported as accounting for (Table 1) when NEMO was profiled on 12 MPI processes on HECToR. A

```
#!/bin/sh
F90_HOST="ifort -r8 -O3"
F90="hmpp -d3 --keep -Xnvcc -pg ${F90_HOST}"
cd ORCA2_LIM/BLD/ppsrc/nemo
mv ../../obj/server.o{,.orig}
rm ../../obj/agrif*.o
mv ../../obj/tra_hmpp.o{,.orig}
cat tra_hmpp.f90 > tra_hmpp_merged.f90
#-------------------------------------------------------------------------
# For ldfslp
mv ../../obj/ldfslp.o{,.orig}
mv ../../obj/ldfslp_kernels_mod.o{,.orig}
cat ldfslp_kernels_mod.f90 >> tra_hmpp_merged.f90
cat ldfslp.f90 >> tra_hmpp_merged.f90
#-------------------------------------------------------------------------
# For traldf_iso
cat traldf_iso_hmpp_mod.f90 >> tra_hmpp_merged.f90
cat traldf_iso.f90 >> tra_hmpp_merged.f90
mv ../../obj/traldf_iso.o{,.orig}
mv ../../obj/traldf_iso_hmpp_mod.o{,.orig}
#-------------------------------------------------------------------------
# Build module USE'd by the ones that we're building with HMPP
$F90 -c -I../../inc par_kind.f90
# Now build HMPP form of accelerated modules
$F90 -c -I../../inc tra_hmpp_merged.f90
# Re-link the executable with the new object files
$F90 tra_hmpp_merged.o  ../../obj/*.o ~/WORK/Intel/lib/libnetcdf.a \
   -o nemo_hwa.exe -L${CUDA_HOME}/lib64 -lcudart
if [ -f "nemo_hwa.exe" ]
then
    mv nemo_hwa.exe ../../bin/.
else
    echo "Build failed: no executable produced."
fi
cd -
```

Figure 7: Build script for NEMO with the HMPP-accelerated version of the *ldfslp* and *traldf_iso* and routines. Sections for the other routines are omitted for brevity.

small speed-up was measured when data was uploaded asynchronously to the GPU. We also tried running the codelet asynchronously on the GPU and overlapped its execution with that of a different tracer-related routine. However, this proved to be slightly slower than using synchronous execution. The reason(s) for this are not fully understood but it is likely to be related to the fact that execution of the kernel on the GPU only takes approximately 0.02s.

10.1 Incorporating the GPU versions of *traadv_tvd*, *trazdf_imp* and *ldf_slp* into NEMO

The new version of the *traadv_tvd* routine was built into the NEMO executable following the approach taken for *tra_ldf_iso*. Since a GPU device must be freed before it can be allocated for use by a different group of codelets it was decided to avoid the associated performance penalty by putting all of the codelets into a single ('tracers') group. Further, as HMPP requires all codelets to be in a single source file, the build script shown in Figure 7 was extended so that all of the codelets are concatenated with *tra_hmpp.F90* file before it is compiled.

We were able to remove some data transfers by keeping two arrays on the GPU between the end of the *traldf_iso* codelet and the start of the codelets associated with *traadv_tvd*. (This was achieved by use of the HMPP *map* directive.) Figure 8 shows the effect of moving the *traadv_tvd* routine off the Nehalem CPU and onto the Tesla GPU. With both *traadv_tvd* and *traldf_iso* running on the GPU, NEMO is 10.8% faster than the case where it executes only on a single Nehalem core.

Figure 8 also shows the performance gains obtained when *trazdf_imp* and finally *ldf_slp* were moved onto the GPU. Note that the data transfer time did not increase significantly when doing this as, again, some uploads and downloads could be avoided by using HMPP's *map* functionality to make use of data already loaded to the card for the previous codelets. For this to be effective, the uploads and downloads that HMPP automatically generates have to be suppressed by setting their 'io' state appropriately and using the 'noupdate' directive at the callsite to prevent uploads. (Setting the 'io' state for a variable overrides the direction that HMPP otherwise takes from the Fortran INTENT attribute for any given routine argument.)

The importance of being able to do asynchronous data transfer to and from the device has already been mentioned. For an array to be involved in such a transfer it must be allocated in pinned memory. Since there is currently no support for this within the directives frameworks,[2] that must be done via a call to a C routine which returns a pointer. As a result, the array itself must be declared as a Fortran POINTER (instead of an allocatable array). We have found that making this change can have unwanted side effects on the performance of code using this array on the CPU. Table 8 quantifies this for the case of the vertical physics routines when the weighting and masking arrays are converted from allocatable arrays; first to pointers (albeit allocated in the normal way) and then to pointers to pinned memory. Declaring the arrays as pointers clearly has a detrimental effect; when using the Intel compiler, the time spent doing vertical physics increases by 8% compared to the case where they were simply declared ALLOCATABLE. However, when the arrays are moved into pinned memory the situation improves and the

[2]PGI's CUDA Fortran adds the capability to specify the 'PINNED' attribute for an array declaration. Since CUDA Fortran can be used with the PGI Accelerator Directives this provides a workaround when using the latter.

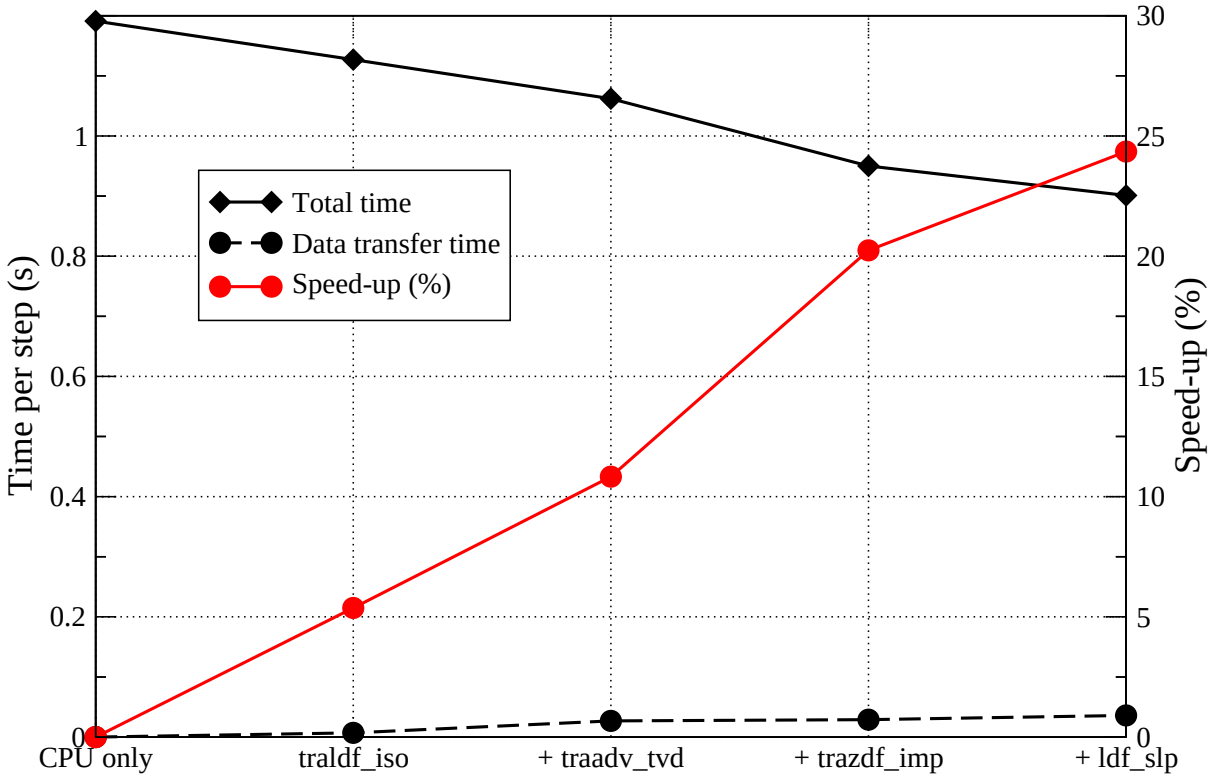

Figure 8: The reduction in execution time (black diamonds) and corresponding speed-up (red) for NEMO as a whole as routines are moved from a Nehalem CPU core onto a Tesla GPU. The black circles show the total amount of time spent transferring data to and from the GPU during each time-step. Lines are guides to the eye.

	Mean time per step (s)	
	Intel v. 11.1	PGI v. 11.9
Original	0.191	0.276
With pointers	0.207	0.286
With pointers and pinned memory	0.198	0.283

Table 8: The time spent in the vertical physics part of NEMO and the effect of changing the weighting and masking arrays to pointers and then to pinned memory. Times are averages over three, 55-step runs of the ORCA2_LIM configuration. Code was linked against version 3.2 of the CUDA run-time library from NVIDIA.

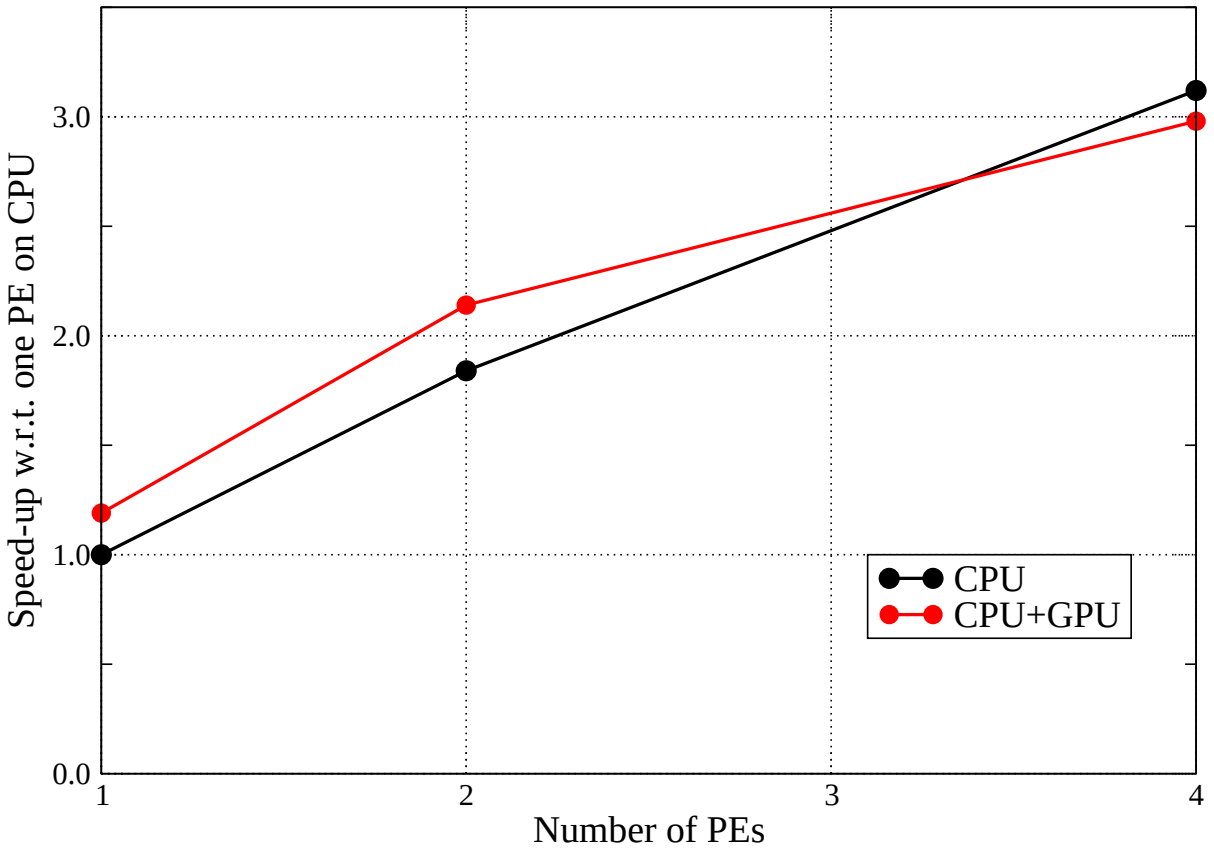

Figure 9: Comparison of NEMO in MPI mode with and without running the accelerated routines on the GPU. Runs done on Nehalem CPUs and Tesla GPUs. Sockets kept minimally populated to maximise available memory bandwidth. Lines are guides to the eye.

affected code is now only some 4% slower, presumably because preventing the arrays from being paged-out reduces the time the CPU must wait for them to be loaded and so improves memory bandwidth. When using the PGI compiler the pattern is the same although moving to pointers only slows the code by 4%. Changing to pinned memory slightly reduces the slow-down to 3%. Happily, in this particular case the arrays involved actually remain constant after initialisation and therefore must only be uploaded to the GPU once. We can therefore afford not to move them into pinned memory and thus can avoid the associated performance hit in the vertical physics section. Obviously this may not always be the case and therefore it is important to be aware of the possibility of slowing down code that is not running on the GPU.

Finally, we experimented with running the MPI version of NEMO with the accelerated routines. Performance plots are shown in Figure 9 for runs done with and without using GPUs. When GPUs are used, each MPI processing element (PE) is associated with one GPU. The linux 'taskset' command was used in conjunction with 'mpirun' in order to ensure that CPU processes were shared equally between sockets. In moving subroutines to execute on the GPU we have reduced the amount of computation (time) available to be done in parallel. Therefore we expect the scaling of the GPU-enabled version of NEMO to be worse than that of the CPU-only version and the results in Figure 9 bear this out. In fact, when four MPI PEs are used we see that the CPU-only version is fastest. However, the ORCA2_LIM configuration is relatively small and by the time it has been

Routine	Time %	Computational intensity (ops/ref)
tra_adv_tvd	6.5	0.82
sol_pcg	6.4	0.78
tra_ldf_iso	5.9	0.88
ldf_slp	5.6	0.98
lim_rhg	3.8	1.17
tra_qsr	3.5	0.81
tra_adv_eiv	3.3	0.42
dyn_zdf_imp	2.8	0.57
tra_zdf_imp	2.6	0.70
mathelp_moycum	2.6	0.62
zdf_ddm	2.3	0.71
eos_insitu_pot	2.0	1.63
eos_bn2	2.0	1.69
eos_insitu	1.6	1.74

Table 9: Profile of NEMO run in serial on a single core of HECToR IIb for the ORCA2_LIM configuration.

sub-divided into four regions there is no-longer enough work for the GPUs to be effective.

11 Understanding GPU Performance

In an attempt to better understand the GPU performance, we used the craypat tool on HECToR to measure the computational intensity of NEMO's more significant subroutines. Table 9 contains another profile of NEMO running ORCA2_LIM but this time including the computational intensity of each subroutine. This shows that, in general, the routines accounting for a greater percentage of run-time have relatively low computational intensities; probably because these routines have greater dependence on available memory bandwidth. In turn, this makes them less suitable for execution on GPUs which favour high compute-intensity kernels. A second point to note is that *tra_ldf_iso* actually has a lower compute intensity than *lim_rhg* and yet performed much better on the GPU.

We also used the craypat API to instrument *traldf_iso* so as to be able to measure the computational intensity of the individual loops in the routine. For the most expensive loop (at line 257) the computational intensity is 0.95 operations per memory reference (ops/ref). For the second most expensive loop (line 373) it is 0.61 ops/ref. Both of these loops consist of triply-nested/3D loops over the full domain depth, height and width respectively giving ample scope to parallelise, even with the relatively small ORCA2 data set. This is to be contrasted with the loops in *lim_rhg* which are predominantly only doubly-nested/2D. To make matters worse, the outer loop over latitude is restricted to either the Arctic or Antarctic (the routine is called once for each case) – approximately 30% of the height of the full domain. The amount of parallelism in these loops is therefore greatly reduced in comparison to those in *traldf_iso*.

The low computational intensity of the routines in NEMO emphasises their need for high bandwidth between the processor and system memory. There are various synthetic benchmarks available which are able to measure this bandwidth. Results obtained using a modified version of STREAM [6] and (the CUDA part of) SHOC [2] are shown in Table 10.

	Units	Nehalem	Westmere	Tesla	Fermi
traldf_iso	(s/call)	0.039	0.032	0.021	0.018
Reads	(GB/s)	17.21[a]	19.85[a]	82.5[b]	93.2[b]
Writes	(GB/s)	11.52[a]	12.80[a]	73.8[b]	102.8[b]

a Results from modified STREAM
b SHOC results from [4]

Table 10: Kernel timings and memory bandwidths for single Nehalem and Westmere Intel CPUs and the Tesla and Fermi NVIDIA GPUs. The OpenMP version of the kernel was used on the CPUs and the PGI Accelerator Directives version on the GPUs. The CPUs were populated with as many OpenMP threads as they had cores – four for the Nehalem and six for the Westmere.

(We modified STREAM such that it separately measured the bandwidth obtained when reading-from and writing-to memory rather than doing a single copy operation.)

The figures in Table 10 show that the Tesla GPU offers a factor 4.8 improvement in memory bandwidth over a single Nehalem socket and the Fermi slightly more. Despite this, the improvement in kernel performance in going from the OpenMP version on a full Nehalem socket (four cores) to the HMPP version on a Tesla GPU is only a factor of 2.6. This may imply that the GPU version of the kernel is not getting the best out of the hardware.

The factor of 1.13 improvement in memory bandwidth (for reads, which are what dominates the *traldf_iso* kernel) in going from Tesla to Fermi is fairly close to the 1.17 speed-up seen when moving the PGI-accelerated kernel from Tesla to Fermi. This limited speed-up also implies that the kernel is unable to take advantage of the factor eight increase in double-precision compute capability of the Fermi.

12 Conclusions

Of the five NEMO routines ported to the GPU in this work, all bar one ran at at least twice the speed obtained on a single core of a Nehalem processor. In fact, even when data transfer costs are included, this remains true for three of the routines. Moreover they have been successfully merged back into the NEMO code without losing this speed-up (*i.e.* much of the data-transfer costs have been hidden and/or shared between the routines), such that these routines are no longer significant in the NEMO profile. However, the sea-ice routine proved very different. Despite having restricted ourselves to the computational part of the kernel (*i.e.* avoiding the thorny issue of data transport), we were unable to get a speed-up over just a single CPU core, even with the ORCA025 grid, because of the limited parallelism present in the loops.

However, comparing the performance of a single CPU core with that of a whole GPU with several hundred cores is inconsistent. We have therefore constructed OpenMP versions of some of the routines which are then able to utilize all of the cores on a CPU socket. (No OpenMP version of NEMO as a whole exists.) These OpenMP versions avoid the significant complexities of programming for two memory address spaces and achieved some 60–70% of the performance of the GPU. Note also that this was done on an Intel Westmere chip which has already been superseded by Intel's Sandy Bridge architecture.

Another useful metric to consider is the amount of time taken to port code to the

GPU. The work described in this report took a total of one year of effort, albeit with maybe two months spent learning to use the directives and associated software tools. In that time, five routines have been ported to GPU using directives, one of which was done first with PGI directives and then again with HMPP directives. In contrast, the two OpenMP implementations took at most a day each to complete and required minimal code changes.

As previously noted, the ORCA2_LIM configuration fits into approximately 1GB of RAM and since NVIDIA's Fermi GPUs have either 3 or 6GB of RAM, another possible approach is to run the whole of NEMO on the GPU. This avoids the difficult problem of minimizing the effect of the PCI-Express bottleneck but obviously requires that the whole code be ported – a non-trivial task that is also likely to have a huge effect on the code base.

We also note that the ORCA*_LIM configurations are only one example of the many model configurations that NEMO is capable of. It may be that aspects related to biologically-active tracers, for instance, are better suited to making use of a GPU.

Back in Section 3 we discussed our motivation for using only directives-based approaches to porting NEMO. A key reason for doing so was the desire to keep changes to the original source code to a minimum. We have found that porting compute-intensive loops to the GPU is relatively simple (although still more complex than using OpenMP) and therefore not overly intrusive. However, the fact that current system architectures have two memory address spaces separated by a slow interconnect is a key issue for an MPI code like NEMO. A great deal of work (and code) is required to efficiently manage data transfers between the two memory address spaces and this makes the resulting program bulky and fragile.

Taking a step back, NVIDIA, AMD and Intel are all working towards integration of the GPU with the CPU which would obviously eliminate the PCIe bottleneck and a lot of the programming pain. On the software side, the SuperComputing 2011 conference saw the announcement of OpenACC — a new, cross-platform collection of compiler directives for accelerator programming supported by Cray, PGI, CAPS and NVIDIA. With such industrial support, it seems likely that OpenACC will be the best directives option for porting existing scientific codes to make use of GPUs in the future.

13 Project Outputs

The outputs and dissemination activities resulting from this work have been as follows:

- Presentation at the University of Manchester GPU Club, Manchester, December 2011;
- Presentation at the 3rd UK GPU Computing Conference, London, December 2011;
- Presentation at the National Oceanographic Centre, Liverpool, December 2011;
- Presentation of results to the NEMO Systems Team and discussion of the implications for the NEMO roadmap (Paris, December 2011);
- Presentation at PRACE IIiP WP8 Meeting, Barcelona, February 2012;
- Daresbury Laboratory Technical Report (this document), March 2012;

- Kernel benchmark codes and GPU version of the NEMO code supplied to the NEMO Systems Team, March 2012.

14 Acronyms

GPU	(General-Purpose) Graphics Processing Unit
HECToR	High-End Computing Terascale Resource
LIM	Louvain-la-Neuve sea-Ice Model
MPI	Message Passing Interface
NEMO	Nucleus of the European Model of the Ocean
PE	(MPI) Processing Element
SIMD	Single Instruction Multiple Data

15 Acknowledgments

This work was funded by the Natural Environment Research Council. The cseht cluster was provided by the DiSCO group at Daresbury, funded under an EPSRC Service-Level Agreement. The PGI and HMPP technical support teams both provided very useful assistance and advice.

References

[1] S. Bouillon, M. A. Morales Maqueda, V. Legat, and T. Fichefet. An elastic-viscous-plastic sea ice model formulated on arakawa b and c grids. *Ocean Modelling*, 27:174–184, 2009.

[2] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. *The Scalable HeterOgeneous Computing (SHOC) Benchmarking Suite*, March 2010.

[3] T. Fichefet and M. A. Morales Maqueda. Sensitivity of a global sea ice model to the treatment of ice thermodynamics and dynamics. *Journal of Geophysical Research*, 102(C6):12609–12646, 1997.

[4] I. N. Kozin. GPU Benchmarks for High Performance Computing: SHOC. http://www.cse.scitech.ac.uk/disco/publications/-WorkingNotes.SHOC.pdf, January 2011.

[5] G. Madec. *NEMO ocean engine*. Note du Pôle de modélisation, Institut Pierre-Simon Laplace (IPSL), France, No 27, ISSN No 1288-1619, 2008.

[6] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, December 1995.

[7] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. The Top500 Supercomputing Sites. http://www.top500.org/.

[8] S. M. Pickles and A. R. Porter. Final Report for the dCSE Project: Developing NEMO for large multi-core scalar systems. To be published in July 2012.

[9] M. Wolfe and C. Toepfer. The PGI Accelerator Programming Model on NVIDIA GPUs. Part 3: Porting WRF. *PGI Insider*, October 2009.