УДК 519.644

НОВЫЕ ТЕХНОЛОГИИ ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ ВЫЧИСЛЕНИЙ: ЧИСЛЕННОЕ ИНТЕГРИРОВАНИЕ НА ГРАФИЧЕСКИХ ПРОЦЕССОРАХ[1]

А. Спенс[2], Н. Скотт[2], Ч. Гиллан[2]

Применение метода R-матрицы к изучению рассеивания электронов промежуточных энергий на атоме водорода приводит к необходимости вычисления большого количества двухэлектронных интегралов от числовых базисных функций. Каждый из этих интегралов может быть вычислен независимо от остальных, что позволяет эффективно распараллелить процедуру вычислений. Рассматривается параллельная реализация этой процедуры, использующая графический процессор (GPU) в качестве сопроцессора, что примерно в 20 раз ускоряет вычисления по сравнению с последовательной версией. Кратко описываются особенности выполняемых расчетов, которые делают применение GPU подходящим для эффективного решения ряда других вычислительных задач. Статья рекомендована к печати программным комитетом международной научной конференции "Математическое моделирование и вычислительная физика 2009" (MMCP2009, http://mmcp2009.jinr.ru).

Ключевые слова: численное интегрирование, метод R-матрицы, параллельная реализация, GPU.

1. Introduction. For the last two decades, most software applications have enjoyed regular performance gains, with little or no software modification, as each successive generation of microprocessors delivered faster CPUs. However, system builders have now hit physical limits which have slowed the rate of increase of CPU performance and the computing industry is moving inexorably towards multiple cores on a single chip. Nevertheless, general purpose many-core processors may not deliver the capability required by leading-edge research applications. High performance may be more cost-effectively achieved in future generation systems by heterogeneous accelerator technologies such as General Purpose Graphical Processing Units (GPGPUs), STI's Cell processors and Field Programmable Gate Arrays (FPGAs).

Innovative algorithms, software and tools are needed so that HPC application scientists can effectively exploit these emerging technologies. Recent progress in this area includes: Ramdas et al. [7] who emphasize the importance of SIMD algorithms; Baboulin et al. [2] who illustrate the potential of mixed-precision algorithms; and the development of numerical libraries that can effectively exploit multi-core architectures such as PLAMSA from the Innovative Computing Laboratory [1] and HONEI by van Dyk et al. [12].

In this paper we describe initial work on deploying the Fortran suite 2DRMP [3, 11] on NVIDIA GPUs. 2DRMP is a collection of two-dimensional R-matrix propagation programs aimed at creating virtual experiments on high performance and grid architectures to enable the study of electron scattering from H-like atoms and ions at intermediate energies [8]. We focus on one of the identified hot-spots [9], namely, the computation of two-dimensional radial or Slater integrals that occur in the construction of Hamiltonian matrices. Using the 2DRMP to model electron scattering by a hydrogen atom requires the calculation (by numerical quadrature) of millions of integrals of the form $I(n_1, l_1, n_2, l_2, n_3, l_3, n_4, n_4, \lambda) = \int\limits_0^a u_{n_1 l_1}(r) u_{n_3 l_3}(r) \big[r^{-\lambda-1} I_1 + r^{\lambda} I_2\big]\, dr$, where the inner integrals I_1 and I_2 are $I_1(r) = \int\limits_0^r t^{\lambda} u_{n_2 l_2}(t) u_{n_4 l_4}(t)\, dt$, $I_2(r) = \int\limits_r^a t^{-\lambda-1} u_{n_2 l_2}(t) u_{n_4 l_4}(t)\, dt$ and the required values of the orbital functions u_{nl} are computed numerically and tabulated in advance. The integrations are carried out numerically using Simpson's Rule and they all use the same mesh of typically 1001 values of r.

It would appear that at each quadrature point of the outer integral I the calculation of two inner integrals I_1 and I_2 is required, leading to a complexity for the entire algorithm of $O(N^2)$, where N is the number of data

[1] Work partially supported by U.K. Engineering and Physical Sciences Research Council (EPSRC) under grants EP/F010052/1, EP/G00210X/1.

[2] School of Electronics, Electrical Engineering & Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, UK; А. Спенс, профессор, e-mail: i.spence@qub.ac.uk; Н. Скотт, профессор, e-mail: ns.scott@qub.ac.uk; Ч. Гиллан, профессор, e-mail: c.gillan@ecit.qub.ac.uk

points. However, because the same mesh is used for both inner and outer integrals, it is possible to coalesce the inner and outer loops giving a complexity which is $O(N)$.

For further details, see Scott et. al. [11].

2. Graphical Processing Unit. Graphical Processing Units, which are co-processors providing significant floating-point capability originally developed to support computationally intensive graphics for computer gaming, have attracted significant recent interest from the scientific computing community [6]. The computational capacity for a relatively modest expenditure makes them a very attractive proposition, but attempts to use them for general-purpose computing have met with varying degrees of success.

The nVidia GeForce 8800 GTX used in this experiment has 768 MB of global memory, and all communication of data between the host computer and the GPU is by writing to and reading from this memory. The device has 128 processors (in groups of 8, each group constituting a multiprocessor) [5].

The processors all have access to the global memory, and there is also 8 KB of local on-chip memory on each multiprocessor which is much faster but can only be accessed by threads running on that multiprocessor. The programming model is that input data is copied to the device global memory and then many copies of the same function are executed as concurrent threads on the different processors, with results being written back to global memory. Finally, results are copied to the host from global memory. In order to compensate for the relatively slow access to global memory, it is important to use the local memory where possible. In addition, the GPU is able to interleave communication and computation and so it is recommended that there are many more threads of execution than there are physical processors.

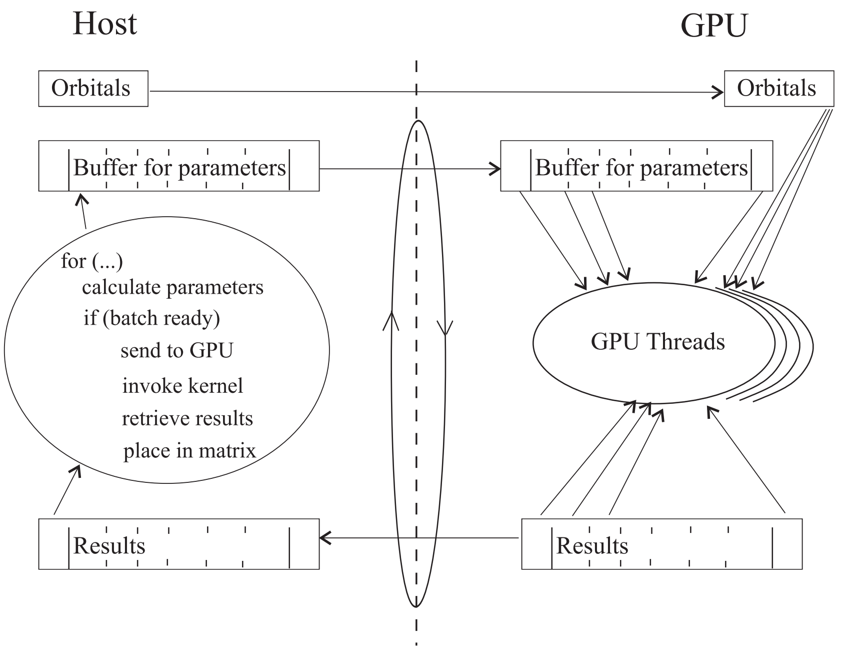

Fig. 1. Co-operation between host and GPU

It is important to note that on a given multiprocessor the execution is forced to be Single Instruction Multiple Data (SIMD) and that if one thread of execution is, for example, executing more iterations of a loop than the others, none of the threads will continue to the next statement in the program until all are ready to do so. This can have a significant impact on performance.

3. GPU Algorithm. There is no interaction amongst the calculations of the separate integrals, making this an obvious candidate for a parallel implementation with each integral being calculated within a separate thread. For an efficient GPU version of the algorithm, as mentioned above, it is essential that the parallel execution can be made SIMD, that is that each thread executes the same sequence of instructions. The control statements within the integration code consist only of `for` statements with fixed bounds. There are no `while` or `if` statements and so SIMD execution is achievable.

In the original program, there is a double loop over the rows and columns of the Hamiltonian matrix. For each matrix element, multiple integrals are evaluated and incorporated into the matrix immediately, which can involve some further simple arithmetic, depending on the particular element.

```
for (...)
{
    $n_1 = \ldots$; ...; $l_4 = \ldots$; $\lambda = \ldots$; factor = ...; mpos = ...;
    result = $I(n_1, l_1, n_2, l_2, n_3, l_3, n_4, l_4, \lambda)$;
    matrix[mpos] += result * factor ...;
}
```

In order to exploit the parallelism of the GPU, it is important that the program be in a position to calculate thousands integrals at once on the GPU, and so a buffer has to be placed between the loop over matrix elements which generates the parameters and the parallel evaluation of the integrals (see Fig. 1). A buffer element has to store details of the parameters and the operation which must be performed to incorporate the value of the integral into the matrix.

```
for (...)
{
    $n_1 = \ldots$; ... ; $l_4 = \ldots$; $\lambda = \ldots$; factor = ...; mpos = ...;
    buffer[bpos++] = ($n_1, \ldots, l_4$ , $\lambda$, factor, mpos);
}
copy buffer to GPU;
perform integrals in parallel;
copy results back;
for (...)
{
    factor = buffer[bpos].factor; mpos = buffer[bpos].mpos;
    result = results[bpos];
    hmat[mpos] += result * factor ... ;
}
```

3.1. GPU Issues. Several problems arose because of constraints of the hardware platform, namely:

(i) the GPU (nVidia GeForce 8800) does not support double precision arithmetic;

(ii) the access to device global memory is relatively slow.

3.1.1. Single Precision. The original program was written to use double precision floating point numbers but double precision arithmetic was not supported by the GPU and single precision had to be used instead. The precision of the results was still acceptable but there were initially some problems with underflow and overflow when calculating r^λ which required the algorithm to be re-cast. A tool such as CADNA [10] is essential to validate the efficacy of the single precision algorithm.

nVidia cards are now available which support double precision but as yet the extent of parallelism supported is significantly less then for single precision (8 times less on the GTX280).

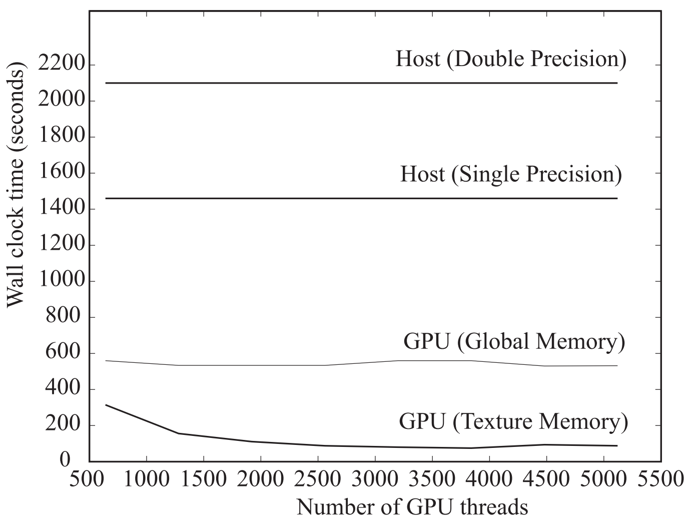

Fig. 2. Wall clock execution times for 99 million integrals

3.1.2. Memory Access. The recommended technique for overcoming the delays introduced by accessing device global memory is first to copy data to local on-chip memory which can be accessed much more quickly. For this to be possible, the amount of data required has to fit in the available memory (8 KB per multiprocessor) and, for it to be effective, each value should be accessed multiple times by a particular thread. In each integral, approximately 4,000 floating point values (16 KB) are required and typically each value is only accessed twice, so neither of these criteria was satisfied in this instance.

There is however an additional method provided by CUDA for reducing memory access times. Data can be stored in *texture memory* provided that it is not modified by the GPU code. This is still device global memory but, because it is guaranteed to be constant during the execution of a given kernel function, it can be efficiently cached for access by multiple threads on the same multiprocessor. This is reported to be particularly effective when there is locality of access to data across multiple threads, which does arise with this application when the values u_{nl} of the orbital functions are placed in texture memory.

4. Results. The host machine on which these experiments were carried out has a 3 GHz processor with 4 cores, although no use was made of the multiple cores during the experiments described here. The GPU was a nVidia GeForce 8800 GTX. The problem instance required the calculation of some 99 million integrals.

As illustrated in Fig. 2, the number of GPU threads used made little difference when the orbitals were all accessed directly from global memory. When texture memory was used, the execution time did decrease as the number of threads increased until there were nearly 4,000 threads. The table lists the execution times using the optimal number of GPU threads.

Using single precision arithmetic on both platforms, it can be seen that the GPU version with 3980 threads is approximately 20 times faster than the sequential version.

5. Conclusions. An efficient sequential solution already existed for the problem in question, yet the performance has been significantly enhanced by the GPU implementation. However, it has not been found to be trivial to obtain good results, in particular the use of different components of the memory hierarchy within the GPU was critical. Future work will look at the other programs within the R-matrix codes.

When trying to identify whether it is likely to be worth investing the effort of producing a bespoke GPU-based version of an exiting algorithm (there is a growing collection of pre-built numerical libraries for GPUs, e.g., CUBLAS [4] and HONEI [12]), the following considerations should be borne in mind. Does the bulk of the computation consist of many ($\gg 1000$) executions of the same code for different data? Are these executions independent? Are the executions SIMD, i.e., is the flow of control independent of the data? Is there significant re-use of input data across different executions? Is there a significant amount of computation for each item of input and output data? Is single precision arithmetic acceptable?

Best execution times

Environment	Time (seconds)
Host (Double Precision)	2100
Host (Single Precision)	1460
GPU (Global Memory)[3]	530
GPU (Texture Memory)[4]	75

If most (ideally all) of the questions can be answered in the affirmative, it would seem that a GPU implementation is worth considering.

References

1. Parallel linear algebra software for multicore architectures (PLASMA) (http://icl.cs.utk.edu/plasma/, visited 14 September, 2009).
2. *Marc Baboulin, Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov.* Accelerating scientific computations with mixed precision algorithms // Computer Physics Communications, In Press, 2008. DOI: 10.1016/j.cpc.2008.11.005.
3. *V.M. Burke, C.J. Noble, V. Faro-Maza, A. Maniopoulou, and N.S. Scott.* FARM_2DRMP: A version of FARM for use with 2DRMP // Computer Physics Communications, In Press, Accepted Manuscript, 2009. DOI: 10.1016/j.cpc.2009.07.017.
4. nVidia Corporation. CUBLAS library (http://www.nvidia.com/, visited 18 September, 2009).
5. nVidia Corporation. nVidia CUDA compute unified device architecture programming guide (version 2.0) (http://www.nvidia.com/cuda, visited 18 September, 2009).
6. *John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips.* GPU computing // Proceedings of the IEEE, 96: 879–899, 2008.
7. *Tirath Ramdas, Gregory K. Egan, David Abramson, and Kim K. Baldridge.* On ERI sorting for SIMD execution of large-scale hartree-fock SCF // Computer Physics Communications, 178 (11): 817–834, 2008. DOI: 10.1016/j.cpc.2008.01.045.
8. *N.S. Scott, V. Faro-Maza, M.P. Scott, T. Harmer, J.M. Chesneaux, C. Denis, and F. Jézéquel.* E-collisions using e-science // Physics of Particles and Nuclei Letters, 5 (3): 150–156, May 2008. DOI:10.1134/S1547477108030023.
9. *N.S. Scott, L.Gr. Ixaru, C. Denis, F. Jézéquel, J.-M. Chesneaux, and M.P. Scott.* High performance computation and numerical validation of e-collision software // In G Maroulis and T Simos, editors, Trends and Perspectives in Modern Computational Science, Invited lectures, Vol. 6 of Lecture Series on Computer and Computational Sciences, pages 561–570, 2006.
10. *N.S. Scott, F. Jézéquel, C. Denis, and J.-M. Chesneaux.* Numerical 'health check' for scientific codes: the CADNA approach // Computer Physics Communications, 176 (8): 507–521, 2007. DOI: 10.1016/j.cpc.2007.01.005.
11. *N.S. Scott, M.P. Scott, P.G. Burke, T. Stitt, V. Faro-Maza, C. Denis, and A. Maniopoulou.* 2DRMP: A suite of two-dimensional R-matrix propagation codes // Computer Physics Communications, In Press, Accepted Manuscript, 2009. DOI: 10.1016/j.cpc.2009.07.018.
12. *Danny van Dyk, Markus Geveler, Sven Mallach, Dirk Ribbrock, Dominik Göddeke, and Carsten Gutwenger.* HONEI: a collection of libraries for numerical computations targeting multiple processor architectures // Computer Physics Communications, In Press, 2009. DOI: 10.1016/j.cpc.2009.04.018.

Received November 2, 2009

[3]Using 1920 GPU threads.
[4]Using 3840 GPU threads.