Методика создания переносимых программ математического моделирования для различных типов гибридных суперЭВМ *

А.В.Снытников[1],Е.А.Генрих[1,2]

Институт Вычислительной Математики и Математической Геофизики СО РАН[1],
Новосибирский Государственный Университет[2]

Разнообразие архитектур суперЭВМ, представленных в Top500, а также необходимость выполнять расчеты по моделированию, например, динамики плазмы на наиболее мощных компьютерах, делают актуальной разработку методики создания программ, легко переносимых между наиболее широко используемыми архитектурами. Представленная методика основана на сведении к минимуму архитектурно-зависимых участков кода и оформлении их в виде процедурных переменных. Использование методики продемонстрировано на примере переноса программы для моделирования динамики плазмы методом частиц в ячейках с графических ускорителей на Intel Xeon Phi. С использованием представленной методики перенос осуществляется фактически за время перекомпиляции.

Ключевые слова: переносимость, гибридные суперЭВМ, графические ускорители вычислений, ускорители Intel Xeon Phi, процедурные переменные, вычислительная физика плазмы

1. Актуальность

Актуальность разработки методики создания переносимых программ связана с необходимостью использования наиболее мощных гибридных суперЭВМ, а такие сейчас строятся (в том числе) на базе Intel Xeon Phi. Кроме того, в докладе А.О.Лациса “Что же делать с этим многообразием суперкомпьютерных миров?” на конференции “Научный сервис в сети Интернет-2014” [1] была предложена методика создания единого переносимого программного обеспечения для решения вычислительных задач, которое могло бы использоваться на многих суперкомпьютерных архитектурах. Эта методика основана на использовании библиотеки BLAS, которая так или иначе существует на всех машинах. В данной работе предлагается еще один вариант ответа на вопрос о многообразии суперкомпьютерных миров.

В данной работе вопрос создания переносимых программ рассматривается на примере кода для моделирования динамики плазмы, тем не менее, эта же методика может быть применена к любым задачам математического моделирования, где есть сетка, модельные частицы или конечные элементы.

Необходимость применения наиболее мощных суперЭВМ для расчета динамики плазмы можно проиллюстрировать следующим образом. Согласно предварительным оценкам, основанным на работе [2], для решения задачи о моделировании излучения турбулентной плазмы в установке ГОЛ-3 (ИЯФ СО РАН), необходима сетка размером от $2000\times500\times500$ узлов при наличии 1000 модельных частиц в каждой ячейке, или всего около 10 млрд. модельных частиц. О том, какие мощности используются для проведения расчетов такой размерности, говорят следующие примеры. В докладе С.А.Горбаса(IBM) [3] на конференции «Научный сервис в сети интернет-2014» была представлена имитация космического пространства на Blue Gene/Q (500 тыс процессоров), проведенное в Аргоннской и Лос-Аламосской лабораториях. Моделирование проводилось с использованием 1,1 трлн. частиц). На крупнейшем

*Работа выполнена при поддержке гранта РНФ 16-11-10028. Разработка программы была поддержана грантами РФФИ 14-07-00241 и 16-07-00434. Расчеты были проведены с использованием ресурсов ЦКП Сибирский Суперкомпьютерный Центр ИВМиМГ СО РАН.

в мире суперкомпьютере Tianhe2 в Национальном Университете Оборонных Технологий (Китай) проведено моделирование космологической эволюции (нейтрино) с использованием 110 миллиардов частиц [4]. Следует отметить, что увеличение количества частиц важно не только для демонстрации возможностей кода, самое важное то, что это показатель качества расчетов, оценка точности расчетов, которые могут быть проведены, как показано в [5]. С помощью гибридного кода, созданного авторами статьи, было проведено моделирование релаксации электронного пучка в плазме с использованием **160 миллионов** частиц, на суперкомпьютере "Ломоносов"(НИВЦ МГУ). Дополнительным подтверждением актуальности данной работы является то, что упомянутые выше расчеты были проведены на различных типах вычислительных устройств.

Большое количество различных суперкомпьютерных архитектур приводит к необходимости разрабатывать отдельный вариант программы под каждую из них. В то же время наиболее распространенные в настоящее время суперкомпьютерные архитектуры строятся на основе одних и тех же принципов, т.е. кластеры с использованием ускорителей вычислений или просто кластеры. Это означает, что задача создания инструмента для облегченного (упрощенного), хотя и не автоматического перехода между двумя разными суперкомпьютерными архитектурами представляется осуществимой.

Вопросы портирования программ на Intel Xeon Phi, в частности, рассматриваются в [6]. Кроме того, в работе [7] изучается проблема достижения максимальной заявленной производительности в 1 Teraflops с помощью ускорителя Intel Xeon Phi. Сравнение производительности ускорителей Intel Xeon Phi и графических ускорителей на различных задачах проведено в [8–10].

Новизна методики переноса, созданной в диссертации заключается в разработке полуавтоматического средства переноса программ, которое с одной стороны, было бы эффективным, с другой стороны, обеспечивало бы полный контроль над процессом переноса для прикладного программиста.

2. Постановка задачи

Необходимо решить вопрос о переносе между наиболее распространенными типами суперкомпьютерных архитектур

1. Кластера на основе Nvidia Kepler
2. Кластера на основе Intel Xeon Phi
3. Кластера на основе Intel Xeon, AMD Opteron, Fujitsu SPARC64 и др.

В данной работе рассматривается перенос программы с GPU на Intel Xeon Phi (не наоборот!!!) и не рассматривается на данный момент вопрос оптимизации под ту или иную архитектуру.

Основные проблемы переноса с архитектуры CUDA [11–13] на архитектуру MIC [14,15] заключаются в следующем:

1. Компиляция ядер CUDA и в особенности вызовов ядер CUDA без компилятора Nvidia
2. Пропуск операций копирования между различными видами памяти в CUDA
3. Определение типов данных и ключевых слов, входящих в расширение языка C, используемое в CUDA.

Для обеспечения переносимости архитектурно-зависимые участки кода (ядра CUDA, функции CUDA API)

- Сводятся к минимуму

- Оформляются в виде процедур
- Выносятся во внешнюю подключаемую библиотеку

Сведение к минимуму архитектурно-зависимых участков кода, реализующего вычислительный алгоритм [16, 17] выполняется следующим образом. В коде имеется 15-20 вызовов небольших вычислительных процедур, выполняющих обработку:

- узлов сетки
- модельных частиц
- границ расчетной области

Эти процедуры оформлены в виде ядер CUDA. Таким образом, эти процедуры не могут быть скомпилированы с помощью компилятора Intel и пр. Основной принцип предлагаемой методики переноса программ: **сделать такой непереносимый участок кода по крайней мере единственным**.

Завершая постановку задачи, необходимо пояснить, почему в данной работе для создания переносимой программы не используются архитектурно-независимые инструменты, такие как OpenCL. Причин для этого две: поддержка OpenCL для MIC декларирована, но полностью пока не реализована, кроме того, технология CUDA предоставляет больше возможностей для оптимизации, чем OpenCL или OpenACC.

3. Описание методики

4. Универсальная процедура запуска

В качестве реализации сформулированного выше принципа (вынести все непереносимые элементы кода в одну процедуру) предлагается универсальная процедура запуска, которая показана на рисунке 1. На вход этой процедуре в качестве параметра подаются процедуры, которые запускаются из-под ядер CUDA. Далее в том случае, если этот код компилируется компилятором CUDA C/C++ и исполняется на машине с GPU, то процедурный параметр передается универсальному ядру (GPU_Universal_Kernel на рисунке) и запускается на GPU внутри ядра. Если же этот код компилируется компилятором Intel (например) и исполняется на ускорителе Intel Xeon Phi (в режиме native) или просто на многоядерном процессоре под OpenMP, то переданная в качестве параметра процедура будет просто вызываться в цикле. Цикл в данном случае имеет шестикратную степень вложенности (соответствующую шести размерностям сетки потоковых блоков, используемой в CUDA - три размерности сетки и три размерности потокового блока).

Небходимость использования процедурных переменных связана с ограниченностью поддержки аппарата объектно-ориентированного программирования C++ в CUDA. По этой причине нельзя оформит вычислительные процедуры как виртуальные, и вызывать тот их вариант, который соответствует архитектуре.

4.0.1. Унифицированная сигнатура расчетных процедур

Все расчетные процедуры, которые ранее запускались как ядра CUDA, должны быть оформлены в виде процедур с единой сигнатурой (одинаковый тип возвращаемого значения и одинаковый набор параметров), показанной на рисунке 2.

Далее, необходимо отработать расширения языка C, используемые в CUDA C/C++, например, специальные типы данных (int3, double3, dim3, и пр.). Их можно либо доопределить, либо при возможности скопировать файл cuda.h. Специальные ключевые слова: __global__ , __device__, и др. можно замаскировать с помощью директив условной компиляции.

```
int Kernel_Launcher(
Cell<Particle>  **cells,KernelParams *params,
unsigned int grid_size_x,unsigned int grid_size_y,
unsigned int grid_size_z,
        unsigned int block_size_x,unsigned int block_size_y,
        unsigned int block_size_z,
        int shmem_size,
        SingleNodeFunctionType h_snf,char *name)
{
struct timeval tv1,tv2;
#ifdef __CUDACC__
    dim3 blocks(grid_size_x,grid_size_y,grid_size_z),
    threads(block_size_x,block_size_y,block_size_z);

    gettimeofday(&tv1,NULL);
    GPU_Universal_Kernel<<<blocks,threads,shmem_size>>>
    (cells,params,h_snf);

    DeviceSynchronize();
    gettimeofday(&tv2,NULL);
#else
    char hostname[1000];
    gethostname(hostname,1000);

#ifdef OMP_OUTPUT
    printf("function␣%s␣executed␣on␣%s␣\n",name,hostname);
#endif

    gettimeofday(&tv1,NULL);

    omp_set_num_threads(OMP_NUM_THREADS);

#pragma omp parallel for
    for(int i = 0;i < grid_size_x;i++)
    {
       //      ....
                h_snf(cells,params,i,j,k,i1,j1,k1);
// ....
```

Рис. 1: Универсальная процедура запуска

Остаются функции CUDA API: копирование из одного типа памяти в другой, обработка ошибок и пр. Эти функции должны вызываться не напрямую, как CUDA API, а через функции-обертки, вынесенные в отдельный заголовочный файл.

```
typedef void (*SingleNodeFunctionType)(GPUCell<Particle>
**cells,KernelParams *params,
                unsigned int bk_nx,unsigned int bk_ny,unsigned int bk_nz,
                unsigned int nx,unsigned int ny,unsigned int nz
                );
```

Рис. 2: Тип универсальной счетной процедуры

Этот процесс будет показан на примере процедуры расчета электрического поля из класса GPU-пространство моделирования ??. На листинге 3 показано первоначально имеющееся ядро CUDA, предназначенное для вычисления электрического поля в одном узле сетки. При этом реальные вычисления проводятся процедурой emeElement. Само ядро только лишь определяет узел сетки с помощью внутренних индексов CUDA, передает параметры (шаги сетки, временной шаг, магнитное поле, соответсвующую компоненту тока, как описано в статье [17]) и вызывает процедуру emeElement.

```
template <template <class Particle> class Cell >
__global__ void GPU_eme(

                            Cell<Particle>  **cells,
                            int i_s,int l_s,int k_s,
        double *E,double *H1, double *H2,
        double *J,double c1,double c2, double tau,
        int dx1,int dy1,int dz1,
        int dx2,int dy2,int dz2
                )
{
        unsigned int nx = blockIdx.x*blockDim.x + threadIdx.x;
        unsigned int ny = blockIdx.y*blockDim.y + threadIdx.y;
        unsigned int nz = blockIdx.z*blockDim.z + threadIdx.z;
        Cell<Particle>  *c0 = cells[0];




        emeElement(c0,i_s+nx,l_s+ny,k_s+nz,E,H1,H2,
                                                J,c1,c2,tau,
                                                dx1,dy1,dz1,dx2,dy2,dz2);
}
```

Рис. 3: Первоначально имеющееся ядро CUDA, предназначенное для вычисления электрического поля.

```
__host__ __device__
        void emeElement(Cell<Particle> *c,int i,int l,int k,double *E,
                        double *H1, double *H2,
                        double *J,double c1,double c2, double tau,
                        int dx1,int dy1,int dz1,int dx2,int dy2,int dz2
                        )
        {
           int n  = c->getGlobalCellNumber(i,l,k);
          int n1 = c->getGlobalCellNumber(i+dx1,l+dy1,k+dz1);
          int n2 = c->getGlobalCellNumber(i+dx2,l+dy2,k+dz2);

          E[n] += c1*(H1[n] - H1[n1]) - c2*(H2[n] - H2[n2]) - tau*J[n];
        }
```

Рис. 4: Процедура, реально выполняющая вычисление электрического поля в узле сетки

4.1. Механизм передачи параметров расчетных процедур

Из рисунка 4 видно, что процедура emeElement имеет некий набор параметров, более того, ясно, что у всех расчетных процедур набор параметров может быть различным. Для того, чтобы привести все такие процедуры к единому формату, так чтобы можно было передавать эти процедуры через параметр типа SingleNodeFunctionType (рисунок 2), была введена структура KernelParams, включающая в себя все возможные наборы параметров для всех расчетных процедур 6. Задача упрощается за счет того, что наборы параметров различных процедур имеют много общего в рамках реализованного вычислительного алгоритма.

Образец приведения расчетной процедуры к универсальному формату показан на рисунке 5. Здесь наиболее важное отличие от 3 состоит в том что, что координаты обрабатываемого узла *nx,ny,nz* вычисляются на основе параметров процедуры, а не на основе внутренних переменных CUDA (*blockId,blockDim.x,threadIdx*).

Таким образом, перед каждым запуском расчетной процедуры (того, что прежде было запуском ядра CUDA), заполняются те поля структуры *KernelParams*, которые нужны именно для данной процедуры. Далее вызывается процедура *Kernel_Launcher*, рисунок 1,

```
template <template <class Particle> class Cell >
__device__ void GPU_eme_SingleNode(

                Cell<Particle>  **cells,
                KernelParams *params,
                unsigned int bk_nx,unsigned int bk_ny,unsigned int bk_nz,
                unsigned int tnx,unsigned int tny,unsigned int tnz
        )
{
        unsigned int nx = bk_nx*params->blockDim_x + tnx;
        unsigned int ny = bk_ny*params->blockDim_y + tny;
        unsigned int nz = bk_nz*params->blockDim_z + tnz;
        Cell<Particle>  *c0 = cells[0];

        emeElement(c0,params->i_s+nx,params->l_s+ny,params->k_s+nz,
                        params->E,params->H1,params->H2,
                        params->J,params->c1,params->c2,params->tau,
                        params->dx1,params->dy1,params->dz1,
                        params->dx2,params->dy2,params->dz2);
}
```

Рис. 5: Ядро CUDA, предназначенное для вычисления электрического поля, адаптированное под универсальный формат вызова

которой передаются массив всех ячеек сетки *cells*, указатель на структуру, содержащую набор параметров *params*, размерности сетки и блока и вызываемая расчетная процедура *h_snf*.

```
typedef struct {
        double d_ee; //electric energy
        double *d_Ex,*d_Ey,*d_Ez; // electric field
        double *d_Hx,*d_Hy,*d_Hz; // magnetic field
        double *d_Jx,*d_Jy,*d_Jz; // currents
        double *d_Rho;
        int nt;                   // timestep
        int *d_stage; // checking system (e.g. for flow-out particles)
        int *numbers;             // number of particles in each cell
        double mass,q_mass;
        double *d_ctrlParticles;
        int jmp;
//                                   for periodical FIELDS
        int i_s,k_s;              //
        double *E;                // the field
        int dir;                  // the direction being processed
        int to,from;              // the range along the direction

//                                   for periodical CURRENTS
        int dirE;                 // directions
        int N;                    // variables

//                                   electric field solver
        int l_s;                  // variables
        double *H1,*H2;           // magnetic fields (orthogonal)
        double *J;                // current
        double c1,c2,tau;         // grid steps squared
        int dx1,dy1,dz1,dx2,dy2,dz2; //shifts

//                                   magnetic field solver
        double *Q;                // magnetic field at half-step
        double *H;                // magnetic field
        double *E1,*E2;           // electric fields (orthogonal)
        int particles_processed_by_a_single_thread;
        unsigned int blockDim_x,blockDim_y,blockDim_z;
//                                   block for field solver
} KernelParams;
```

Рис. 6: Структура, включающая в себя все возможные наборы параметров для всех расчетных процедур

В заключение необходимо ответить на вопрос, насколько серьезно отличается архитектура CUDA от MIC и, соответственно, насколько отличаются стратегии оптимизации?

Основным вопросом в обоих случаях является локальность данных, которая выражается для случая моделирования плазмы в том, что все модельные частицы в каждой ячейке должны хранится близко в памяти вычислительного устройства, будь то процессор общего назначения,или ускоритель любого типа. Причем важно отметить, что вычислительная производительность зависит от локальности в гораздо большей степени, чем от архитектурно-зависимых оптимизаций.

Далее, одним из основных инструментов повышения производительности для Intel Xeon Phi является векторизация. Подготовка программы для векторизации, направленной на MIC во всяком случае, не ухудшит производительность CUDA.

Наконец, имитация вызова ядер CUDA с помощью директив OpenMP для MIC требует дополнительных индивидуальных настроек для достижения высокой производительности, которые априорно не ясны, но могут быть внесены в код без ущерба для производительности на CUDA.

5. Производительность

В таблице 1 показано сравнение производительности программы на графическом ускорителе Nvidia Kepler, на процессоре Intel Xeon и на ускорителе Intel Xeon Phi. В таблице рассматривается наиболее трудоемкий этап программы - движение модельных частиц. Производительность рассчитывается исходя из того, что на каждую частицу делается приблизительно 250 операций.

Таблица 1: Сравнение производительности программы моделирования динамики плазмы (этап расчета движения модельных частиц) на различных вычислительных устройствах

название этапа	вычислительная производительность, GygaFlops
Nvidia Kepler	3.23
Intel Xeon	0.06
Intel Xeon Phi	0.19

Можно обратить внимание на то, что производительность для Intel Xeon Phi заметно меньше заявленной пиковой в 1 Teraflops, но на данный момент специальные методы оптимизации под это ускоритель не использовались, при том, что достижение пиковой производительность на методе частиц в ячейках является нетривиальной задачей [?].

В данном случае в расчете использовано 6.4 млн. частиц (в среднем 3.5-4 тыс. в одной ячейке). Отметим также что процессор Intel Xeon имеет 6 ядер (Intel Xeon Phi – 61), т.е. всего в 10 раз больше ядер, причем существенно менее мощных, таким образом ускорение в 3 раза при переходе с Intel Xeon на Intel Xeon Phi - это приемлемо.

6. Заключение: основные принципы методики

Или как писать программу, чтобы она легко переносилась?

1. Оформлять все фрагменты программы, обрабатывающие узлы сетки, частицы, элементы базиса, собств. функции и пр. в виде небольших процедур с унифицированной сигнатурой

2. Вызывать эти процедуры не в ядрах CUDA или в циклах OpenMP, а с помощью универсальной процедуры запуска

3. Архитектурно-зависимые функции (копирование данных, определение ошибки и пр.) вызывать не напрямую, а через библиотеку подстановок archAPI.h

Литература

1. Лацис А.О. Ну и что же нам теперь делать с этим многообразием суперкомпьютерных миров? Попытка конструктивного систематического подхода к сопоставлению суперкомпьютерных архитектур. In *Научный сервис в сети Интернет: многообразие суперкомпьютерных миров: Труды Международной суперкомпьютерной конференции*, pages 188–190. Изд-во МГУ, 2014.

2. Annenkov V.V., Timofeev I.V., Volchok E.P. Simulations of electromagnetic emissions produced in a thin plasma by a continuously injected electron beam. //http://arxiv.org/abs/1512.07167

3. Горбас С.А. Высокопроизводительные решения ibm. In *Научный сервис в сети Интернет: многообразие суперкомпьютерных миров: Труды Международной суперкомпьютерной конференции (22-27 сентября 2014 г., г. Новороссийск).*, 2014.

4. Yutong Lu. Overview of tianhe2 system and applications. In *Russian Supercomputer Days*, 2015.

5. K. V. Lotov, I. V. Timofeev, E. A. Mesyats, A. V. Snytnikov, and V. A. Vshivkov. Note on quantitatively correct simulations of the kinetic beam-plasma instability. *Physics of Plasmas*, 22(2), 2015.

6. C.Rosales. Porting to the intel xeon phi: Opportunities and challenges (abstract), 2013.

7. Hiroshi Nakashima. Manycore challenge in particle-in-cell simulation: How to exploit 1 {TFlops} peak performance for simulation codes with irregular computation. *Computers & Electrical Engineering*, 46:81 – 94, 2015.

8. Dmitry I. Lyakh. An efficient tensor transpose algorithm for multicore cpu, intel xeon phi, and {NVidia} tesla {GPU}. *Computer Physics Communications*, 189:84 – 91, 2015.

9. T. Liu, X.G. Xu, and C.D. Carothers. Comparison of two accelerators for monte carlo radiation transport calculations, nvidia tesla {M2090} {GPU} and intel xeon phi 5110p coprocessor: A case study for x-ray {CT} imaging dose calculation. *Annals of Nuclear Energy*, 82:230 – 239, 2015. Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013, {SNA} + {MC} 2013. Pluri- and Trans-disciplinarity, Towards New Modeling and Numerical Simulation Paradigms.

10. M. Bernaschi, M. Bisson, and F. Salvadore. Multi-kepler {GPU} vs. multi-intel {MIC} for spin systems simulations. *Computer Physics Communications*, 185(10):2495 – 2503, 2014.

11. Nvidia cuda home page.

12. Харламов А. А. Боресков А. В. *Основы работы с технологией CUDA*. ДМК Пресс, 2010.

13. Эдвард Кэндрот Джейсон Сандерс. *Технология CUDA в примерах.* ДМК Пресс, 2011.

14. Timothy Prickett Morgan. Intel slaps xeon phi brand on mic coprocessors.

15. Jianbin Fang, Henk Sips, LiLun Zhang, Chuanfu Xu, Yonggang Che, and Ana Lucia Varbanescu. Test-driving intel xeon phi. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 137–148, New York, NY, USA, 2014. ACM.

16. Boris Glinskiy, Igor Kulikov, Alexey Snytnikov, Alexey Romanenko, Igor Chernykh, and Vitaly Vshivkov. Co-design of parallel numerical methods for plasma physics and astrophysics. *Supercomputing frontiers and innovations*, 1(3), 2015.

17. Дудникова Г.И. Вшивков В.А., Вшивков К.В. Алгоритмы решения задачи взаимодействия лазерного импульса с плазмой. *Вычислительные технологии*, 6(2):47–63, 2001.