УДК 004.4422

СРАВНЕНИЕ ЭФФЕКТИВНОСТИ ТЕХНОЛОГИЙ OpenMP, nVidia CUDA И StarPU НА ПРИМЕРЕ ЗАДАЧИ УМНОЖЕНИЯ МАТРИЦ

К.М. Ханкин

EFFICIENCY COMPARISON OF OpenMP, nVidia CUDA AND StarPU TECHNOLOGIES BY THE EXAMPLE OF MATRIX MULTIPLICATION

K.M. Khankin

Приведено описание технологий OpenMP, nVidia CUDA и StarPU, варианты решения задачи умножения двух матриц с задействованием каждой из технологий и результаты сравнения реализаций по требовательности к ресурсам.

Ключевые слова: технологии OpenMP, CUDA, StarPU; энергосбережение.

In the article the description of OpenMP, nVidia CUDA and StarPU technologies, probable solutions of two matrix multiplication problem applying these technologies and the result of solution comparison by the criterion of resource consumption are considered.

Keywords: OpenMP, CUDA, StarPU technologies; energy saving.

Введение

В докладе на 64-й научной конференции «Наука ЮУрГУ», которая состоялась в апреле 2011 года, было обращено внимание на технологии OpenMP, nVidia CUDA и систему диспетчеризации StarPU, позволяющие автоматизировать распределение задач между узлами вычислительной системы и потенциально за счёт этого снизить энергопотребление. Так как эти три технологии используют различные подходы к решению задачи распределения вычислений, имеет смысл сравнить, насколько эффективна, универсальна и проста в использовании каждая из них.

Технология OpenMP

OpenMP (Open Multiprocessing) – это прикладной интерфейс программирования (API), предназначенный для параллельного программирования с применением разделяемой памяти. Поддерживаются языки программирования C, C++ и Fortran, операционные системы Solaris, AIX, HP-UX, Linux, Max OS X, Windows. Разработка OpenMP ведётся с участием крупных ИТ-компаний, таких как AMD, Intel, IBM, Cray и др. На момент написания статьи последняя версия стандарта – 3.1 [1]

OpenMP представляет собой набор директив для компилятора, библиотечных вызовов и переменных окружения, соответственно для использования OpenMP нужен компилятор, умеющий учитывать директивы OpenMP. Работа по задействованию параллелизма возлагается на программиста, который должен указывать среде времени исполнения и компилятору, что они должны делать. OpenMP не занимается обнаружением конфликтов данных, поиском состояний состязания и т. д. Таким образом, OpenMP позволяет создавать переносимые программы, в которых за корректность параллельного алгоритма отвечает программист.

OpenMP использует модель fork-join. API допускает написание программ, которые корректно работают, исполняясь и параллельно, и последовательно. Если запрашивается последовательное исполнение программы, библиотека OpenMP подменяет реальные функции заглушками, то есть такты на работу OpenMP всё же тратятся.

Для создания нового потока необходимо указать директиву parallel. Эта директива приводит к созданию набора (team) из потока, который обработал директиву, и любого количества новых потоков (в том числе и ни одного), при этом поток, обработавший директиву, становится главным в наборе. Внутри блока parallel могут располагаться несколько заданий (task), каждое задание будет присвоено отдельному потоку. В конце блока parallel находится неявный барьер, позволяющий

Ханкин Константин Михайлович – аспирант кафедры ЭВМ, Южно-Уральский государственный университет; hc@comp.susu.ac.ru

Khankin Konstantin Mikhaylovich – Post-Graduate Student of Electronic Computer Department of South Ural State University; hc@comp.susu.ac.ru

синхронизировать потоки, работающие над задачами. Вся программа сама по себе уже находится в неявно заданном блоке parallel.

Таким образом, OpenMP позволяет программисту указывать блоки, которые можно распараллелить, и задания, которые должны выполняться параллельно. OpenMP не способен задействовать средства GPGPU. В библиотеку также входят примитивы синхронизации, отличные от барьера. Так как для исполнения используется модель потоков, то память является общей для всех порождённых потоков. OpenMP можно рассматривать как стандартизованное средство создания переносимых параллельных программ, исполняемых на центральном процессоре (или процессорах), позволяющее программисту не задумываться об особенностях средств параллельного программирования на каждой платформе.

Технология nVidia CUDA

CUDA (Compute Unified Device Architecture) – это программно-аппаратная платформа для параллельных вычислений, задействующая ресурсы графического процессора nVidia для неграфических вычислений [2]. Разработка CUDA была начата в 2006 году. Поддерживаются языки программирования C, C++ и Fortran, операционные системы Windows 7, Windows XP, Windows Vista, Linux, Mac OS X. Все поддерживаемые графические процессоры nVidia разбиты на несколько классов в зависимости от аппаратных средств и поддерживаемых операций, поддерживается обратная совместимость. CUDA доступна как на бытовых видеоадаптерах, так и в виде специализированных сопроцессоров Tesla и Fermi. CUDA используется и при построении кластеров, например, в кластере Tianhe-1A, что позволило ему попасть на 1 место 36 редакции списка Top500 и оставаться на 2 месте в 37 и 38 редакциях при энергопотреблении 4 МВт [3], тогда как кластер, опередивший его, потребляет около 12,5 МВт и имеет почти в 4 раза больше вычислительных ядер общего назначения и в 7 раз больше ОЗУ при выигрыше в производительности всего почти в 3 раза [4].

CUDA использует модель SIMD, что накладывает ограничения на алгоритмы, которые можно эффективно исполнять на такой платформе. Так, например, при выполнении условного оператора возникает проблема с обработкой альтернативной ветки – приходится выполнять операции, результат которых будет отброшен. Однако CUDA маскирует эту проблему, программисту не приходится решать её самому. Из-за особенностей процессора CUDA может работать только с ограниченным набором данных – одномерные, двумерные и трёхмерные блоки с данными одного типа. Каждый блок может обрабатываться несколькими потоками, использующими разделяемую память. CUDA позволяет выполнять вычисления одновременно на центральном и графическом процессорах за счёт асинхронности вызова вычислительного ядра. С технологией GPUDirect возможно осуществлять прямой доступ к памяти (DMA) графического процессора.

С точки зрения программиста программа с использованием CUDA состоит из нескольких фаз, каждая из которых выполняется на центральном или графическом процессоре. Функции, которые могут быть исполнены на графическом процессоре, помечаются специальными ключевыми словами (расширение языка) и называются ядрами. Задействованные структуры данных также помечаются ключевыми словами. При компиляции специальный препроцессор разделяет фазы на две части, эти части компилируются отдельно, то есть модификация компилятора исходного языка программирования не требуется. Адресные пространства центрального и графического процессоров независимы друг от друга, что вызывает задержку при копировании данных между ними. Опять же из-за архитектуры графического процессора память разделяется на несколько видов – регистры (на каждый поток), локальная (на каждый поток), разделяемая (на набор потоков), глобальная (на сетку данных), константная (на сетку данных). Разные типы памяти имеют разную производительность и область видимости, некоторые виды памяти не имеют кэша, некоторые имеют.

Таким образом, CUDA позволяет программисту ускорять выполнение участков кода, хорошо ложащихся на модель исполнения SIMD, с привлечением ресурсов графического процессора. Не требуется модификация уже используемых средств разработки, нужно только настроить связь с препроцессором и компилятором от nVidia. Негативными сторонами являются необходимость наличия графического процессора именно от nVidia (что, впрочем, исправляется использованием обобщённой библиотеки OpenCL) и необходимость в копировании данных между адресными пространствами, а также сложность модели памяти на графическом ядре.

Система диспетчеризации StarPU

StarPU – система, позволяющая прозрачно для программиста задействовать средства центрального процессора и других вычислителей, маскируя низкоуровневые операции (например, пересылки между устройствами памяти) и автоматически распределяя задачи в гетерогенной среде [5]. Поддерживаются языки C, C++, SMP-процессоры архитектур x86, PowerPC и др., графические процессоры nVidia, устройства, совместимые с OpenCL, процессоры Cell, операционные системы Linux, Windows, Mac OS X. На момент написания статьи последней версией является 1.0.1 от мая 2012 года.

StarPU является библиотекой времени исполнения. Обращение к этой библиотеке возможно либо через API, либо с помощью директив для компилятора. На момент написания статьи под-

держка компиляторных директив была реализована только для GCC в виде расширения (plugin) и имела экспериментальный статус, поэтому в эксперименте используется API. StarPU реализует программную модель, ориентированную на задачи: программист производит декомпозицию алгоритма на отдельные участки, называемые задачами, реализует их средствами центрального или графического процессора, а StarPU выполняет диспетчеризацию этих задач, самостоятельно выполняя необходимые пересылки между устройствами памяти. В StarPU реализовано несколько алгоритмов диспетчеризации, по умолчанию используется жадный приоритетный диспетчер (eager), есть возможность реализации собственных диспетчеров.

С точки зрения программиста программа состоит из так называемых кодлетов (codelet) – вычислительных участков кода, которые могут быть исполнены на центральном, графическом процессоре или другом сопроцессоре. Задачей (task) является привязка кодлета к наборам данных, специфических для архитектуры, для которой реализован кодлет. Запуск новой задачи выполняется асинхронно, может быть определена функция отклика (callback). Также задача может содержать указания планировщику. Зависимости задач автоматически наследуются от зависимостей данных, однако программист может определить зависимости задач вручную. Пересылки данных между устройствами памяти выполняются по требованию, а не предварительно.

Таким образом, StarPU позволяет создавать программы, автоматически распределяемые между доступными исполнителями. Однако для достижения максимальной эффективности требуется реализовывать задачи для всех доступных архитектур.

Эксперимент

Будем решать задачу умножения двух квадратных матриц размером 1024×1024 действительных чисел типа float (размер матрицы выбран с целью кратности 2, тип данных выбран в связи с ограничениями имеющегося GPU). Решать будем «в лоб», без оптимизаций. Будем отслеживать время выполнения, использованные ресурсы (загрузка процессора, количество загруженных процессоров, объём потреблённой памяти) и энергопотребление, а также субъективно оценим трудозатраты на реализацию. Затраченное время, загрузка процессора и объём потреблённой памяти учитываются утилитой GNU time, мощность подсчитывается ваттметром производства НПИ «Учебная техника и технологии» ЮУрГУ с выходом USB. Одинаковость исходных данных обеспечивается одинаковостью начального зерна ГПСЧ. Время генерации исходных данных учитывается, но не влияет на соотношение результатов, так как генерация исходных данных всегда выполняется на центральном процессоре до начала работы вычислительных блоков программ.

Измерения проводились на двух платформах. Параметры первой платформы:

- процессор AMD Athlon 64 X2 Dual Core 4400+ 2.30 ГГц (2 ядра);
- 2 ГБ оперативной памяти DDR2 677 МГц, шина ОЗУ 128 бит;
- видеокарта nVidia GeForce 8600 GT, CUDA Compute 1.1;
- ОС CentOS 6.2 x86_64, linux 2.6.32-220.17.1.el6, GCC/libgomp 4.4.6 20110731, StarPU 1.0.1, CUDA Toolkit 4.2 V0.2.1221;
- мощность в простое 75 Вт.
- Параметры второй платформы:
- Intel Atom 330 1.60 ГГц (2 ядра), Hyper-Threading включен;
- 2 ГБ оперативной памяти DDR2 667 МГц, шина ОЗУ 64 бит;
- видеокарта nVidia ION (GeForce 9400M), CUDA Compute 1.1;
- ОС CentOS 6.2 x86_64, linux 2.6.32-220.17.1.el6, GCC/libgomp 4.4.6 20110731, StarPU 1.0.1, CUDA Toolkit 4.2 V0.2.1221;
- мощность в простое 35 Вт.

Каждая программа запускалась по 10 раз, в качестве результата бралось среднее арифметическое полученных данных. Между запусками очищался кэш ОЗУ и запускалась генерация случайных чисел для сброса кэша процессора. Компиляция CUDA-кода производилась для платформы 1.1. При использовании CUDA полученные показатели энергопотребления могут не соответствовать действительности, так как ваттметр может выдавать результат только раз в секунду. При измерении результатов для StarPU производилось два запуска: в одном запрещалось использование CUDA, в другом разрешалось. Перед запуском программы с использованием StarPU проводился пробный запуск (не учитывался), чтобы StarPU создал модель производительности для каждого вычислителя. (См. листинг 1–6).

Сырые результаты эксперимента и средства обработки доступны по адресу http://hc.comp.susu.ac.ru/omp_cuda_starpu/. Обработанные результаты сведены в таблицу.

Кратко сравним варианты реализации по трудозатратам:

- при использовании OpenMP модификации свелись к добавлению нескольких директив компилятору, все необходимые библиотеки уже распространяются с компилятором GCC;
- при использовании CUDA обязательно наличие графического процессора от nVidia не старше серии 8000, а также требуется изучение расширений языка и установки средств разработки;
- при использовании StarPU необходимо писать отдельные реализации для каждого используемого типа процессора и иметь соответствующие средства разработки, а также саму библиотеку StarPU.

Листинг 1. Решение задачи в один поток на центральном процессоре

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 1024

int main() {
    float *a, *b, *c, x;
    int i,j,r;

    a = (float *) malloc(SIZE*SIZE*sizeof(float));
    b = (float *) malloc(SIZE*SIZE*sizeof(float));
    c = (float *) malloc(SIZE*SIZE*sizeof(float));

    for (i=0;i<SIZE*SIZE;i++) {
        a[i] = random()/random();
        b[i] = random()/random();
        c[i] = 0;
    }

    for (i=0;i<SIZE;i++)
        for (r=0;r<SIZE;r++) {
            x = 0;
            for (j=0;j<SIZE;j++)
                x += a[SIZE*i+j]*b[r+SIZE*j];
            c[SIZE*i+r] = x;
        }

    printf ("[ %f, %f, ...\n", c[0], c[1]);
    printf ("  %f, %f, ... ]\n", c[SIZE], c[SIZE+1]);

    free(a);
    free(b);
    free(c);

    return 0;
}
```

Листинг 2. Решение задачи в несколько (по числу процессорных ядер) потоков на центральном процессоре

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define SIZE 1024

int main() {
    float *a, *b, *c, x;
    int i,j,r;

    a = (float *) malloc(SIZE*SIZE*sizeof(float));
    b = (float *) malloc(SIZE*SIZE*sizeof(float));
    c = (float *) malloc(SIZE*SIZE*sizeof(float));

    for (i=0;i<SIZE*SIZE;i++) {
        a[i] = random()/random();
        b[i] = random()/random();
        c[i] = 0;
    }

    #pragma omp parallel shared(a,b,c) private(x,i,j,r)
    {
        #pragma omp for schedule (static)
        for (i=0;i<SIZE;i++)
            for (r=0;r<SIZE;r++) {
                x = 0;
                for (j=0;j<SIZE;j++)
                    x += a[SIZE*i+j]*b[r+SIZE*j];
                c[SIZE*i+r] = x;
            }
    }
```

```
    printf ("[ %f, %f, ...\n", c[0], c[1]);
    printf ("  %f, %f, ... ]\n", c[SIZE], c[SIZE+1]);

    free(a);
    free(b);
    free(c);

    return 0;
}
```

Листинг 3. Решение задачи в несколько (по одному на ячейку таблицы) потоков только на графическом процессоре

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 1024
#define BLOCK_SIZE 16

__global__ void multiply(float *a, float *b, float *c) {
    int elem_x = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    int elem_y = blockIdx.y * BLOCK_SIZE + threadIdx.y;
    int i = 0;
    float x = 0;

    for (i=0; i<SIZE; i++) x += a[elem_y*SIZE + i] * b[elem_x + i*SIZE];
    c[elem_y*SIZE + elem_x] = x;
}

int main() {
    float *a, *b, *c;
    float *ca, *cb, *cc;
    int i;

    a = (float *) malloc(SIZE*SIZE*sizeof(float));
    b = (float *) malloc(SIZE*SIZE*sizeof(float));
    c = (float *) malloc(SIZE*SIZE*sizeof(float));

    for (i=0;i<SIZE*SIZE;i++) {
        a[i] = random()/random();
        b[i] = random()/random();
        c[i] = 0;
    }

    cudaMalloc(&ca, SIZE*SIZE*sizeof(float));
    cudaMemcpy(ca, a, SIZE*SIZE*sizeof(float), cudaMemcpyHostToDevice);
    cudaMalloc(&cb, SIZE*SIZE*sizeof(float));
    cudaMemcpy(cb, b, SIZE*SIZE*sizeof(float), cudaMemcpyHostToDevice);
    cudaMalloc(&cc, SIZE*SIZE*sizeof(float));

    dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
    dim3 grid(SIZE / threads.x, SIZE / threads.y);
    multiply<<<grid,threads>>> (ca, cb, cc);

    cudaMemcpy(c, cc, SIZE*SIZE*sizeof(float), cudaMemcpyDeviceToHost);

    printf ("[ %f, %f, ...\n", c[0], c[1]);
    printf ("  %f, %f, ... ]\n", c[SIZE], c[SIZE+1]);

    cudaFree(ca);
    cudaFree(cb);
    cudaFree(cc);

    free(a);
    free(b);
    free(c);

    return 0;
}
```

Листинг 4. Решение задачи с применением StarPU

```
#include <stdio.h>
#include <stdlib.h>
#include <starpu.h>

#define SIZE 1024

extern void multiply_cpu(void *buffers[], void *args);
extern void multiply_cuda(void *buffers[], void *args);

struct starpu_codelet cl = {
    .where = STARPU_CPU | STARPU_CUDA,
    .cpu_funcs = { multiply_cpu, NULL },
    .cuda_funcs = { multiply_cuda, NULL },
    .nbuffers = 3,
    .modes = { STARPU_R, STARPU_R, STARPU_RW }
};

int main() {
    float *a, *b, *c;
    int i;

    a = (float *) malloc(SIZE*SIZE*sizeof(float));
    b = (float *) malloc(SIZE*SIZE*sizeof(float));
    c = (float *) malloc(SIZE*SIZE*sizeof(float));

    for (i=0;i<SIZE*SIZE;i++) {
        a[i] = random()/random();
        b[i] = random()/random();
        c[i] = 0;
    }

    starpu_init(NULL);

    starpu_data_handle_t a_handle;
    starpu_vector_data_register(&a_handle, 0, (uintptr_t)a, SIZE*SIZE, sizeof(float));
    starpu_data_handle_t b_handle;
    starpu_vector_data_register(&b_handle, 0, (uintptr_t)b, SIZE*SIZE, sizeof(float));
    starpu_data_handle_t c_handle;
    starpu_vector_data_register(&c_handle, 0, (uintptr_t)c, SIZE*SIZE, sizeof(float));

    struct starpu_task *task = starpu_task_create();
    task->synchronous = 1;
    task->cl = &cl;
    task->handles[0] = a_handle;
    task->handles[1] = b_handle;
    task->handles[2] = c_handle;

    starpu_task_submit(task);
    starpu_data_unregister(a_handle);
    starpu_data_unregister(b_handle);
    starpu_data_unregister(c_handle);
    starpu_shutdown();

    printf ("[ %f, %f, ...\n", c[0], c[1]);
    printf ("  %f, %f, ... ]\n", c[SIZE], c[SIZE+1]);

    free(a);
    free(b);
    free(c);

    return 0;
}
```

Листинг 5. Функция, решающая задачу на CPU для кодлета StarPU из листинга 4

```
#include <starpu.h>

#define SIZE 1024

int multiply_cpu(void *buffers[], void *cl_arg) {

    int i,j,r;
    float x;

    struct starpu_vector_interface *a_handle = (struct starpu_vector_interface
*) buffers[0];
    struct starpu_vector_interface *b_handle = (struct starpu_vector_interface
*) buffers[1];
    struct starpu_vector_interface *c_handle = (struct starpu_vector_interface
*) buffers[2];

    float *a = (float*) STARPU_VECTOR_GET_PTR(a_handle);
    float *b = (float*) STARPU_VECTOR_GET_PTR(b_handle);
    float *c = (float*) STARPU_VECTOR_GET_PTR(c_handle);

    for (i=0;i<SIZE;i++)
        for (r=0;r<SIZE;r++) {
            x = 0;
            for (j=0;j<SIZE;j++)
                x += a[SIZE*i+j]*b[r+SIZE*j];
            c[SIZE*i+r] = x;
        }

    return 0;
}
```

Листинг 6. Функция, решающая задачу на GPU для кодлета StarPU из листинга 4

```
#include <starpu.h>
#include <starpu_cuda.h>

#define SIZE 1024
#define BLOCK_SIZE 16

__global__ void multiply(float *a, float *b, float *c) {
    int elem_x = blockIdx.x * BLOCK_SIZE + threadIdx.x;
    int elem_y = blockIdx.y * BLOCK_SIZE + threadIdx.y;
    int i = 0;
    float x = 0;

    for (i=0; i<SIZE; i++) x += a[elem_y*SIZE + i] * b[elem_x + i*SIZE];
    c[elem_y*SIZE + elem_x] = x;
}

extern "C" void multiply_cuda(void *buffers[], void *args) {

    struct starpu_vector_interface *a_handle = (struct starpu_vector_interface
*) buffers[0];
    struct starpu_vector_interface *b_handle = (struct starpu_vector_interface
*) buffers[1];
    struct starpu_vector_interface *c_handle = (struct starpu_vector_interface
*) buffers[2];

    float *ca = (float*) STARPU_VECTOR_GET_PTR(a_handle);
    float *cb = (float*) STARPU_VECTOR_GET_PTR(b_handle);
    float *cc = (float*) STARPU_VECTOR_GET_PTR(c_handle);

    dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
    dim3 grid(SIZE / threads.x, SIZE / threads.y);
    multiply<<<grid,threads,0,starpu_cuda_get_local_stream()>>> (ca, cb, cc);

    cudaStreamSynchronize(starpu_cuda_get_local_stream());
}
```

Результаты экспериментов

	Реализация	Время, с	Загрузка процессора, %	Занятая оперативная память, КБ	Потреблённая мощность, Вт	Энергопотребление, (кВт·ч)·10^3
Платформа 1	CPU	34,08	99,00	51 083,20	3 858,08	1,072
	OpenMP	25,16	197,60	53 316,80	3 112,35	0,865
	CUDA	2,80	81,40	119 265,60	289,33	0,080
	StarPU/CPU	35,38	193,80	135 208,00	5 214,74	1,449
	StarPU/CUDA	11,13	146,70	243 329,60	1 310,20	0,364
Платформа 2	CPU	157,33	99,00	51 196,80	5 449,58	1,514
	OpenMP	51,06	394,80	55 180,80	1 894,73	0,526
	CUDA	3,97	59,20	102891,2	189,56	0,053
	StarPU/CPU	155,76	196	121644,8	5225,45	1,452
	StarPU/CUDA	*	*	*	*	*

* При исполнении программы в варианте с использованием StarPU и задействованием CUDA на второй платформе среда StarPU для выполнения кодлета периодически выбирала центральный процессор вместо графического по неизвестным причинам. Возможно, это связано с программными ошибками в StarPU.

Сделаем выводы по полученным результатам эксперимента:

- применение CUDA позволило примерно в 12 раз для первой и в 40 раз для второй платформы сократить время выполнения программы, при этом примерно в 13 раз для первой и в 30 раз для второй платформы снизилось энергопотребление;
- применение OpenMP даёт экономию времени и электроэнергии пропорционально количеству ядер;
- применение StarPU неоднозначно: с одной стороны, если код выполняется на центральном процессоре, то это не даёт выигрыша, так как одно процессорное ядро отдаётся под работу самого StarPU, если задача не разбивается на несколько независимых подзадач, то выгоднее использовать код только для графического процессора, но с другой стороны, если задача разделяется на несколько независимых подзадач, то StarPU может дать выигрыш засчёт одновременного задействования всех доступных процессоров. Вероятно, из-за недостаточной отлаженности алгоритмов диспетчеризации StarPU не всегда корректно выбирает тип процессора для выполнения кода. Эффект от StarPU требует дополнительного изучения на других задачах и платформах.

Выводы

Дадим следующие рекомендации:

- если платформа не имеет графического процессора с поддержкой ускорения вычислений (nVidia CUDA / AMD Fusion / OpenCL), но имеет многоядерный (или несколько) процессор, то применение OpenMP – простой способ добиться повышения производительности и снижения энергозатрат пропорционально количеству процессоров / ядер;
- если платформа имеет графический процессор, задача является SIMD-реализуемой, но представляет собой набор последовательных участков параллельного SIMD-кода, то применение графического процессора даёт увеличение производительности и снижение энергозатрат на порядок (в литературе описаны случаи прироста производительности на несколько порядков), однако потребует значительной модификации кода;
- если алгоритм можно разложить на несколько участков, не зависящих друг от друга, то применение StarPU позволяет получить прирост производительности и снижение энергозатрат, причём можно разделить последовательный код (отдав его выполнение центральному процессору) и параллельный код (перенеся его на графический процессор), однако придётся включать в код обращение к StarPU и реализовывать кодлеты для графического процессора.

Литература

1. OpenMP Application Program Interface. Version 3.1 July 2011. – http://www.openmp.org/mp-documents/OpenMP3.1.pdf

2. What is CUDA. – http://developer.nvidia.com/what-cuda

3. Tianhe-1A. – http://i.top500.org/system/176929

4. K computer. – http://i.top500.org/system/177232

5. StarPU. A Unified Runtime System for Heterogeneous Multicore Architectures. – http://runtime.bordeaux.inria.fr/StarPU/

Поступила в редакцию 3 декабря 2012 г.