GPU programming with OpenACC

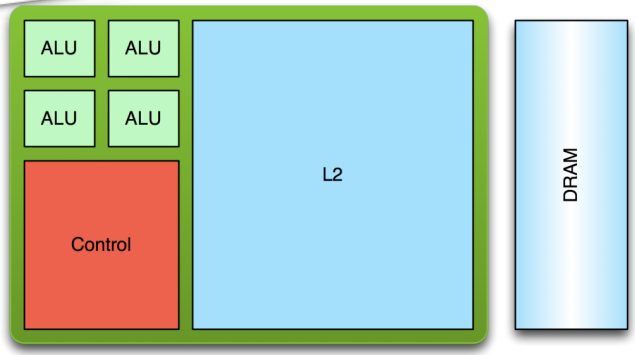
Nguyen Minh Duc

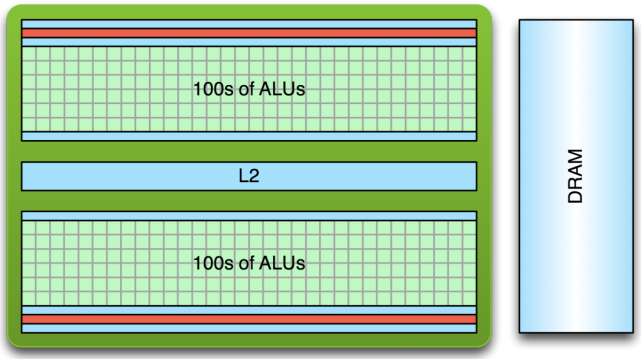
Contents

- Архитектура GPU
- Модель программирования CUDA
- OpenACC
- OpenACC API
- Примеры
- Ресурсы

Архитектура GPU

CPU vs. GPU

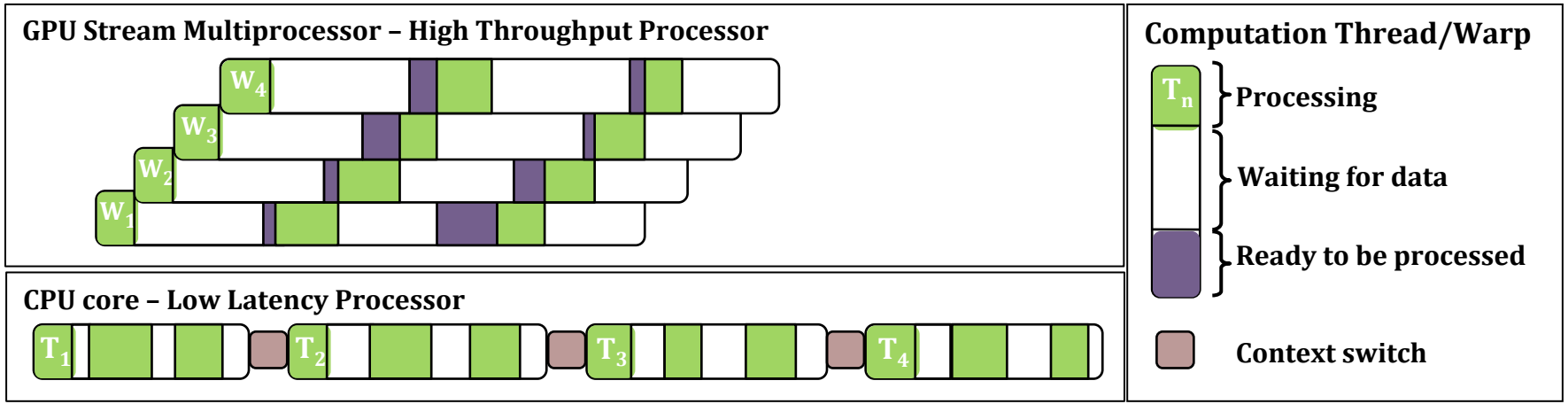




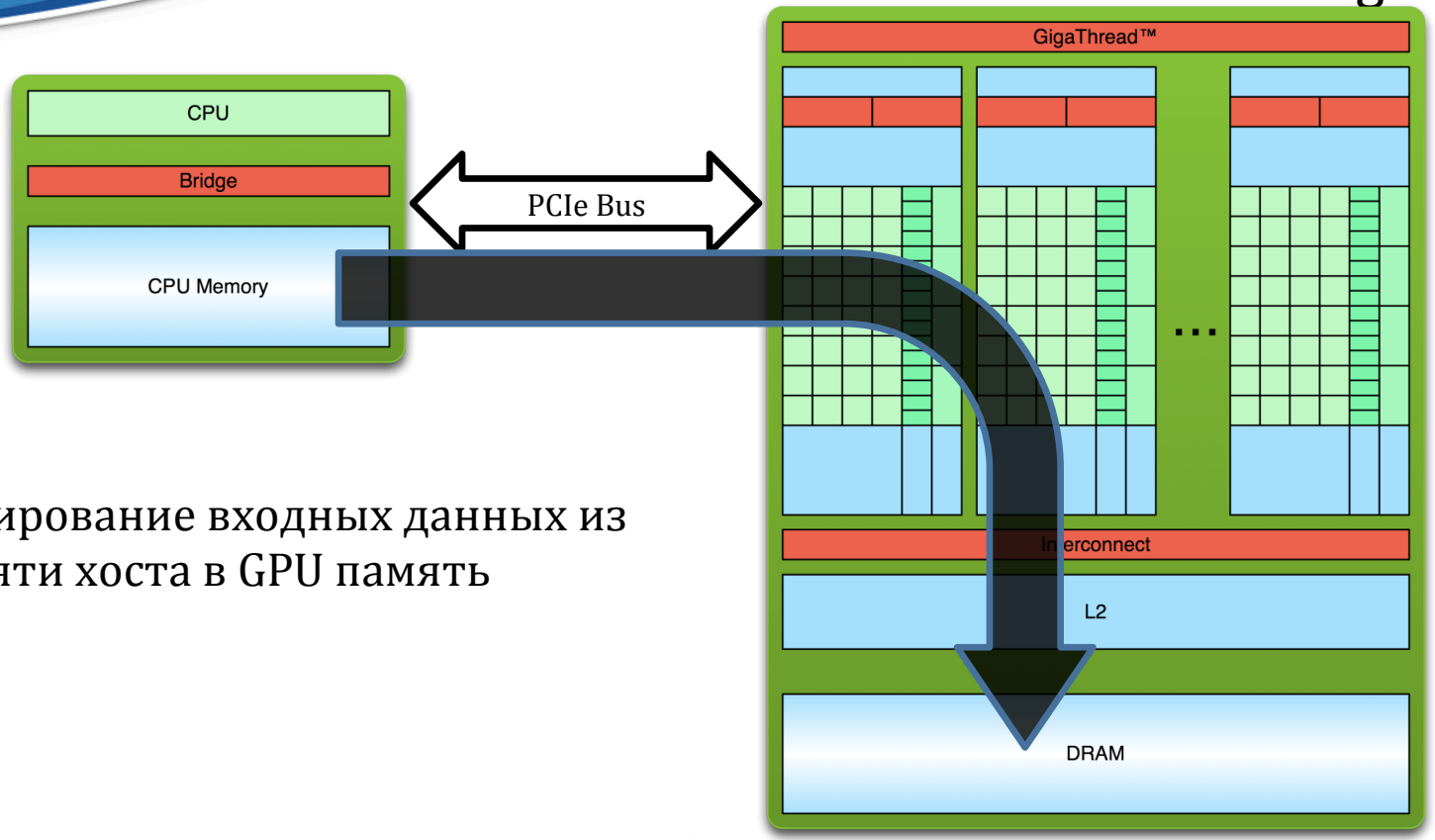
- **CPU**
  - Оптимизирован для доступа с малой задержкой к кэшированным данным
  - Логика управления и внепорядковое выполнение
- **GPU**
  - Больше транзисторов выделено на обработку данных
  - Оптимизирован для параллельного вычисления
  - Чувствителен к задержке доступа к памяти

Low Latency or High Throughput?

- **CPU** архитектура должна минимизировать задержку в каждой нити
- **GPU - скрывает задержку** за счет вычисления из других warps

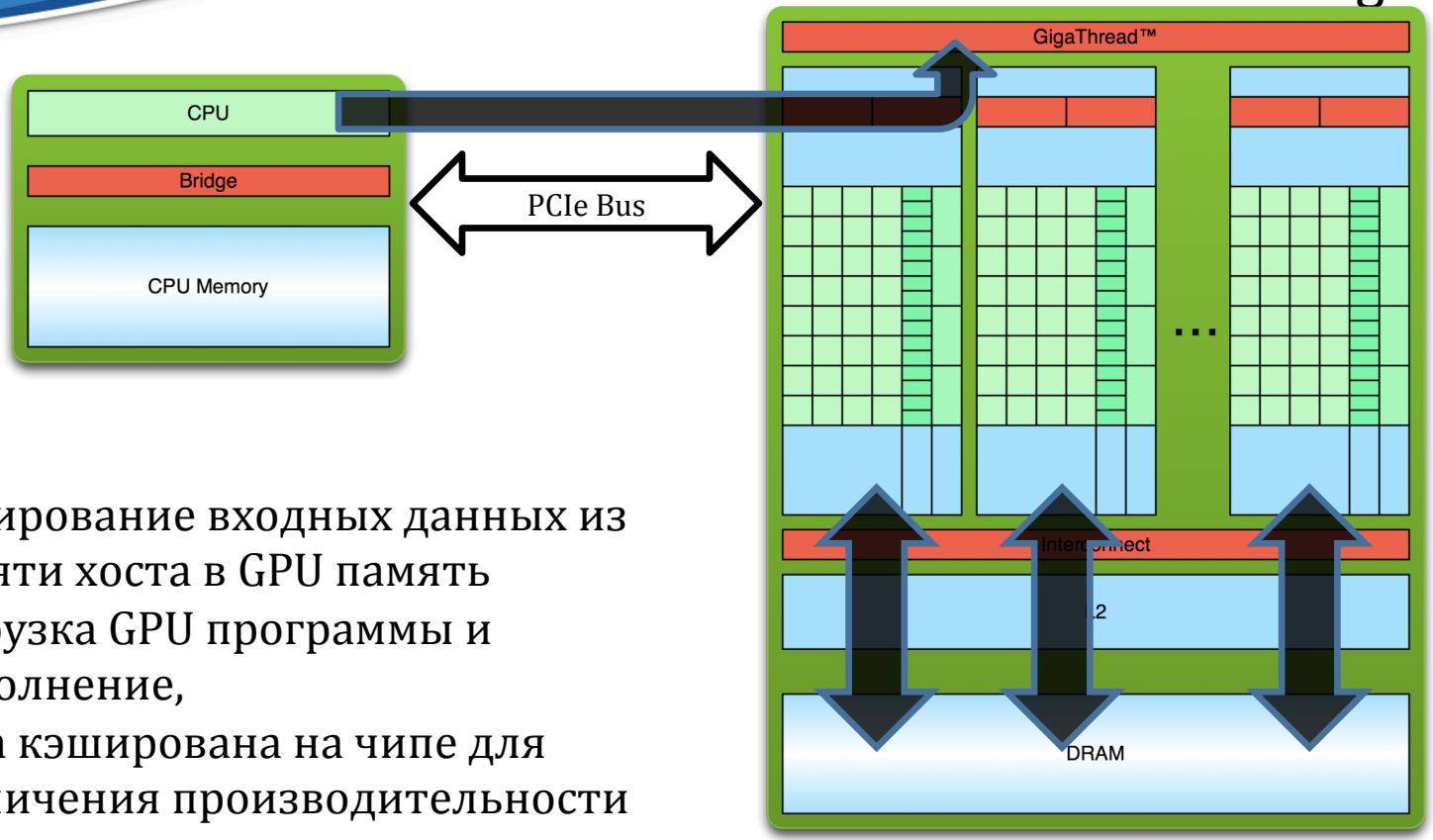


Processing Flow



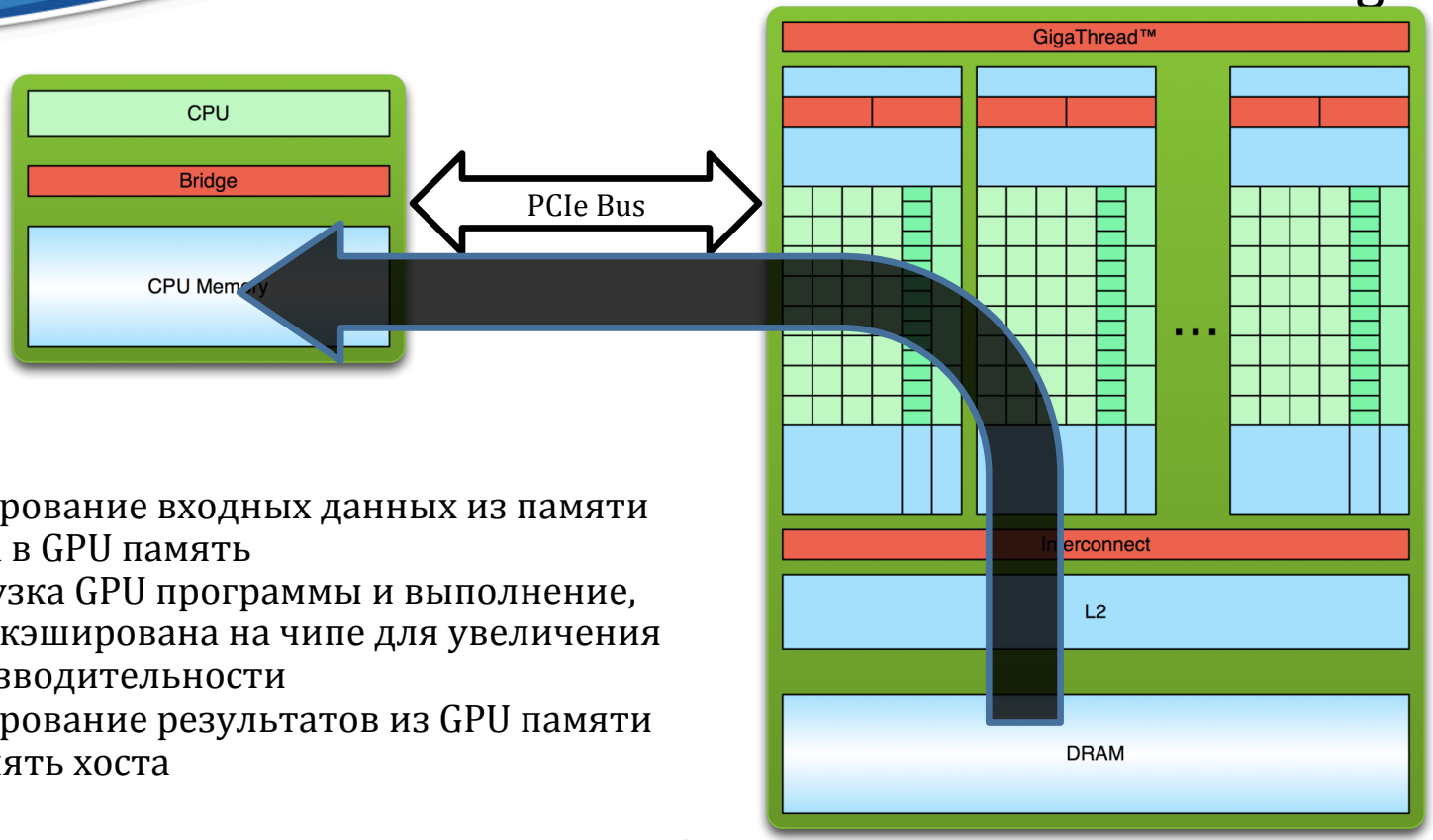
1. Копирование входных данных из памяти хоста в GPU память

Processing Flow



1. Копирование входных данных из памяти хоста в GPU память
2. Загрузка GPU программы и выполнение,
   Дата кэширована на чипе для увеличения производительности

Processing Flow

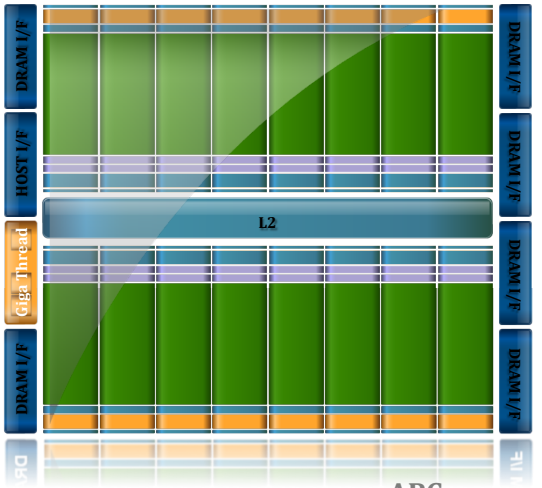


1. Копирование входных данных из памяти хоста в GPU память
2. Загрузка GPU программы и выполнение, Дата кэширована на чипе для увеличения производительности
3. Копирование результатов из GPU памяти в память хоста

GPU Architecture: Two Main Components

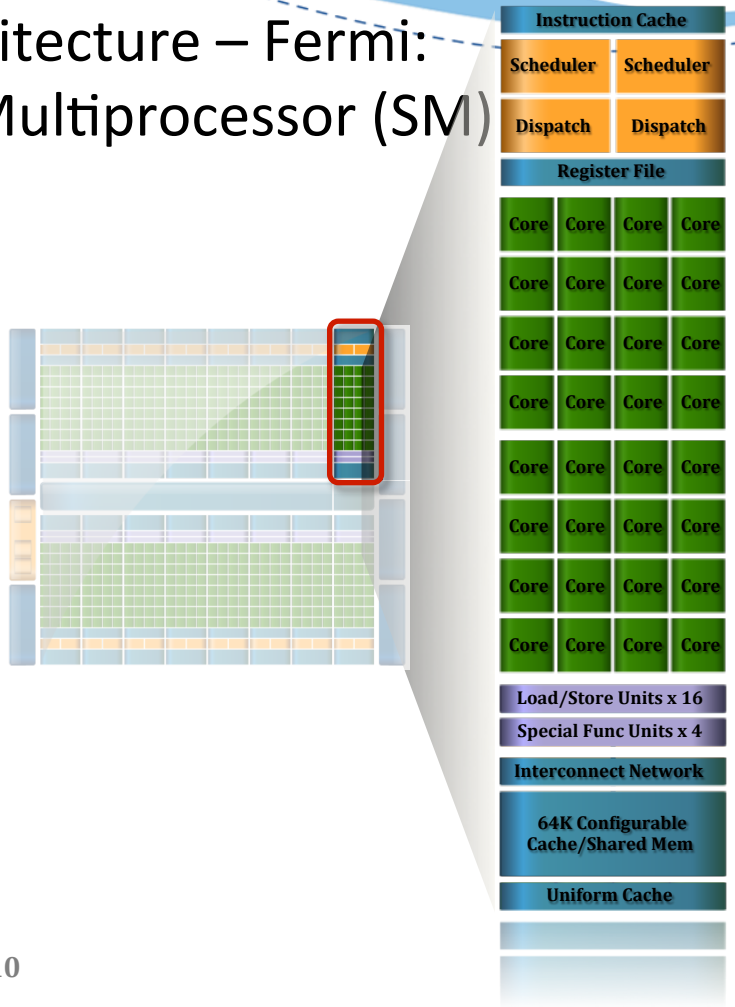
- Глобальная Память
  - Подобна памяти на хосте
  - Доступна и GPU и CPU
  - Текущий максимальный объем - 6 GB
  - Текущая пропускная способность - 150 GB/s на Quadro и Tesla моделях
  - ECC on/off опция для Quadro и Tesla моделей

- Потоковые Мультипроцессоры (SMs)
  - Осуществляют фактическое вычисление
  - Каждый SM имеет свои:
    - Блоки управления, регисторы, конвейерное выполнение, кэш



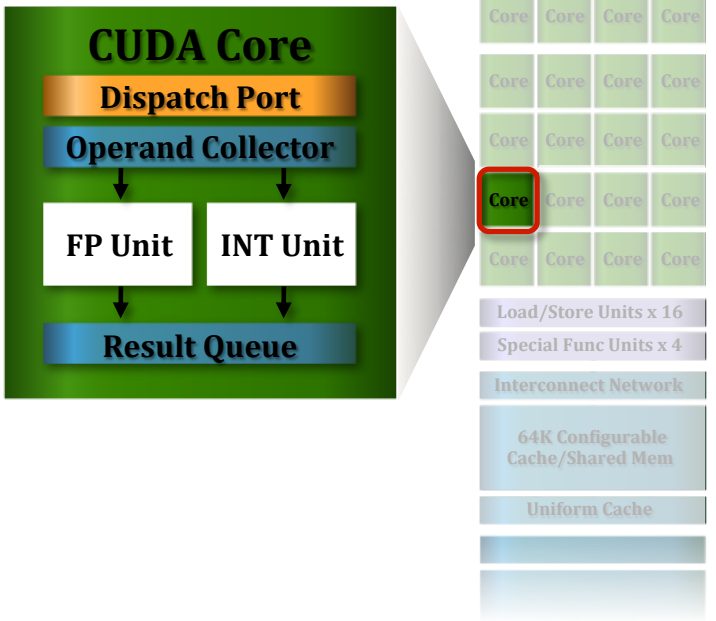
GPU Architecture – Fermi: Streaming Multiprocessor (SM)

- 32 CUDA Cores per SM
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 warp schedulers
  - Up to 1536 threads concurrently
- 4 special-function units
- 64KB shared memory + L1 cache
- 32K 32-bit registers



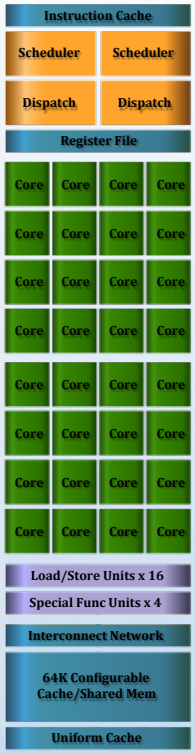
GPU Architecture – Fermi: CUDA Core

- Floating point & Integer unit
  - IEEE 754-2008 floating-point standard
  - Fused multiply-add (FMA) instruction for both single and double precision
- Logic unit
- Move, compare unit
- Branch unit

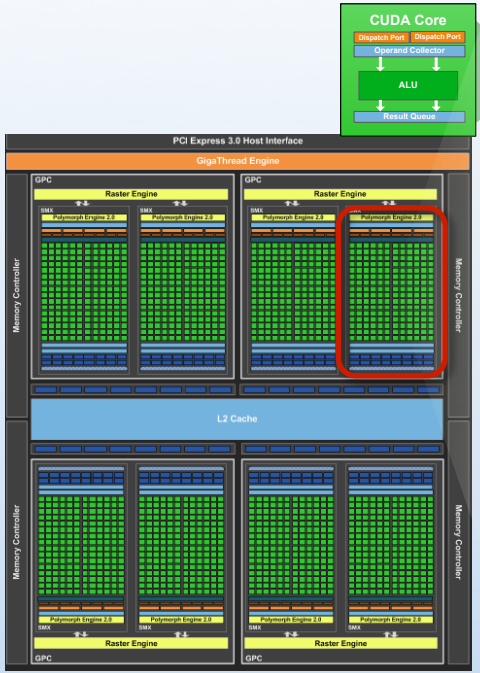


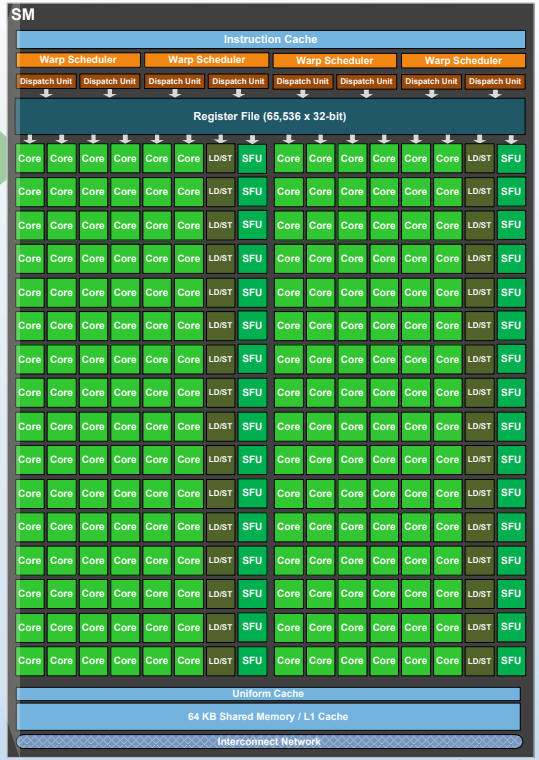
Kepler

Fermi

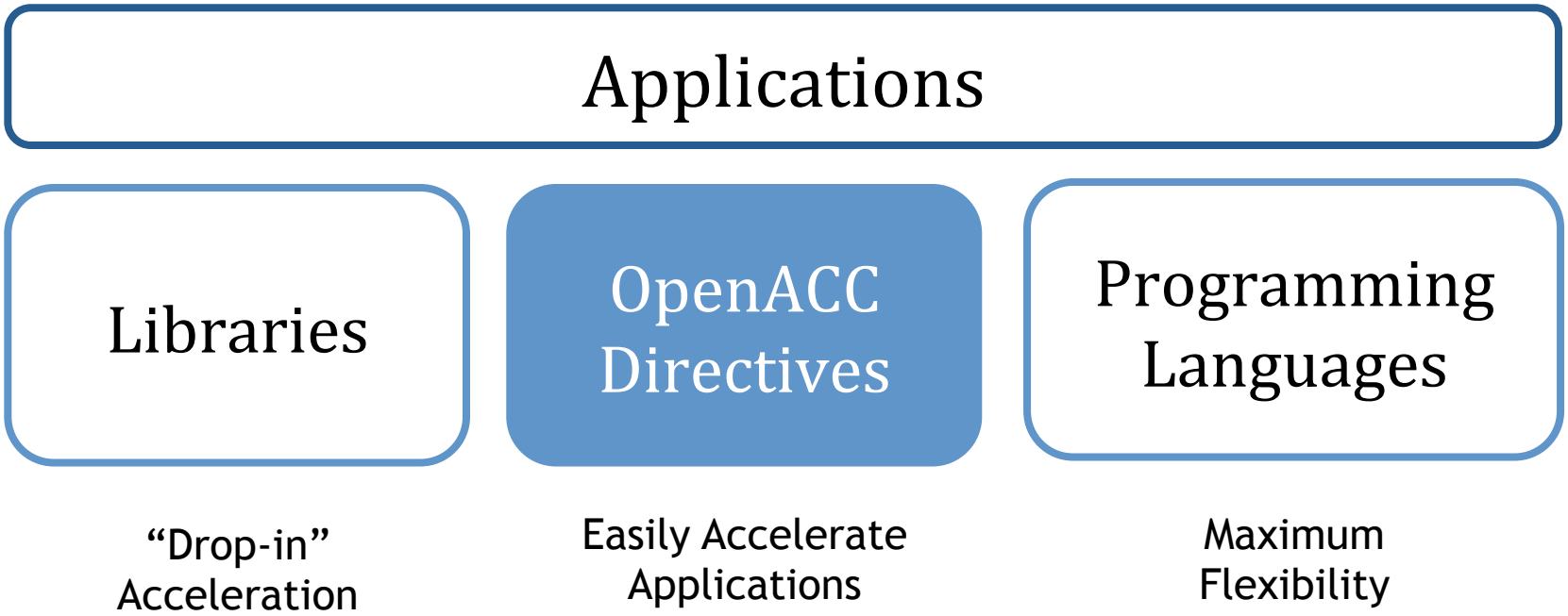


Kepler





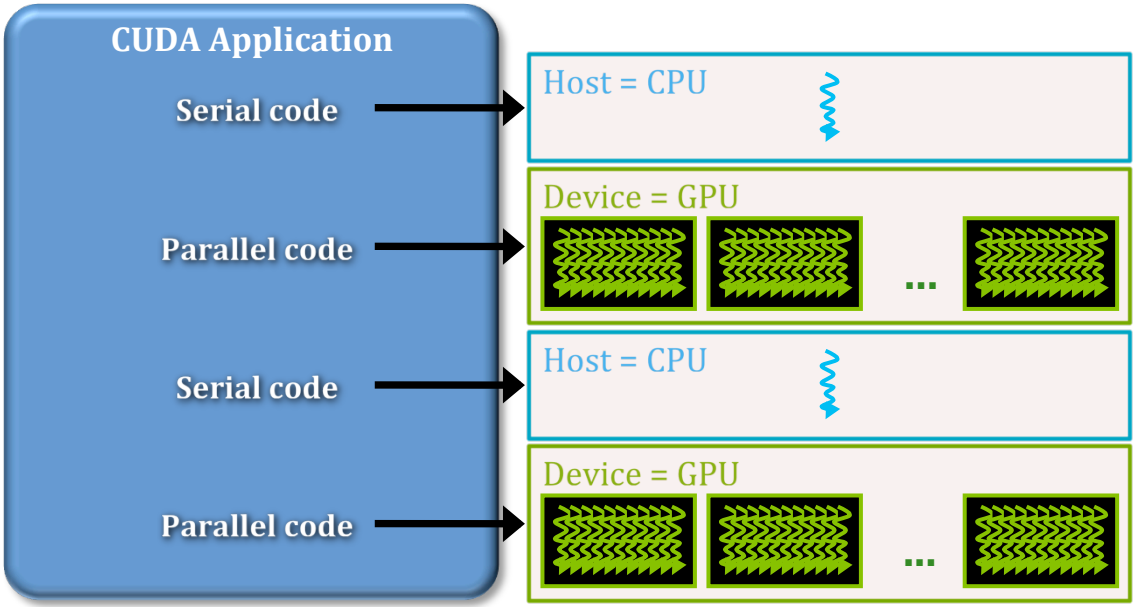
3 Ways to Accelerate Applications



Модель программирования CUDA

Anatomy of a CUDA Application

- Последовательный код выполняет в одна Host (CPU) нить
- Параллельный код выполняют множество Device (GPU) нитей на разных вычислительных элементах



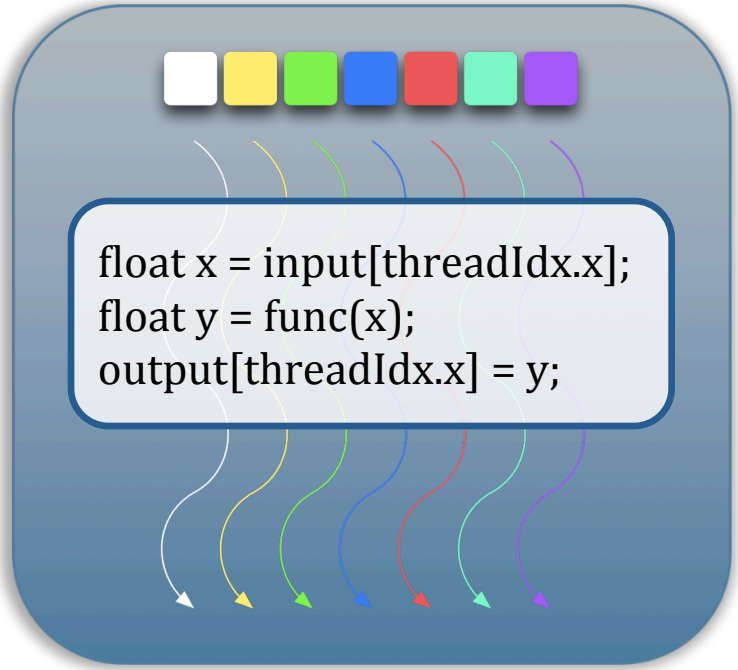
CUDA Kernels

- Параллельная часть приложения выполняется как ядро
  - Целый GPU выполняет ядро во многих нитях
- CUDA нити:
  - Легковесные
  - Быстрое переключение
  - 1000s выполняются одновременно

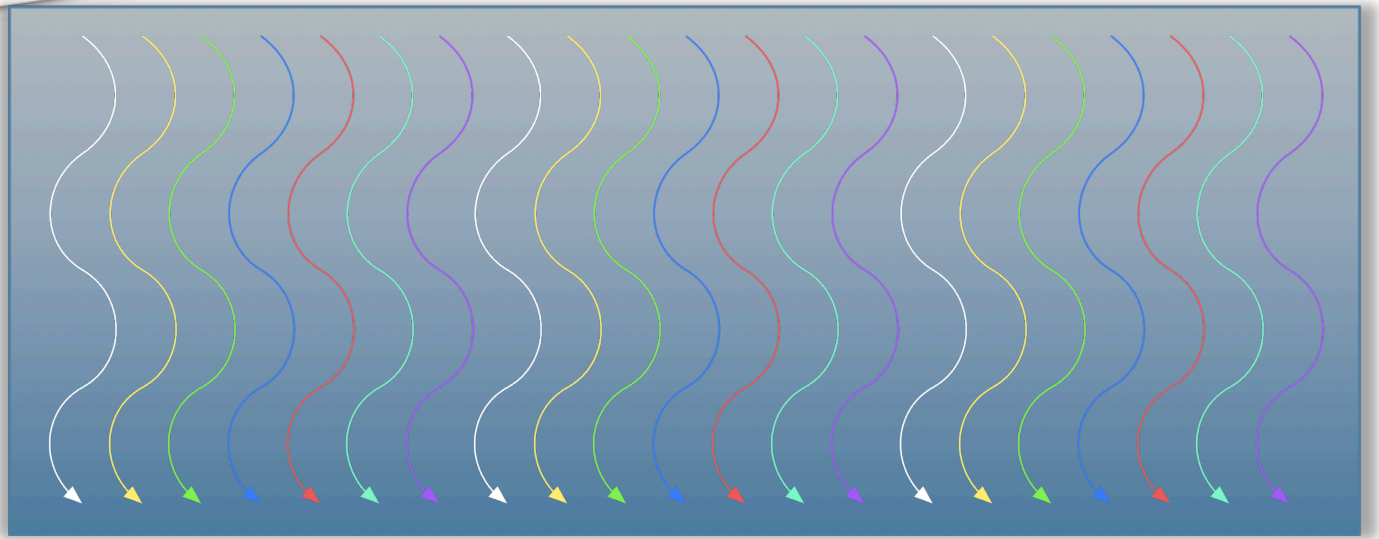
CPU	Host	выполняет функции
GPU	Device	выполняет ядра

CUDA Kernels: Parallel Threads

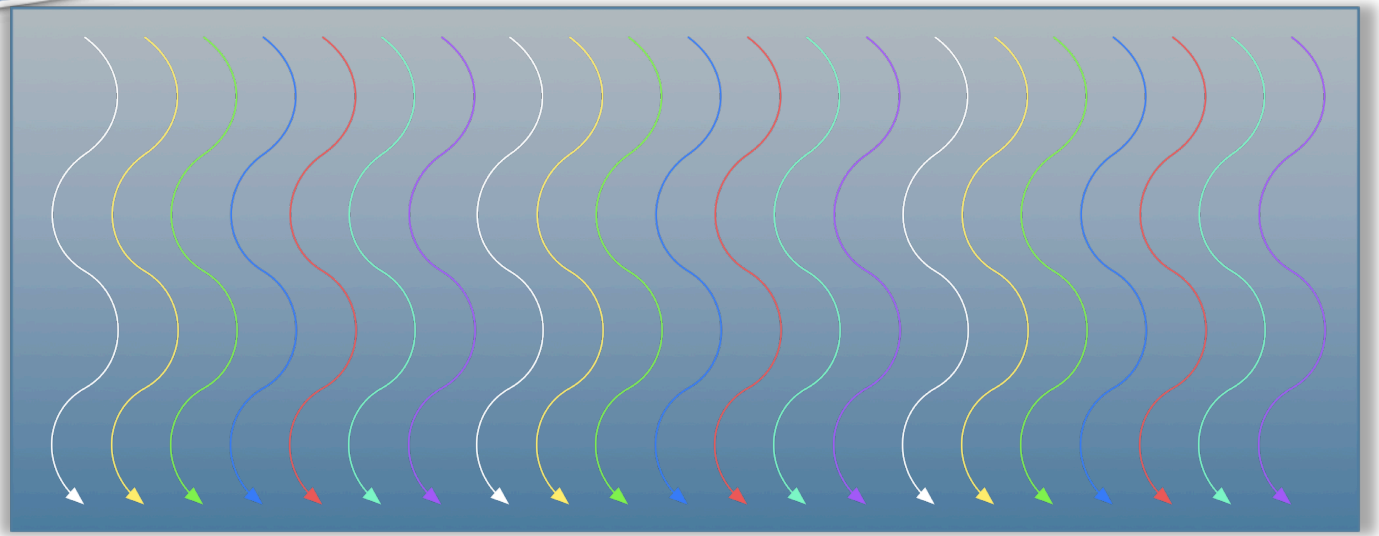
- Ядро - это функция, выполняющаяся на GPU как параллельный массив нитей
- Все нити выполняют один и тот же код, но может иметь разные ветви
- Каждая нить имеет свой ID
  - Свои входные/выходные данные
  - Принятие решений



CUDA Kernels: Subdivide into Blocks

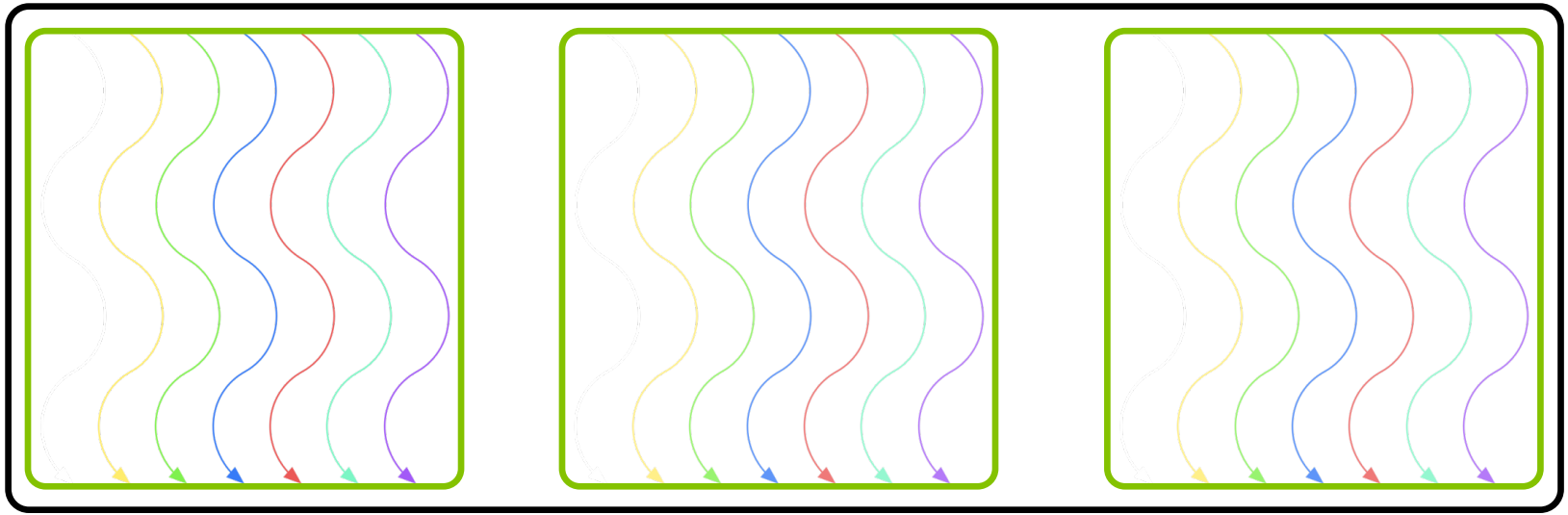


CUDA Kernels: Subdivide into Blocks



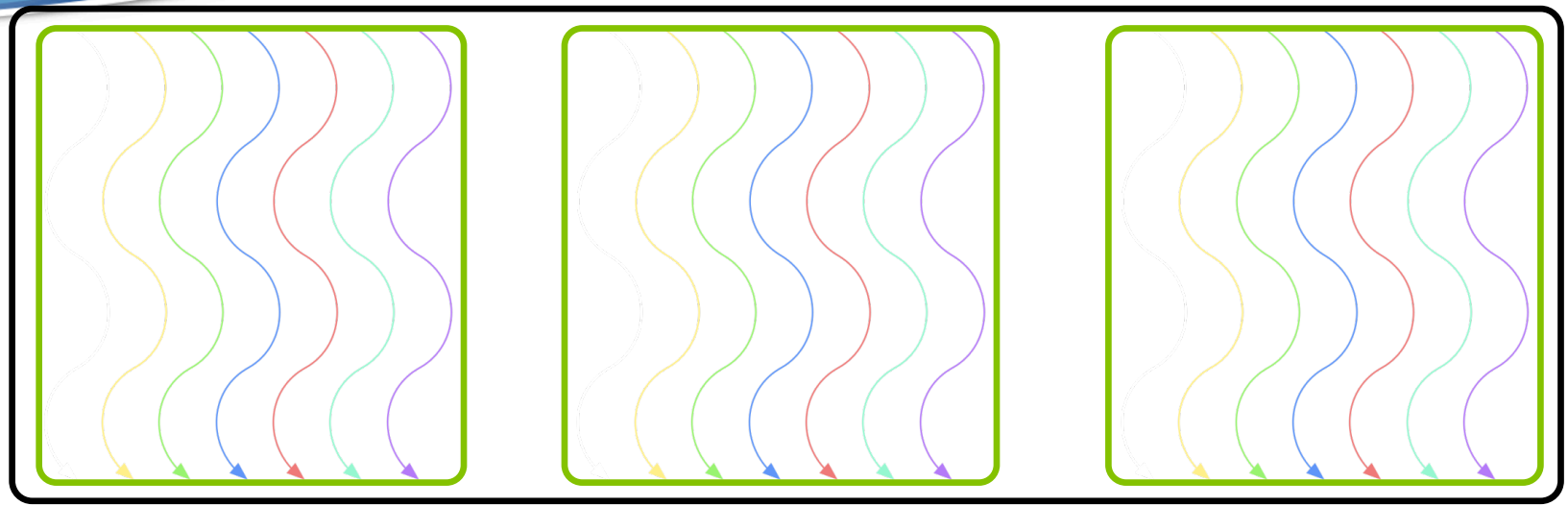
- Нити группированы в блоки

CUDA Kernels: Subdivide into Blocks



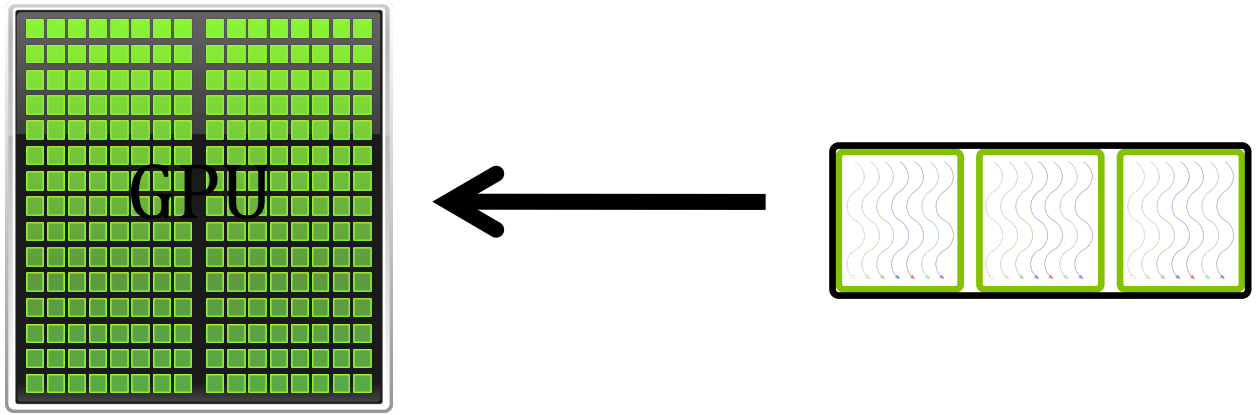
- Нити группированы в блоки
- Блоки группированы в одну решетку

CUDA Kernels: Subdivide into Blocks



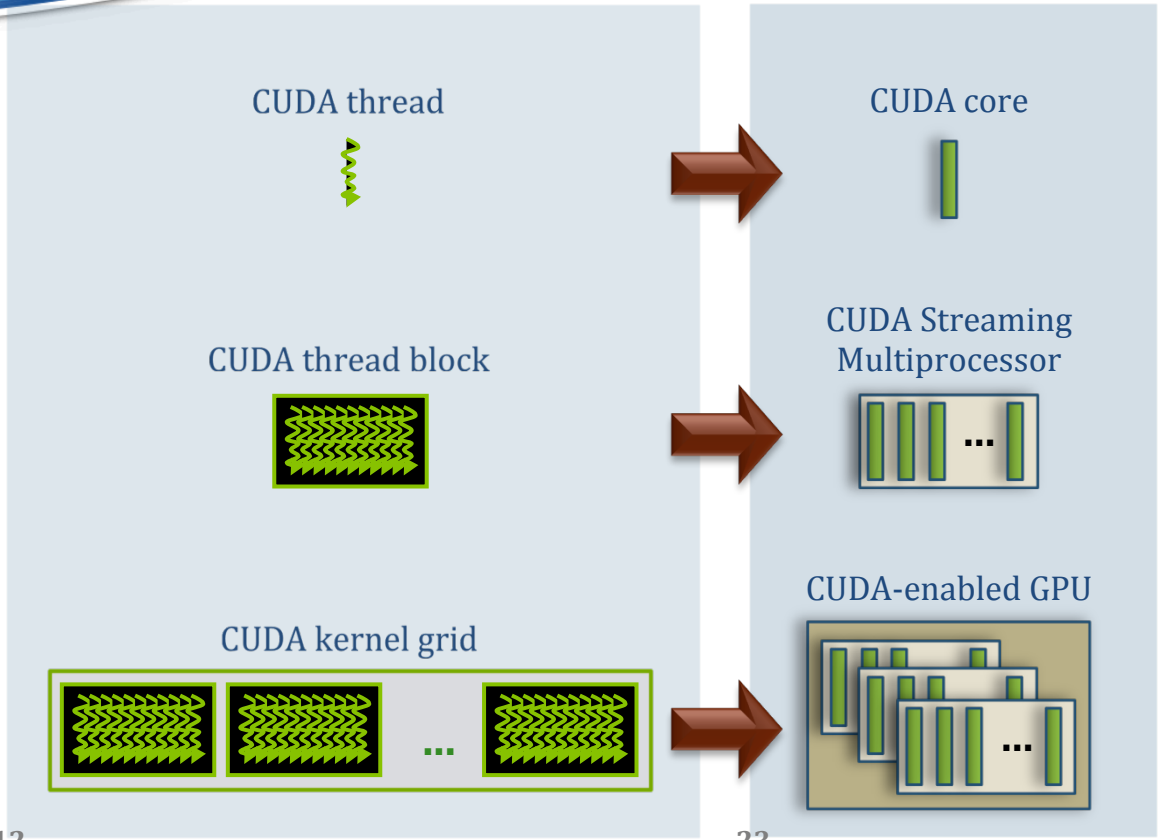
- Нити группированы в блоки
- Блоки группированы в одну решетку
- Ядро выполняется как решетка из блоков из нитей

CUDA Kernels: Subdivide into Blocks



- Нити группированы в блоки
- Блоки группированы в одну решетку
- Ядро выполняется как решетка из блоков из нитей

Kernel Execution



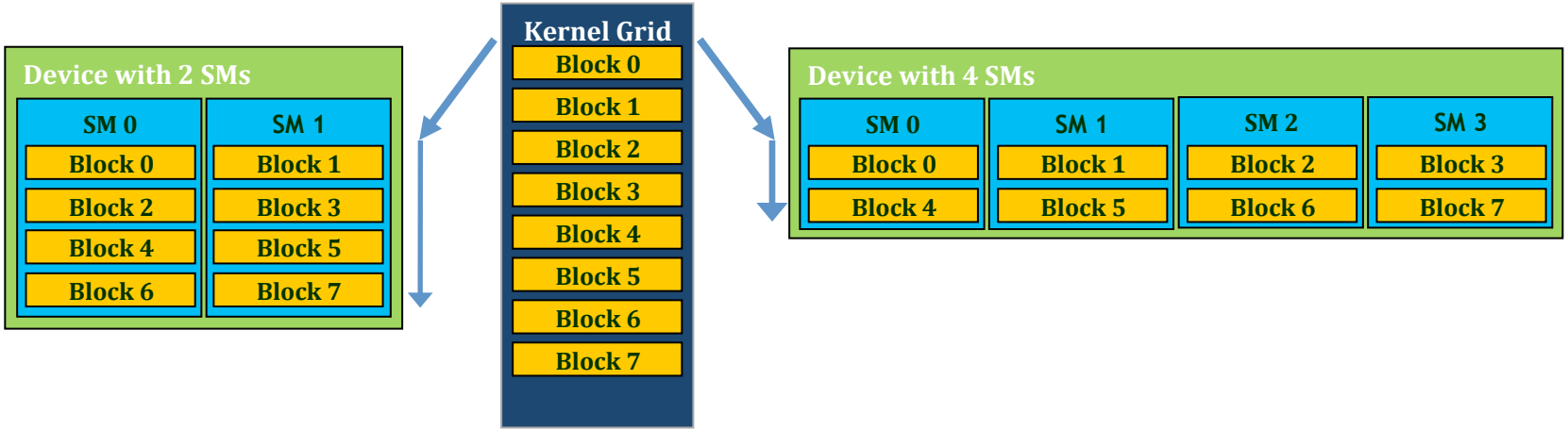
- Каждая нить выполняется на одном физ. ядре
- Каждый блок выполняется на одном SM и не мигрируется на другой
- Несколько конкурирующих блоков может находиться на одном SM в зависимости от требуемой памяти этих блоков и объема ресурсов этого SM
- Каждое ядро выполняется на одном GPU
- Множество ядер может выполняться на одном GPU одновременно

Thread blocks allow cooperation

- Нити могут взаимодействовать друг с другом:
  - Совместно грузить/хранить блоки памяти, используемые ими
  - Разделять результаты между собой или кооперироваться, для получения единого результата
  - Синхронизироваться с другими

Thread blocks allow scalability

- Блоки могут выполняться в любом порядке, параллельно или последовательно
- Независимость и масштабируемость: Ядро может иметь разные отображения на разные GPU

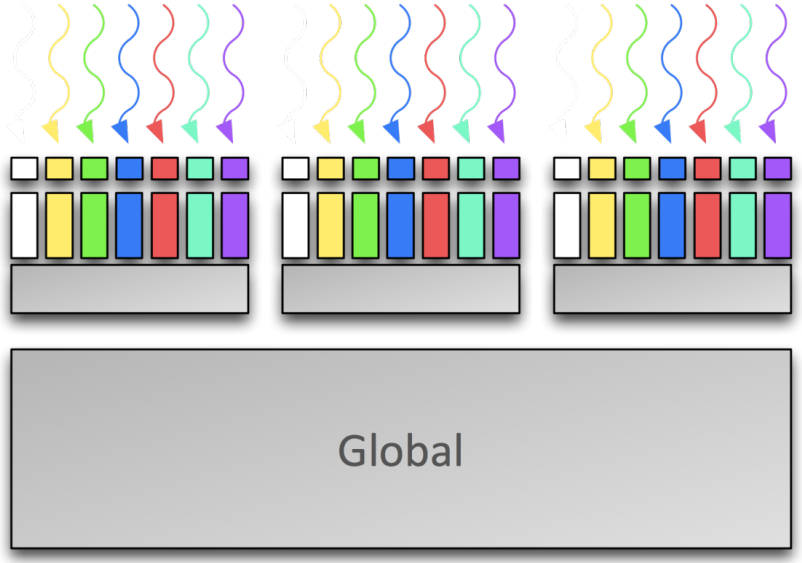


Warps

- Блоки делятся на так называемые warp-ы из 32 нитей
  - Размер warp-а зависит от реализации и может измениться в будущем
- SM создает, управляет, планирует и выполняет нити на уровне warp-ов
  - Каждый warp состоит из 32 последовательных нитей (thread IDs)
- Все нити в одном warp выполняют один и тот же код
  - Если нити warp разветвляются, warp последовательно выполняет каждую ветвь
- Когда warp выполняет код, который обращается к глобальной памяти, он по возможности объединяет (coalesce) обращения к памяти из разных нитей в пределе warp-а в одну или несколько транзакций

Memory hierarchy

- Нить:
  - Регистры
  - Локальная память
- Блок из нитей:
  - Разделяемая память
- Все блоки
  - Глобальная память



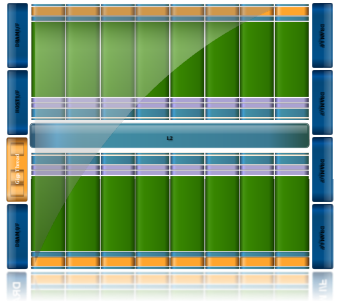

OpenACC®
DIRECTIVES FOR ACCELERATORS

SAXPY Example in C

- Легкость
- Открытость
- Производительность

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc kernels
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```



OpenACC

- **Набор директив для разметки параллельных областей** в коде на C, C++, Fortran
  - отображение параллельных областей из CPU на GPU
  - поддержка различных ОС, CPU, GPU, и компиляторов
- Создание высокоуровневых **гибридных (CPU+GPU) программ**
  - без явных инициализаций GPU
  - без явных передачи и синхронизации данных между CPU и GPU

OpenACC

- **Модель программирования** позволяет программистам начать разработку параллельных программ на GPU с *легкостью*, подсказывая компилятору с помощью директив:
  - спецификацию данных на GPU
  - отображение циклов на GPU
  - различные опции оптимизации
- Совместим с другими языками программирования на GPU и различными библиотеками
  - Взаимодействие с CUDA C/Fortran и GPU библиотеками
  - e.g. CUFFT, CUBLAS, CUSPARSE, etc.

Specification v1.0

- Полная спецификация OpenACC версия 1.0
  http://www.openacc-standard.org
- Быстрое руководство пользователя
- Бесплатная пробная версия компилятора
  http://www.nvidia.com/object/openacc-gpu-directives.html
- Онлайн курсы и вебинары
  http://www.nvidia.com/object/webinar.html

OpenACC Execution Model

- Host
  - выполняет большую часть кода;
  - выделяет память на GPU;
  - управляет передачей данных и кода на GPU;
  - управляет запуском и синхронизацией;
  - загружает результаты из памяти GPU;
  - высвобождает память на GPU.
- GPU
  - выполняет ядра одно за другим
  - генерирует синхронную/асинхронную передачу данных между хостом и GPU

OpenACC Execution Model

- Модель исполнения OpenACC имеет 3 уровня параллелизма (абстрактные):
  - *gang*, *worker* and *vector*
- Они используются для отображение на реальную физическую архитектуру, которая представляет собой группа обрабатывающих элементов (Processing Elements (PEs))
  - Каждый элемент (PE) имеет много нитей и каждая нить может выполнять векторные операции
- Одно из возможных отображений на GPU: gang=block, worker=warp, vector=threads внутри warp-а
  - Схема отображения зависит от того, что компилятор считает оптимальным в конкретной ситуации

Mapping to multi dimensional blocks and grids

- Группа вложенных for-циклов генерирует многомерные блоки и решетки

```
#pragma acc kernels loop gang(100), vector(16)
    for( Y… )
#pragma acc loop gang(200), vector(32)
        for( X… )
```

Memory Model

- Некоторые GPU не поддерживают когерентности памяти при параллельном выполнении разных обрабатывающих элементов
- Программно-управляемый или аппаратно-управляемый кэш

CUDA C/Fortran vs. OpenACC

- **CUDA C/Fortran:**
  - \+ Высокая производительность при ручной настройке ядер
  - \+ Инкрементальная переносимость на GPU
  - \- Только CUDA-платформы, несовместимость с другими
  - \- Необходима поддержка 2 наборов кода

- **OpenACC:**
  - \+ Возможна высокая производительность
  - \+ Инкрементальная переносимость на GPU
  - \+ Совместимость с другими non-CUDA-платформами
  - \+ Необходима поддержка только 1 набора кода
  - \- Неясное поведение компилятора
  - \- Пока нет доступного «best practice» руководства
  - \- Зависимость от конкретной реализации компилятора

OpenACC API

Directive Syntax

- **Fortran**

  !$acc *directive [clause [, clause] …]*
  structured block
  !$acc end *directive*

- **C**

  #pragma acc *directive [clause [, clause] …] new-line*
  structured block

- **Compilation (with PGI Accelerator compiler)**

  pgfortran -acc -Minfo=accel -ta=nvidia <file_name>
  pgcc –acc -Minfo=accel -ta=nvidia <file_name>

Parallel Construct

Fortran

```
!$acc parallel [clause [, clause]…]
        structured block
!$acc end parallel
```

C

```
#pragma acc parallel [clause [, clause]…] new-line
        structred block
```

Clauses

General clauses

- if(condition)
- async [(exp)]
- num_gangs(exp)
- num_workers(exp)
- vector_length(exp)
- reduction(operator:list)

Data clauses

- copy(list)
- copyin(list)
- copyout(list)
- create(list)
- present(list)
- present_or_copy(list)
- present_or_copyin(list)
- present_or_copyout(list)
- present_or_create(list)
- deviceptr(list)
- private(list)
- firstprivate(list)

Kernels Construct

Fortran

```
!$acc acc kernels [clause [, clause]…]
        structured block
!$acc end kernels
```

C

```
#pragma acc kernels [clause [, clause]…] new-line
        structred block
```

Kernels Construct

```
!$acc kernels
   do i = 1,n                          kernel 1
   do j = 1,n
      a(i,j) = 0.0
   enddo
   enddo
   do k = 1,n                          kernel 2
      b(k) = 1.0
   enddo
!$acc end kernels
```

Clauses

General clauses

- if(condition)
- async [(exp)]

Data clauses

- copy(list)
- copyin(list)
- copyout(list)
- create(list)
- present(list)
- present_or_copy(list)
- present_or_copyin(list)
- present_or_copyout(list)
- present_or_create(list)
- deviceptr(list)

Loop Construct

Fortran

```
!$acc loop [clause [, clause]…]
        do loop
```

C

```
#pragma acc loop [clause [, clause]…]
new-line
        for loop
```

Clauses

- collapse(n)
- gang[(exp)]
- worker[(exp)]
- vector [(exp)]
- seq
- independent
- private(list)
- reduction(op: list)

Combined Directives

Fortran

```
!$acc parallel loop [clause ...]
        do loop
[!$acc end parallel loop]

!$acc kernels loop [clause...]
        do loop
[!$acc end kernels loop]
```

C

```
#pragma acc parallel loop [clause...]
        for loop

#pragma acc kernels loop [clause...]
        for loop
```

Mapping to CUDA threads and blocks

```
#pragma acc kernels
   for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

16 блоков, 256 нитей в каждом

```
#pragma acc kernels loop gang(100) vector(128)
   for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

100 блоков, 128 нитей в каждом, каждая нить выполняет 1 итерацию цикла, используя директиву kernels

```
#pragma acc parallel num_gangs(100) vector_length(128)
{
   #pragma acc loop gang vector
   for( int i = 0; i < n; ++i ) y[i] += a*x[i]; }
```

100 блоков, 128 нитей в каждом, каждая нить выполняет 1 итерацию цикла, используя директиву parallel

Mapping to CUDA threads and blocks

```
#pragma acc parallel num_gangs(100)
{
   for( int i = 0; i < n; ++i ) y[i] += a*x[i]; }
```

100 блоков, каждый имеет ровно 1 нить, каждая нить избыточно выполняет цикл

```
#pragma acc parallel num_gangs(100)
{
   #pragma acc loop gang
   for( int i = 0; i < n; ++i ) y[i] += a*x[i]; }
```

компилятор замечает, что только gangs создаются, и он может решить использовать threads вместо gangs, например, 2 блока из 50 нитей.

Mapping to CUDA threads and blocks

```
#pragma acc kernels loop gang(100) vector(128)
   for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

100 блоков, 128 нитей в каждом, каждая нить выполняет 1 итерацию цикла, используя директиву kernels

```
#pragma acc kernels loop gang(50) vector(128)
   for( int i = 0; i < n; ++i ) y[i] += a*x[i];
```

50 блоков, 128 нитей в каждом, каждая нить выполняет 2 итерации

Выполнение множество итераций в каждой нити может увеличить производительность путем уменьшения времени инициализации

Data Clauses

Fortran

Syntax: array(lower_bound:upper_bound [, lb:ub] …)

Examples: a(:, :), a(1:100, 2:n)

C

Syntax: array[start: length]

Examples: a[2:n] // means a[2], a[3], …, a[2+n-1]

Other Directives

- host_data construct
  - доступ к адресам данных ускорителя на хосте
- cache construct
  - кэширование данных в разделяемой памяти (shared memory)
- update directive
  - обновление значений элементов всего массива или части его в соответствии со значениями этого массива в памяти GPU
- wait directive
  - ожидание завершения асинхронных активностей на GPU (parallel or kernel s region or update directive)
- declare directive
  - описание данных, выделяемых в памяти GPU в неявном дата-регионе, созданном при выполнении подпрограммы

Host_Data Construct

Fortran

```
!$acc host_data [clause [, clause]…]
        structured block
!$acc end host_data
```

C

```
#pragma acc host_data [clause [, clause]…] new-line
        structred block
```

Clause: use_device(list)

Cache Directive

Fortran

!$acc cache (list)

Example: !$acc cache (arr (l1:b1, l2:b2))

C

#pragma cache (list) new-line

Example: #pragma cache (arr[start:length])

Update Directive

Fortran

!$acc update clause [[,] clause]]…

C

#pragma acc update clause [[,] clause]]…
new-line

Clauses

- host(list)
- device(list)
- if(condition)
- async [(exp)]

Wait Directive

Fortran

```
!$acc wait [(expression)]
```

C

```
#pragma acc [(expression)]new-line
```

Declare Directive

Fortran

```
!$acc declare [declclause [, declclause]…]
```

C

```
#pragma acc declare [clause [, clause]…]
new-line
        for loop
```

Clauses

- copy(list)
- copyin(list)
- copyout(list)
- create(list)
- present(list)
- present(list)
- present_or_copy(list)
- present_or_copyin(list)
- present_or_copyout(list)
- present_or_create(list)
- deviceptr(list)
- device_resident(list)

Runtime Library Routines

Fortran

use openacc
#include “openacc_lib.h”

- acc_get_num_devices
- acc_set_device_type
- acc_get_device_type
- acc_get_device_num
- acc_set_device_num
- acc_async_test
- acc_async_test_all

C

#include “openacc.h”

- acc_async_wait
- acc_async_wait_all
- acc_shutdown
- acc_init
- acc_malloc
- acc_free

…

Env and Conditional Compilation

- ACC_DEVICE device
  тип ускорителей (GPU)
- ACC_DEVICE_NUM num
  номер ускорителя
- _OPENACC
  препроцессорная директива для условной компиляции

Примеры

Example

```
!Jacobi solver
        do while (change > tolerance)
            change = 0.0
            iter = iter + 1
!Parallel region to be executed on GPU
            do  j = 2, n-1
            do  i = 2, n-1
                newa(i,j)= w0 * a(i,j) + &
                    w1 * (a(i-1,j)+a(i,j-1)+a(i+1,j)+a(i,j+1)) + &
                    w2 * (a(i-1,j-1)+a(i-1,j+1)+a(i+1,j-1)+a(i+1,j+1))
               change = max(change, abs(newa(i,j) - a(i, j)))
            enddo
            enddo
           a(2:n-1,2:n-1) = newa(2:n-1,2:n-1)
!end of parallel region
        enddo
```

Example

```
!Jacobi solver
        do while (change > tolerance)
            change = 0.0
            iter = iter + 1
!$acc parallel
            do  j = 2, n-1
            do  i = 2, n-1
                newa(i,j)= w0 * a(i,j) + &
                    w1 * (a(i-1,j)+a(i,j-1)+a(i+1,j)+a(i,j+1)) + &
                    w2 * (a(i-1,j-1)+a(i-1,j+1)+a(i+1,j-1)+a(i+1,j+1))
               change = max(change, abs(newa(i,j) - a(i, j)))
            enddo
            enddo
           a(2:n-1,2:n-1) = newa(2:n-1,2:n-1)
!$acc end parallel
        enddo
```

Example

```
!$acc data copy(a, newa)
        do while (change > tolerance)
            change = 0.0
            iter = iter + 1
!$acc parallel reduction(max:change)
            do  j = 2, n-1
            do  i = 2, n-1
                newa(i,j)= w0 * a(i,j) + &
                    w1 * (a(i-1,j)+a(i,j-1)+a(i+1,j)+a(i,j+1)) + &
                    w2 * (a(i-1,j-1)+a(i-1,j+1)+a(i+1,j-1)+a(i+1,j+1))
               change = max(change, abs(newa(i,j) - a(i, j)))
            enddo
            enddo
!$acc end parallel
!$acc parallel
            a(2:n-1,2:n-1) = newa(2:n-1,2:n-1)
!$acc end parallel
        enddo
!$acc end data
```

Example

```
!$acc data copyin(a), create(newa)
        do while (change > tolerance)
            change = 0.0
            iter = iter + 1
!$acc parallel reduction(max:change)
            do  j = 2, n-1
            do  i = 2, n-1
                newa(i,j)= w0 * a(i,j) + &
                    w1 * (a(i-1,j)+a(i,j-1)+a(i+1,j)+a(i,j+1)) + &
                    w2 * (a(i-1,j-1)+a(i-1,j+1)+a(i+1,j-1)+a(i+1,j+1))
               change = max(change, abs(newa(i,j) - a(i, j)))
            enddo
            enddo
!$acc end parallel
!$acc parallel
            a(2:n-1,2:n-1) = newa(2:n-1,2:n-1)
!$acc end parallel
        enddo
!$acc end data
```

Example

```
!$acc data copyin(a), create(newa)
        do while (change > tolerance)
            change = 0.0
            iter = iter + 1
!$acc kernels loop reduction(max:change), gang(32), worker(8)
            do  j = 2, n-1
            do  i = 2, n-1
                newa(i,j)= w0 * a(i,j) + &
                    w1 * (a(i-1,j)+a(i,j-1)+a(i+1,j)+a(i,j+1)) + &
                    w2 * (a(i-1,j-1)+a(i-1,j+1)+a(i+1,j-1)+a(i+1,j+1))
               change = max(change, abs(newa(i,j) - a(i, j)))
            enddo
            enddo
!$acc end kernels loop
!$acc parallel
            a(2:n-1,2:n-1) = newa(2:n-1,2:n-1)
!$acc end paralel
        enddo
!$acc end data
```

Example

```
pgfortran -fast -ta=nvidia:4.0 -Mcuda=cc13 -Minfo=accel -o jac_p jac.f90
jacobi:
     42, Generating copyin(a(1:n,1:n))
         Generating copyout(a(2:n-1,2:n-1))
         Generating local(b(:,:))
     43, Loop is parallelizable
     44, Loop is parallelizable
         Accelerator kernel generated
         43, !$acc do parallel, vector(16) ! blockidx%y threadidx%y
         44, !$acc do parallel, vector(16) ! blockidx%x threadidx%x
             Cached references to size [18x18] block of 'a'
         48, Max reduction generated for cur_eps
     51, Loop is parallelizable
         Accelerator kernel generated
         51, !$acc do parallel, vector(16) ! blockidx%x threadidx%x
             !$acc do parallel, vector(16) ! blockidx%y threadidx%y
```

Example

- **Tesla T10 Processor**

```
[conqueror@tesla-cmc Tutorial2]$ ./J4.exe 1024
reached delta= 0.09998 in          3430 iterations for 1024 x 1024
array
time = 25.8760 seconds
```

- **Intel(R) Xeon(R) CPU E5620 @2.40GHz**

```
[conqueror@tesla-cmc sc1]$ ./j5 1024
Jacobi 1024 x 1024
JacobiHost converged in 3427 iterations to residual 0.099976
time(host) = 140.386185 seconds
```

Example

	N=400	N=512	N=1024
CPU Single thread	6.7759	16.0250	140.3861
OpenMP (hand-tuned)	1.8580	3.7771	29.6452
PGI Accelerator	6.8860	8.9890	25.8760
CUDA C (hand-tuned)	3.8095	4.1140	6.5899

Resources

- http://parallel-compute.com
- http://www.openacc-standard.org
- http://www.pgroup.com/resources/accel.htm
- http://developer.nvidia.com/category/zone/cuda-zone

http://www.parallel-computing.pro

e-mail: dn@parallel-computing.pro

Thank you! ☺