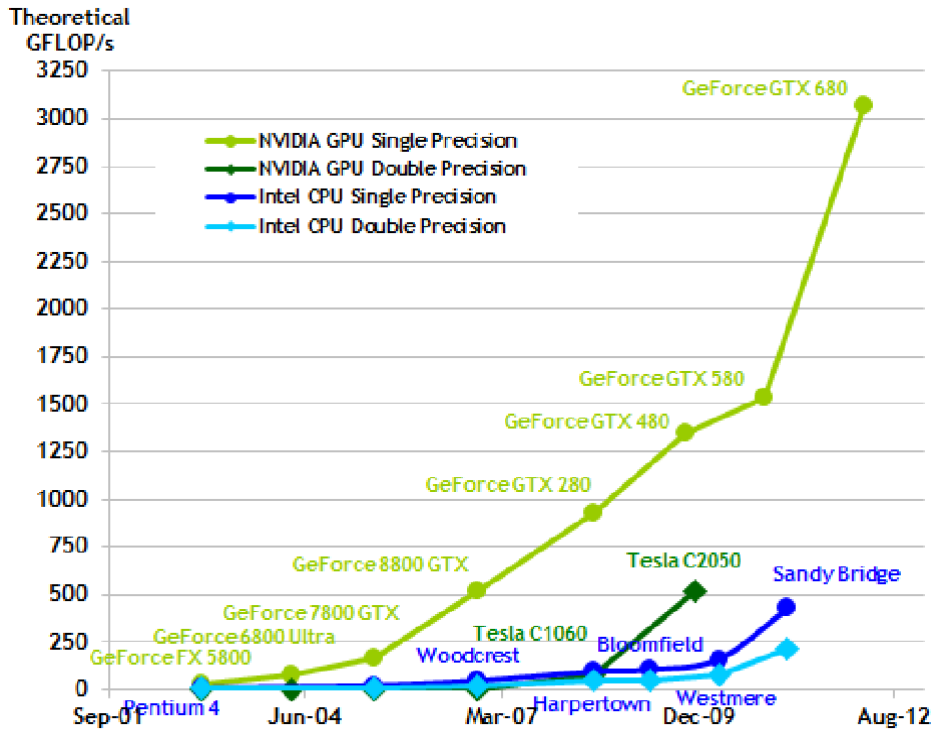
Лекция 1: *содержание курса; обзор использования GPU для прикладных вычислений; архитектура CUDA*.

Содержание курса

- аппаратные особенности графических процессоров;
- линейка графических процесоров NVIDIA;
- архитектура CUDA – основные свойства и принципы;
- программная модель: хост, устройства, ядра, иерархия нитей (*threads*);
- иерархия памяти;
- работа с глобальной памятью, выделение памяти, совместный доступ к памяти (*coalescing*);
- профилирование и отладка программ, выполняемых на GPU;
- работа с разделяемой памятью;
- работа с текстурной памятью;
- использование библиотеки *thrust*;
- потоки CUDA (*streams*);
- использование нескольких GPU.

Рост производительности GPU (август 2012)

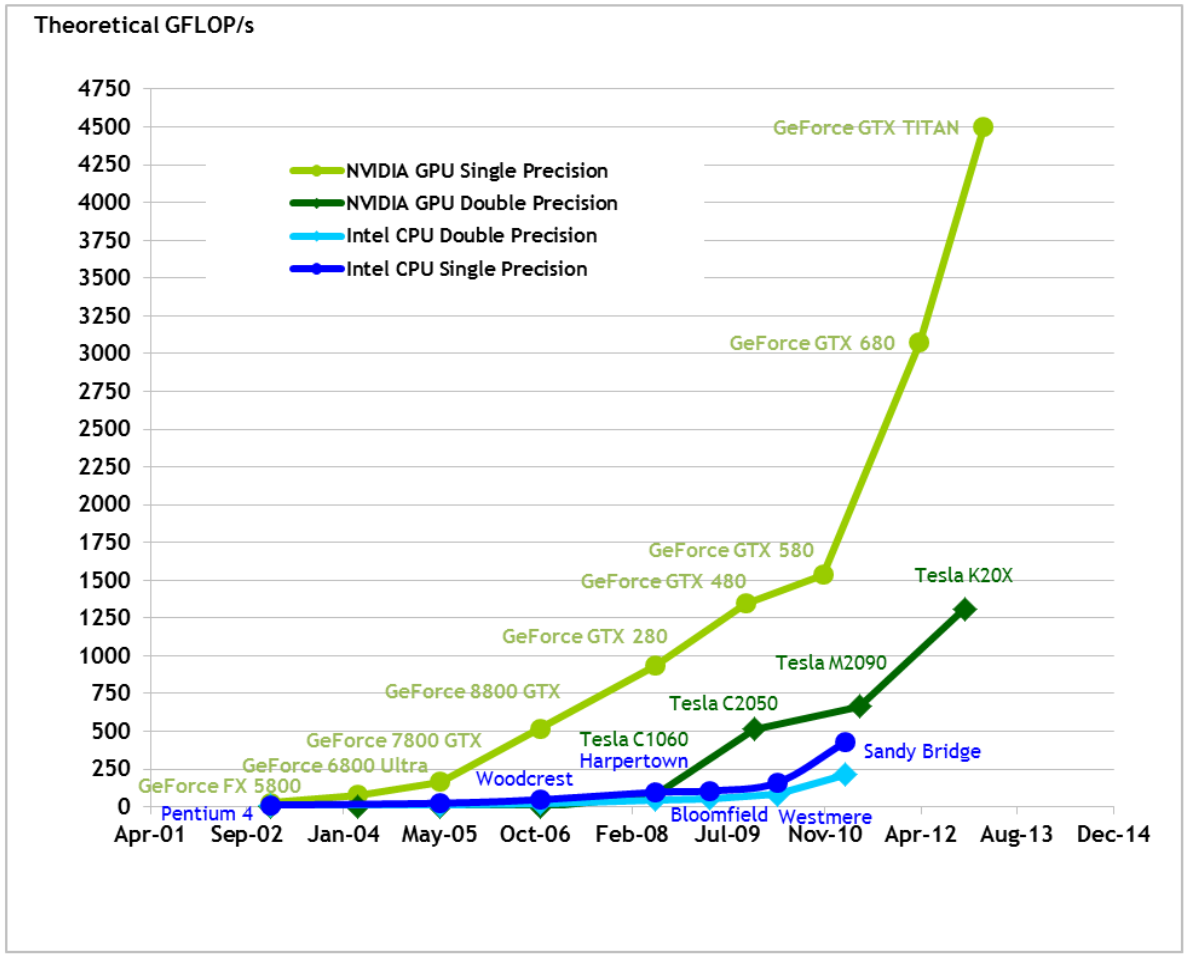
http://www.nvidia.ru/object/cuda-parallel-computing-ru.html



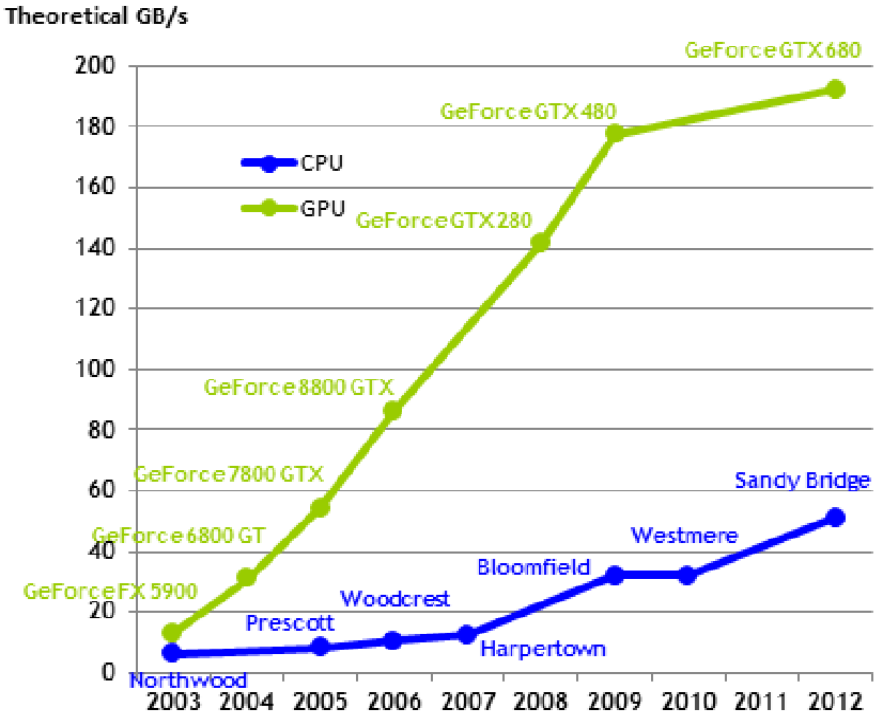
Intel(R) Core(TM) i5-2500K CPU @ 3.30GH - **100Gflop/s**
GeForce GTX 560 Ti - **1260 Gflop/s**

Производительность CPU и GPU.

Рост производительности GPU (август 2013)



Рост пропускной способности GPU



Пропускная способность CPU и GPU.

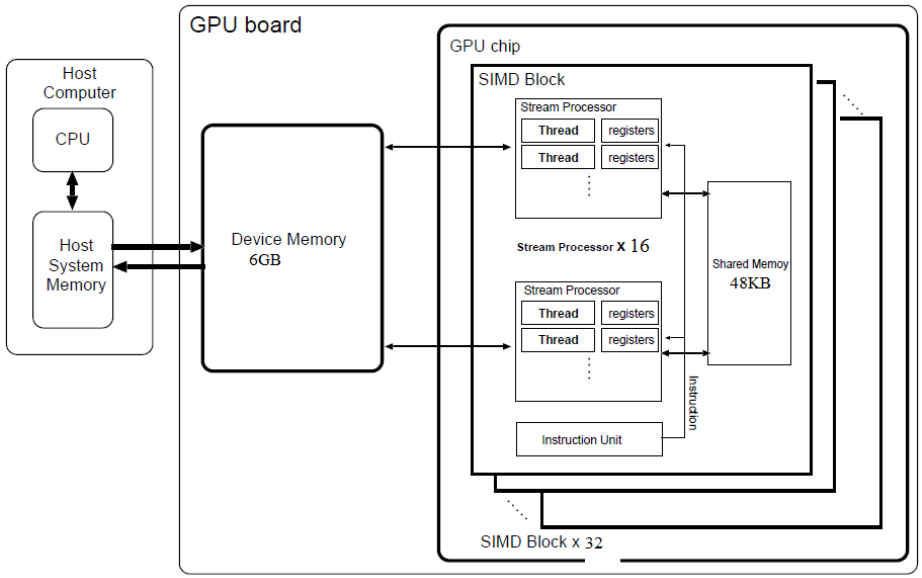
PCI Express 1.0	4Gb
PCI Express 2.0	16Gb
PCI Express 3.0	25.6Gb

Пропускная способность внешнего интерфейса.

Физическое представление GPU

В общих чертах графический процессор состоит из набора SIMD (*Single Instruction –Multiple Data*) блоков или multiprocessors, каждый из которых включает в себя определенное количество stream processors (обычно 8-16), несколько ALU (Arithmetic Logical Unit) и единственный instruction unit. Микросхема SIMD блока также содержит небольшую быструю память, выполняющую роль кэша – shared memory, и небольшую память для чтения – кэши константной и текстурной памяти. Каждый из stream processors имеет набор регистров, куда загружаются данные, обрабатываемые одной инструкцией одновременно.

На плате GPU располагается микросхема памяти большого объема. GPU соединяется с узлом вычислительной системы посредством интерфейса PCI Express.

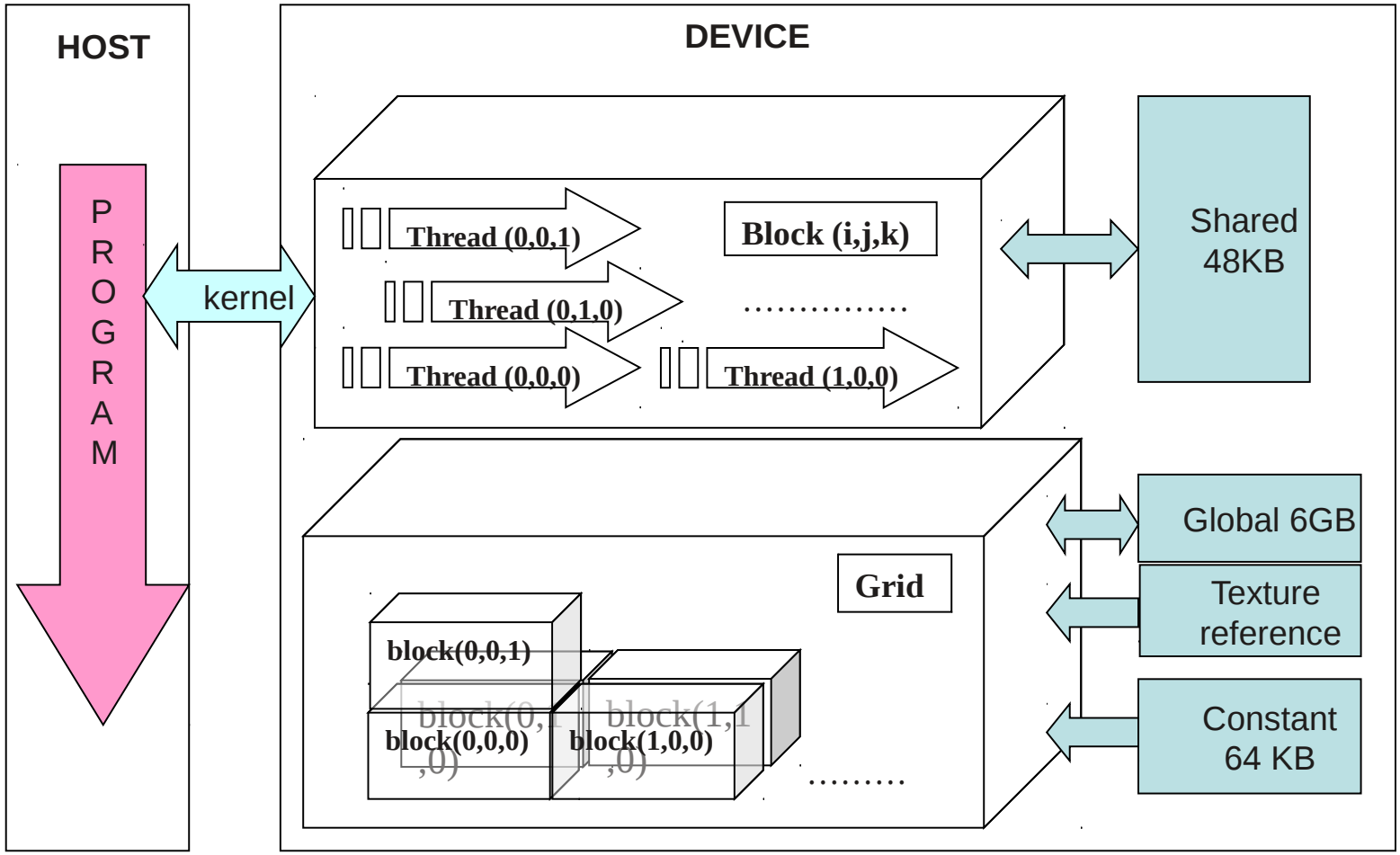


Логическое представление GPU

Активное использование графических процессоров (GPU) для прикладных расчетов научно-технического назначения во многом связано с предоставлением компанией NVIDIA технологии ***CUDA*** (*Cuda Unified Device Architecture*) (позже конкурентом CUDA выступила технология *OpenCL*, представляющая универсальный подход для программирования специализированных вычислительных устройств, и поддерживаемая компаниями *AMD*, *Intel* и *Nvidia*). Технология CUDA предоставляет понятную для прикладного программиста абстракцию графического процессора (GPU) и простой интерфейс прикладного программирования (*API – Application Programming Interface*).

По терминологии *CUDA* вычислительный узел с *CPU* и main memory называется ***host***, *GPU* называется ***device***. Программа, выполняемая на host'е содержит код – ядро (***kernel***), который загружается на device в виде многочисленных копий. Все копии загруженного кода – нити (***threads***), объединяются в блоки (***blocks***) по 512-1024 нити в каждом. Все блоки объединяются в сеть (***grid***) с максимальным количеством блоков 65536. Все нити имеют совместный доступ на запись/чтение к памяти большого объема - global memory, на чтение к кэшируемым ***constant memory*** и ***texture memory***. Нити одного блока имеют доступ к быстрой памяти небольшого объема – ***shared memory***.

Логическое представление GPU (схема)



Compute capabilities (вычислительные возможности)

Compute capabilities (вычислительные возможности) представляют спецификацию GPU. Особенности “вычислительных возможностей” включают допустимость операций с плавающей точкой, допустимость атомарных операций, возможность синхронизации нитей, кэшируемость глобальной памяти и т.д. Описание различных версий Compute capabilities можно найти, например, в CUDA C Programming Guid – руководстве по CUDA C компании NVIDIA.

Архитектура GPU	Compute capabilities	Версия CUDA
Tesla	1.*	CUDA 2.*-3.*
Fermi	2.*	CUDA 4.*-5.*
Kepler	3.*	CUDA 5.*

Характеристики GPU на серверах 'home' и 'dew'

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 560 Ti"
CUDA Driver Version / Runtime Version 5.0 / 5.0
CUDA Capability Major/Minor version number: 2.1
Total amount of global memory: 2048 MBytes (2147024896 bytes)
(8) Multiprocessors x (48) CUDA Cores/MP: 384 CUDA Cores
GPU Clock rate: 1645 MHz (1.64 GHz)
Memory Clock rate: 2004 Mhz
Memory Bus Width: 256-bit
L2 Cache Size: 524288 bytes
Max Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
Max Layered Texture Size (dim) x layers 1D=(16384) x 2048, 2D=(16384,16384) x 2048
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 32768
Warp size: 32
Maximum number of threads per multiprocessor: 1536
Maximum number of threads per block: 1024
Maximum sizes of each dimension of a block: 1024 x 1024 x 64
Maximum sizes of each dimension of a grid: 65535 x 65535 x 65535
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Concurrent copy and kernel execution: Yes with 1 copy engine(s)
Run time limit on kernels: Yes
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support: Disabled
Device supports Unified Addressing (UVA): Yes
Device PCI Bus ID / PCI location ID: 1 / 0
Compute Mode:
< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

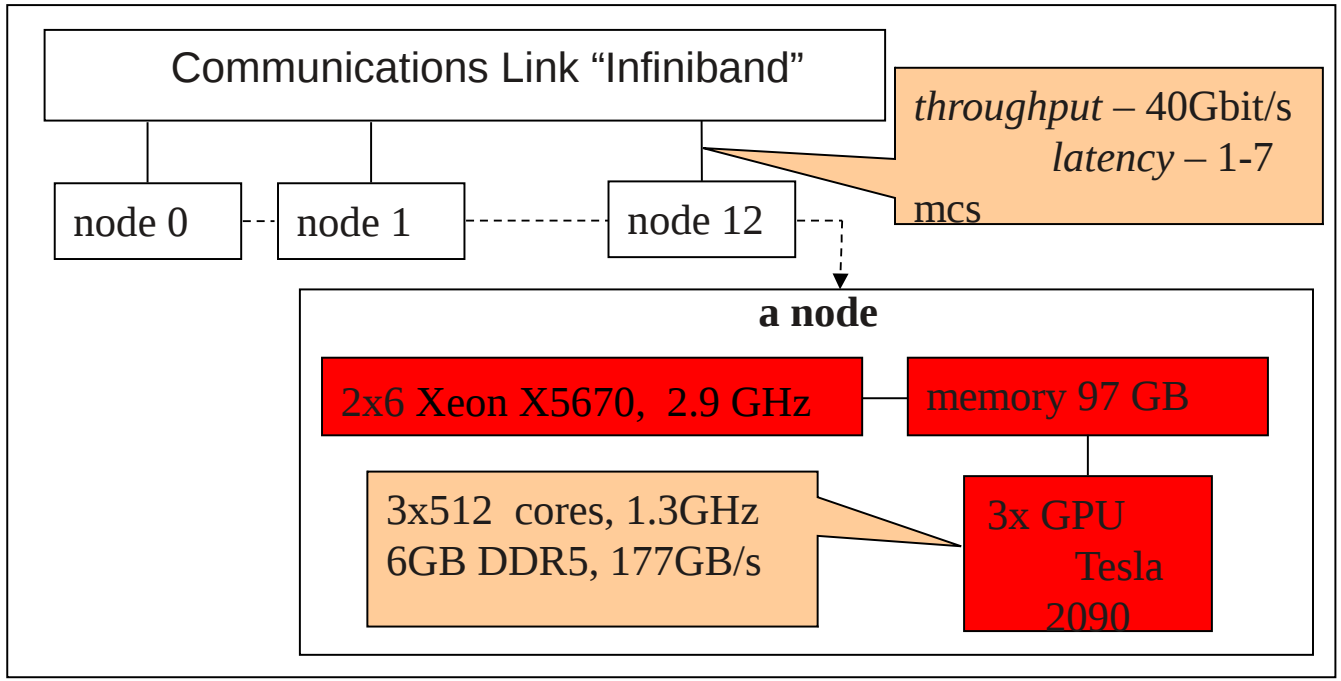
Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 650"
CUDA Driver Version / Runtime Version 5.5 / 5.5
CUDA Capability Major/Minor version number: 3.0
Total amount of global memory: 2048 MBytes (2147155968 bytes)
(2) Multiprocessors x (192) CUDA Cores/MP: 384 CUDA Cores
GPU Clock rate: 1110 MHz (1.11 GHz)
Memory Clock rate: 2500 Mhz
Memory Bus Width: 128-bit
L2 Cache Size: 262144 bytes
Max Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536,65536), 3D=(4096,4096,4096)
Max Layered Texture Size (dim) x layers 1D=(16384) x 2048, 2D=(16384,16384) x 2048
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 65536
Warp size: 32
Maximum number of threads per multiprocessor: 2048
Maximum number of threads per block: 1024
Maximum sizes of each dimension of a block: 1024 x 1024 x 64
Maximum sizes of each dimension of a grid: 65535 x 65535 x 65535
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Concurrent copy and kernel execution: Yes with 1 copy engine(s)
Run time limit on kernels: Yes
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support: Disabled
Device supports Unified Addressing (UVA): Yes
Device PCI Bus ID / PCI location ID: 1 / 0
Compute Mode:
< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

Гибридный кластер НГУ (http://nusc.ru)

Кластер состоит из 12 узлов *HP SL390s G7* , каждый из которых содержит два 6-ядерных CPU *Xeon X5670* и три графические карты *NVIDIA Tesla M2090*.

Каждый GPU имеет **512 ядер** (cores) с частотой 1.3GHz и память размером **6GB** с пропускной способностью (bandwidth) 177 GB/s.



Характеристики GPU на гибридном кластере 'nusc.ru'

Found 3 CUDA Capable device(s)
Device 0: "Tesla M2090"
CUDA Driver Version / Runtime Version 4.2 / 4.1
CUDA Capability Major/Minor version number: 2.0
Total amount of global memory: 5375 MBytes (5636554752 bytes)
(16) Multiprocessors x (32) CUDA Cores/MP: 512 CUDA Cores
GPU Clock Speed: 1.30 GHz
Memory Clock rate: 1848.00 Mhz
Memory Bus Width: 384-bit
L2 Cache Size: 786432 bytes
Max Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
Max Layered Texture Size (dim) x layers 1D=(16384) x 2048, 2D=(16384,16384) x 2048
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 32768
Warp size: 32
Maximum number of threads per block: 1024
Maximum sizes of each dimension of a block: 1024 x 1024 x 64
Maximum sizes of each dimension of a grid: 65535 x 65535 x 65535
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Concurrent copy and execution: Yes with 2 copy engine(s)
Run time limit on kernels: No
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Concurrent kernel execution: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support enabled: Yes
Device is using TCC driver mode: No
Device supports Unified Addressing (UVA): Yes
Device PCI Bus ID / PCI location ID: 6 / 0
Compute Mode:
< Exclusive Process (many threads in one process is able to use ::cudaSetDevice() with this device) >

Device 1: "Tesla M2090"
CUDA Driver Version / Runtime Version 4.2 / 4.1
CUDA Capability Major/Minor version number: 2.0
Total amount of global memory: 5375 MBytes (5636554752 bytes)
(16) Multiprocessors x (32) CUDA Cores/MP: 512 CUDA Cores
GPU Clock Speed: 1.30 GHz
Memory Clock rate: 1848.00 Mhz
Memory Bus Width: 384-bit
L2 Cache Size: 786432 bytes
Max Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
Max Layered Texture Size (dim) x layers 1D=(16384) x 2048, 2D=(16384,16384) x 2048
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 32768
Warp size: 32
Maximum number of threads per block: 1024
Maximum sizes of each dimension of a block: 1024 x 1024 x 64
Maximum sizes of each dimension of a grid: 65535 x 65535 x 65535
……

Device 2: "Tesla M2090"
CUDA Driver Version / Runtime Version 4.2 / 4.1
CUDA Capability Major/Minor version number: 2.0
Total amount of global memory: 5375 MBytes (5636554752 bytes)
(16) Multiprocessors x (32) CUDA Cores/MP: 512 CUDA Cores
GPU Clock Speed: 1.30 GHz
Memory Clock rate: 1848.00 Mhz
Memory Bus Width: 384-bit
L2 Cache Size: 786432 bytes
Max Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536,65535), 3D=(2048,2048,2048)
Max Layered Texture Size (dim) x layers 1D=(16384) x 2048, 2D=(16384,16384) x 2048
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 32768
Warp size: 32
Maximum number of threads per block: 1024
Maximum sizes of each dimension of a block: 1024 x 1024 x 64
Maximum sizes of each dimension of a grid: 65535 x 65535 x 65535
……

Программный интерфейс CUDA

CUDA C - расширение языка *C* , и набор функций и структур *CUDA API* предоставляют простой инструмент для программирования на GPU.

Некоторые конструкции программного интерфейса CUDA.

Функция-ядро (*kernel*)

Код, выполняемый на устройстве (ядро), определяется в виде функции типа *void* со спецификатором __global__:

```
__global__ void gFunc(<params>){…}
```

Конфигурация нитей

При вызове ядра программист определяет количество нитей в блоке и количество блоков в grid. При этом допустима линейная, двумерная или трехмерная индексация нитей:

```
gFunc<<<dim3(bl_xdim, bl_ydim, bl_zdim),
         dim3(th_xdim, th_ydim, th_zdim)>>>(<params>);
```

Программный интерфейс CUDA (самый простой пример)

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

__global__ void gTest(float* a){
  a[threadIdx.x+blockDim.x*blockIdx.x]=(float)(threadIdx.x+blockDim.x*blockIdx.x);
}

int main(){
  float *da, *ha;
  int num_of_blocks=10, threads_per_block=32;
  int N=num_of_blocks*threads_per_block;

  ha=(float*)calloc(N, sizeof(float));
  cudaMalloc((void**)&da, N*sizeof(float));

  gTest<<<dim3(num_of_blocks), dim3(threads_per_block)>>>(da);
  //cudaThreadSynchronize();
  CudaDeviceSynchronize();
  cudaMemcpy(ha,da,N*sizeof(float), cudaMemcpyDeviceToHost);

  for(int i=0;i<N;i++)
          printf("%g\n", ha[i]);

  free(ha);
  cudaFree(da);
  return 0;
}
```

CUDA C - расширение языка C , и набор функций и структур CUDA API предоставляют простой инструмент для программирования на GPU.

Комментарии: *использование глобальной памяти*

cudaError_t cudaMalloc (void ∗∗ devPtr, size_t size)

cudaError_t cudaFree (void ∗ devPtr)

cudaError_t cudaMemcpy (void ∗ dst, const void ∗ src, size_t count, enum cudaMemcpyKind kind)

```
enum cudaError
{
  cudaSuccess = 0,
  cudaErrorMissingConfiguration,
  cudaErrorMemoryAllocation,
  cudaErrorInitializationError,
  cudaErrorLaunchFailure,
…..............................................................
};

typedef enum cudaError cudaError_t;
```

```
enum cudaMemcpyKind
{
  cudaMemcpyHostToHost = 0,
  cudaMemcpyHostToDevice,
  cudaMemcpyDeviceToHost,
  cudaMemcpyDeviceToDevice
};
```

Глобальная память ***выделяется только на хосте***, к глобальной памяти возможен ***доступ только на устройстве.***

Документация CUDA: http://docs.nvidia.com/cuda/index.html

Комментарии: *встроенные типы и переменные*

- uint3 **threadIdx** – индекс нити в блоке
- dim3 **blockDim** – размер блока
- uint3 **blockIdx** – индекс блока в гриде
- dim3 **gridDim** - размер грида
- int **warpSize** - количество нитей в варпе (warp)

Комментарии: *синхронизация всех нитей*

Запуск ядра на устройстве (вызов функции с модификатором __global__) происходит в асинхронном режиме.

Для синхронизации нитей служат следующие вызовы:

```
//cudaThreadSynchronize();  //устаревшая функция (depricated)
 cudaDeviceSynchronize();
```

Обработка ошибок (код с обработчиком ошибок)

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

#define CUDA_CHECK_RETURN(value) {
        cudaError_t _m_cudaStat = value;
        if (_m_cudaStat != cudaSuccess) {
                fprintf(stderr, "Error %s at line %d in file %s\n",
                                cudaGetErrorString(_m_cudaStat), __LINE__, __FILE__);
                exit(1);
        } }  //макрос для обработки ошибок

__global__ void gTest(float* a){
  a[threadIdx.x+blockDim.x*blockIdx.x]=(float)(threadIdx.x+blockDim.x*blockIdx.x);
}

int main(){
  float *da, *ha;
  int num_of_blocks=10, threads_per_block=32;
  int N=num_of_blocks*threads_per_block;

  ha=(float*)calloc(N, sizeof(float));
  CUDA_CHECK_RETURN(cudaMalloc((void**)&da, N*sizeof(float)));
```

Обработка ошибок (код с обработчиком ошибок, продолжение)

```
  gTest<<<dim3(num_of_blocks), dim3(threads_per_block)>>>(da);
 // cudaThreadSynchronize();
  CUDA_CHECK_RETURN(cudaDeviceSynchronize());
  CUDA_CHECK_RETURN(cudaGetLastError());
  CUDA_CHECK_RETURN(cudaMemcpy(ha,da,N*sizeof(float), cudaMemcpyDeviceToHost));

  for(int i=0;i<N;i++)
          printf("%g\n", ha[i]);

  free(ha);
  cudaFree(da);
  return 0;
}
```

Обработка ошибок (комментарии)

char * cudaGetErrorString (cudaError_t code); - возвращает строку в таблице кодов ошибок.

cudaError_t cudaGetLastError (); -возвращает код ошибки при асинхронном вызове.

Контроль производительности (код с контрольными точками)

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

#define CUDA_CHECK_RETURN(value) {
        cudaError_t _m_cudaStat = value;
        if (_m_cudaStat != cudaSuccess) {
                fprintf(stderr, "Error %s at line %d in file %s\n",
                                cudaGetErrorString(_m_cudaStat), __LINE__, __FILE__);
                exit(1);
        } }

__global__ void gTest(float* a){
  a[threadIdx.x+blockDim.x*blockIdx.x]=(float)(threadIdx.x+blockDim.x*blockIdx.x);
}

int main(){
  float *da, *ha;
  int num_of_blocks=10, threads_per_block=32;
  int N=num_of_blocks*threads_per_block;

  ha=(float*)calloc(N, sizeof(float));
  CUDA_CHECK_RETURN(cudaMalloc((void**)&da, N*sizeof(float)));
```

Контроль производительности (код с контрольными точками, продолжение)

```
cudaEvent_t  start,stop;  // встроенный тип данных – структура, для фиксации контрольных точек
float elapsedTime;

cudaEventCreate(&start);  // инициализация
cudaEventCreate(&stop); // событий

cudaEventRecord(start,0); // привязка события

gTest<<<dim3(num_of_blocks), dim3(threads_per_block)>>>(da);
// cudaThreadSynchronize();
//CUDA_CHECK_RETURN(cudaDeviceSynchronize());
//CUDA_CHECK_RETURN(cudaGetLastError());

cudaEventRecord(stop,0); // привязка события
cudaEventSynchronize(stop); // синхронизация по событию
cudaEventElapsedTime(&elapsedTime,start,stop); // вычисление затраченного времени

fprintf(stderr,"gTest took %g\n", elapsedTime);

cudaEventDestroy(start);  // освобождение
cudaEventDestroy(stop);  // памяти
```

Контроль производительности (код с контрольными точками, окончание)

```
  CUDA_CHECK_RETURN(cudaMemcpy(ha,da,N*sizeof(float), cudaMemcpyDeviceToHost));

  for(int i=0;i<N;i++)
          printf("%g\n", ha[i]);

  free(ha);
  cudaFree(da);
  return 0;
}
```

```
ewgenij@linux-715l:~/EDUCATION/workshop/Practicals/Practical1> ./3>tmp
gTest took 0.030656
ewgenij@linux-715l:~/EDUCATION/workshop/Practicals/Practical1> time(./3>tmp)
gTest took 0.030848

real    0m0.086s
user    0m0.006s
sys     0m0.058s
```

Компиляторы

Сравнение компиляторов nvcc и pgi:

```
ewgenij@linux-715l:~/EDUCATION/workshop/Practicals/Practical1> pgcpp 3.cu -o 3p
ewgenij@linux-715l:~/EDUCATION/workshop/Practicals/Practical1> nvcc 3.cu -o 3n
ewgenij@linux-715l:~/EDUCATION/workshop/Practicals/Practical1> time(./3p>tmp)
gTest took 24.0531

real    0m0.025s
user    0m0.025s
sys     0m0.001s
ewgenij@linux-715l:~/EDUCATION/workshop/Practicals/Practical1> time(./3n>tmp)
gTest took 0.004992

real    0m0.113s
user    0m0.011s
sys     0m0.058s
```

Пример строки компиляции nvcc (компиляция и компоновка dll):

```
librelaxation.so: relaxation.cpp kerns_st.cu
        nvcc -arch=compute_20  -Xcompiler -fPIC  -shared relaxation.cpp kerns_st.cu   -I../include
-I/home/ewgenij/common/inc      -L/home/ewgenij/common/lib -lcutil_x86_64    -o   ../../lib/librelaxation.so
```

Упражнения

- Сравнить время выполнения последовательного и параллельного кода инициализации и сложения двух векторов большой размерности на выбранном CPU и GPU, установленными на серверах home, dew и кластере clu.nusc.ru.
- То же самое для задачи транспонирования матрицы.