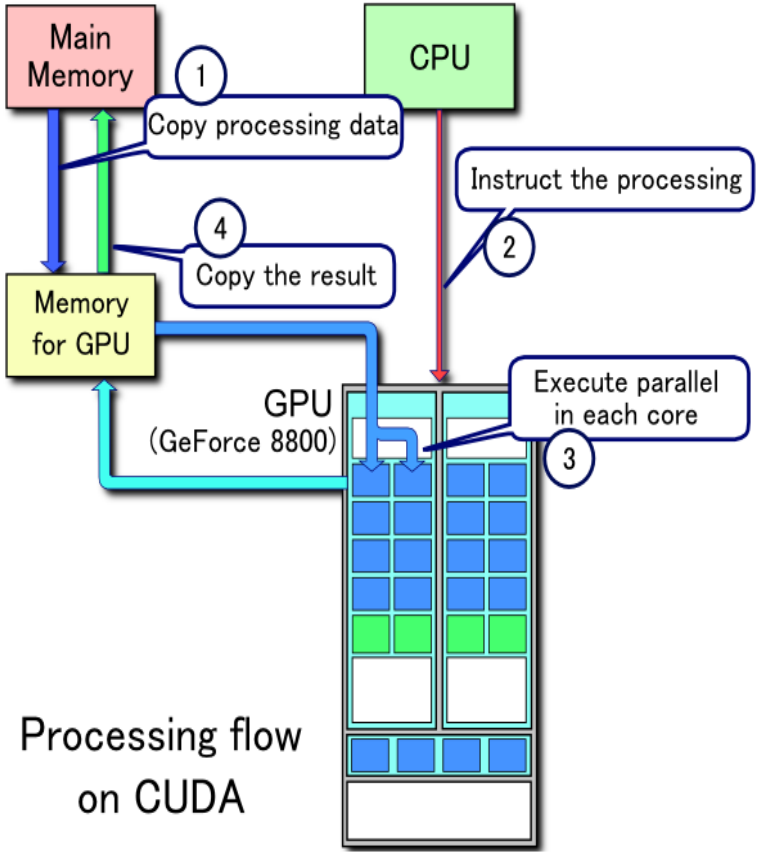
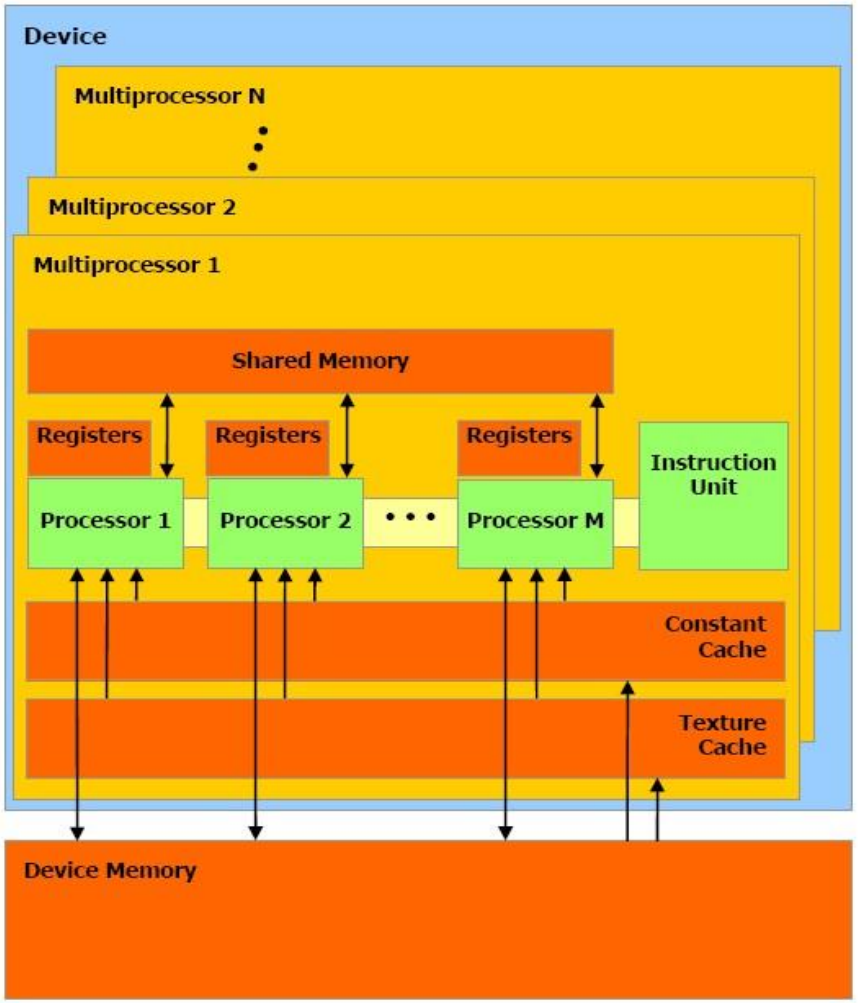
CUDA

CUDA

- CUDA mint architektúra
  - Párhuzamos feldolgozásra optimalizált architektúra
- CUDA mint GPGPU keretrendszer
  - Runtime és Driver API
  - CUDA C/C++
  - NVCC fordító
- CUDA ecosystem
  - CUBLAS
  - CUFFT
  - CUSPARSE
  - CURAND
  - Thrust

CUDA architektúra





CUDA architektúra

- Compute Capability
  - Az eszközök csoportosítása a támogatott funkciók szerint
  - Az eszközök visszafelé kompatibilisek
  - Major.Minor sorszámozás
    - Major: architektúra jelölés
      - 1.x: Tesla
      - 2.x: Fermi
      - 3.x: Kepler
    - Minor: architektúrán belüli fejlesztések

CUDA architektúra

- Compute Capability
  - 1.x: Első generációs GPGPU képességek
    - 1.1: Atomi műveletek a globális memórián
    - 1.2: Atomi műveletek az osztott memórián
    - 1.3: Dupla pontosságú számítások
  - 2.x: Erősebb GPGPU támogatás
    - Atomi műveletek float értékeken
    - Szinkronizációs primitívek
    - Hatékonyabb memória kezelés
  - 3.x: Nagy problématér adaptív támogatása
    - 3.5: Dynamic Parallelism

CUDA keretrendszer

- Driver API
  - Alacsony szintű hívások
  - Hasonló koncepcióra épül mint az OpenCL
    - Device, Context, Module, Function
    - Heap memory, CUDA Array, Texture, Surface
- Runtime API
  - Magas szintű felületet nyújt a programozáshoz
  - Támogatja a host és device függvények keverését
  - Automatikus keretrendszer menedzsment

CUDA C/C++

- Támogatja a C/C++ szabvány jelentős részét
  - Adatgyűjtő osztályok
  - Osztályok származtatása
  - Osztály sablonok
  - Függvény sablonok
  - Funktorok
- Nem támogatja
  - Futásidejű típus információk (RTTI)
  - Kivételek
  - C++ Standard Library

NVCC fordító

- A fordítás menete
  - A forráskód szétválasztása host és device kódra
  - A host kód kiegészítése CUDA specifikus kódrészekkel
    - A továbbiakban a host fordító dolgozik vele
  - A device kód fordítása a megfelelő architektúrára
    - Az NVIDIA device fordító hozza létre belőle a binárist
  - A host és device binárisok összeszerkesztése

CUDA példa

```
#include <cuda.h>

__global__ void square(int* dataGPU, int dataSize){
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  dataGPU[index] = dataGPU[index] * dataGPU[index]
}

int main(int argc, char* argv[]){
  const int dataSize = 1024;
  int* dataCPU = (int*)malloc(sizeof(int)*dataSize);
  for(int i = 0; i < dataSize; ++i){
    dataCPU[i] = i;
  }

  int* dataGPU;
  cudaMalloc(&dataGPU, sizeof(int)*dataSize);
  cudaMemcpy(dataGPU, dataCPU, sizeof(int)*dataSize, cudaMemcpyHostToDevice);

  int threadsPerBlock = 256;
  int blocksPerGrid = 4;
  square<<<blocksPerGrid, threadsPerBlock>>>(dataGPU, dataSize);

  cudaMemcpy(dataCPU, dataGPU, sizeof(int)*dataSize, cudaMemcpyDeviceToHost);

  int wrongCount = 0;
  for(int i = 0; i < dataSize; ++i){
    if(dataCPU[i] != i * i) wrongCount++;
  }
  printf(„Number of wrong squares: %d\n”, wrongCount);
  cudaFree(dataGPU);
}
```

CUDA inicializáció

- CUDA Runtime api inicializálása
  - Automatikus az első CUDA hívásnál
  - Minden elérhető eszközhöz kontextust hoz létre
  - cudaDeviceReset() megszünteti a kontextusokat
- Eszköz memória kezelése

```
int* dataGPU;
cudaMalloc(&dataGPU, sizeof(int)*dataSize);
```

  - Lineáris memória - cudaMalloc() és cudaFree()
  - CUDA Array – textúra memória

CUDA memória

- Memória típusok
  - Szálankénti lokális memória
    - A regiszter tömbben allokált terület
    - Csak az adott szálban elérhető
    - Spill-store használata, ha nincs elegendő regiszter!
  - Konstans memória
  - Osztott memória
    - Közösen használható memória blokkonként
  - Globális memória
    - GPU saját memóriájában
    - Lineáris memória terület
    - Textúra memória

CUDA memória

- Memória típusok
  - Host oldali CUDA memória
    - Page-Locked Host Memory
      - Nem lapozható memória terület a host oldalon
      - cudaHostAlloc(), cudaFreeHost(), cudaHostRegister()
      - A másolása átlapolható a kernel futásával
      - Nagyobb sebességgel másolható mint a sima host oldali memória

CUDA memória

- Memória típusok
  - Host oldali CUDA memória
    - Portable Memory
      - A page-locked memória előnyös tulajdonságai csak az adott kontextusra érvényesek
      - A portable flag az allokáláskor kiterjeszti ezt az összes eszközre
    - Write-Combining Memory
      - Nem cache-elhetővé teszi a lefoglalt területet
      - Akár 40%-al gyorsabban másolható
      - Host oldali olvasása a cache hiánya miatt lassú

CUDA memória

- Memória típusok
  - Host oldali CUDA memória
    - Mapped Memory
      - A host oldalon lefoglalt memóriát elérhetővé teszi a GPU-n is
      - A kernel futása alatt igény szerint másolódik a GPU-ra
        - + Nincs szükség a GPU-n lefoglalt területre másolásra
        - + Nincs szükség a streamekre a memória másolás és kernel futtatás átlapolásához
        - - A PCI-E busz sávszélessége korlátozó tényező
- Unified Virtual Address Space
  - Compute Capability 2.0-tól elérhető
  - Egységes címtérbe rendezi a host és device memóriát
  - Egy pointerről egyértelműen eldönthető melyik eszközre mutat

CUDA memória

- Eszköz memória kezelése

```
cudaMemcpy(dataGPU, dataCPU, sizeof(int)*dataSize,
                                cudaMemcpyHostToDevice);
```

  - cudaMemcpy* függvény csoport
  - Az utolsó paraméter határozza meg az irányt
    - cudaMemcpyHostToDevice
    - cudaMemcpyDeviceToHost
    - cudaMemcpyDeviceToDevice

CUDA memória

- Eszköz memória kezelése
  - Peer-To-Peer memória elérés
    - Legalább CC2.0 Tesla kártyák között
    - cudaDeviceCanAccessPeer() – a támogatás lekérdezése
    - cudaDeviceEnablePeerAccess() – engedélyezése

```
cudaSetDevice(0);
float* p0;
cudaMalloc(&p0, size);
cudaSetDevice(1);
cudaDeviceEnablePeerAccess(0,0);
kernel<<<grid, block>>>(p0);
```

  - Peer-To-Peer memória másolás
    - Különböző eszközök között, ha nincs UVAS

CUDA streamek

- CUDA streamek
  - Párhuzamos kernel indítás
  - Másolás és kernel futtatás átlapolása

```
cudaStream_t stream[numberOfStreams];
for(int i = 0; i < numberOfStreams; ++i)
        cudaStreamCreate(&stream[i]);
…
for(int i = 0; i < 2; ++i)
{
        cudaMemcpyAsync(…,cudaHostToDevice, stream[i]);
        kernel<<<grid,block,shared,stream[i]>>>(params, …);
        cudaMemcpyAsync(…, cudaDeviceToHost, stream[i]);
}
…
for(int i = 0; i < numberOfStreams; ++i)
        cudaStreamDestroy(stream[i]);
```

CUDA streamek

- CUDA streamek
  - Alapértelmezett stream, ha nincs megadva
- Szinkronizáció
  - Explicit szinkronizáció
    - cudaDeviceSynchronize() – minden stream befejezése
    - cudaStreamSynchronize() – adott stream befejezése
    - cudaStreamWaitEvent() – streamen belüli szinkronizáció
  - Implicit szinkronizáció
    - Page-locked memória foglalás
    - Device memória foglalás
    - Eszközön belüli másolás
    - Az alapértelmezett streamen parancs végrehajtás
    - Cache konfiguráció változtatás

CUDA kernel

- Kernel indítás

```
int threadsPerBlock = 256;
int blocksPerGrid = 4;
square<<<blocksPerGrid, threadsPerBlock>>>(dataGPU, dataSize);
```

  - Az nvcc alakítja át a valódi formára
    - kernel<<<grid, block, shared,stream>>>(paraméterek, …)
  - A munkaméret megadása bottom-up módú
    - A blokk méret megadja a blokkonkénti szálak számát
    - A grid méret megadja az elindítandó blokkok számát
  - Aszinkron kernel futás

CUDA kernel

- Kernel program

```
 __global__ void square(int* dataGPU, int dataSize){
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  dataGPU[index] = dataGPU[index] * dataGPU[index]
}
```

  - Az nvcc fordítja egy köztes kódra
  - PTX – Parallel Thread Execution ISA
    - Virtuális gép és annak utasításkészlete
    - A cél a GPU generációk közötti hordozhatóság
    - Eszközfüggetlen, a GPU driver fordítja a végső bináris formára

CUDA C

- Függvény jelölők
  - __device__ : csak kernel kódból hívható
  - __global__ : host és kernel kódból hívható
  - __host__ : csak a host kódból hívható

  - __noinline__ : nem beágyazható (cc2.0)
  - __forceinline__ : mindenképpen beágyazandó

CUDA C

- Változó jelölők
  - __device__ : az eszköz memóriájában található
    - Globális memória terület
    - A teljes alkalmazás élete alatt elérhető
    - Minden szálból és a hostról is elérhető
  - __constant__ : konstans változó
    - A konstans memóriában van (cachelt)
    - A __device__ memóriával azonos tulajdonságok
  - __shared__ : osztott memória a blokkon belül
    - Az on-chip memóriában van
    - A blokk futása alatt érhető el
    - Csak a blokkban levő szálak által elérhető

CUDA C

- Változó jelölők
  - __restrict__ : nem átfedő pointerek jelzése
    - Ígéret a fordítónak, hogy a pointerek függetlenek
    - Optimalizációt segíti

CUDA C

- Vektor típusok
  - Host és device oldalon is léteznek
  - type make_type() alakú konstruktorok
  - Maximum négy eleműek lehetnek
    - Kivétel a double és a longlong típusok
  - A szokásos C típusok mindegyikéhez
- dim3 típus a dimenzók jelzéséhez
  - Kernel munkaméret meghatározásához

CUDA C

- Speciális változók
  - dim3 gridDim : aktuális grid méret
  - uint3 blockIdx : aktuális blokk index
  - dim3 blockDim: aktuális blokk méret
  - uint3 threadIdx: aktuális szál azonosító
  - int warpSize: warp méret

CUDA C

- Szinkronizációs függvények
  - Az aktuális blokkon belüli memória szinkronizáció
  - Egyetlen threadből is kiváltható
  - void threadfence_block()
    - Szinkronizál a globális és osztott memóriára
  - void threadfence()
    - Szinkronizál a globális memóriára
  - void threadfence_system()
    - Minden elérhető memória területre szinkronizál

CUDA C

- Szinkronizációs függvények
  - void __syncthreads()
    - Minden szálnak rá kell futni
    - A kódot szinkronizálja a szálak között
  - int __syncthreads(int predicate)
    - Visszaadja azon szálak számát, ahol a predikátum nem nulla
  - int __synchtreads_and(int predicate)
    - Visszaadja, hogy minden szálon nem nulla a predikátum
  - int __syncthreads_or(int predicate)
    - Visszadja, hogy volt-e olyan szál, ahol nem nulla a predikátum

CUDA C

- Beépített függvények
  - Matematikai függvények
  - Textúra és Surface kezelő függvények
  - Idő lekérdezés
  - Atomikus függvények
  - Szavazó függvények
  - Változók cseréje szálak között
  - Formázott kiíró függvény (printf a kernel kódban!)
  - Memória foglalás
  - Kernel indítás

NSight

- Fejlesztést segítő eszköz
  - Visual Studio 2008,2010,2012
  - Eclipse
- Memória ellenőrzés
- Hibakeresés
  - CUDA kódban
  - OpenGL vagy DirectX shaderben
- Teljesítmény analízis

NSight

- Kernel változók értéke
- Töréspontok a kernelen belül
- Szálak közötti váltás

```
__global__ void addKernel(int *c, const int *a, const int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
```

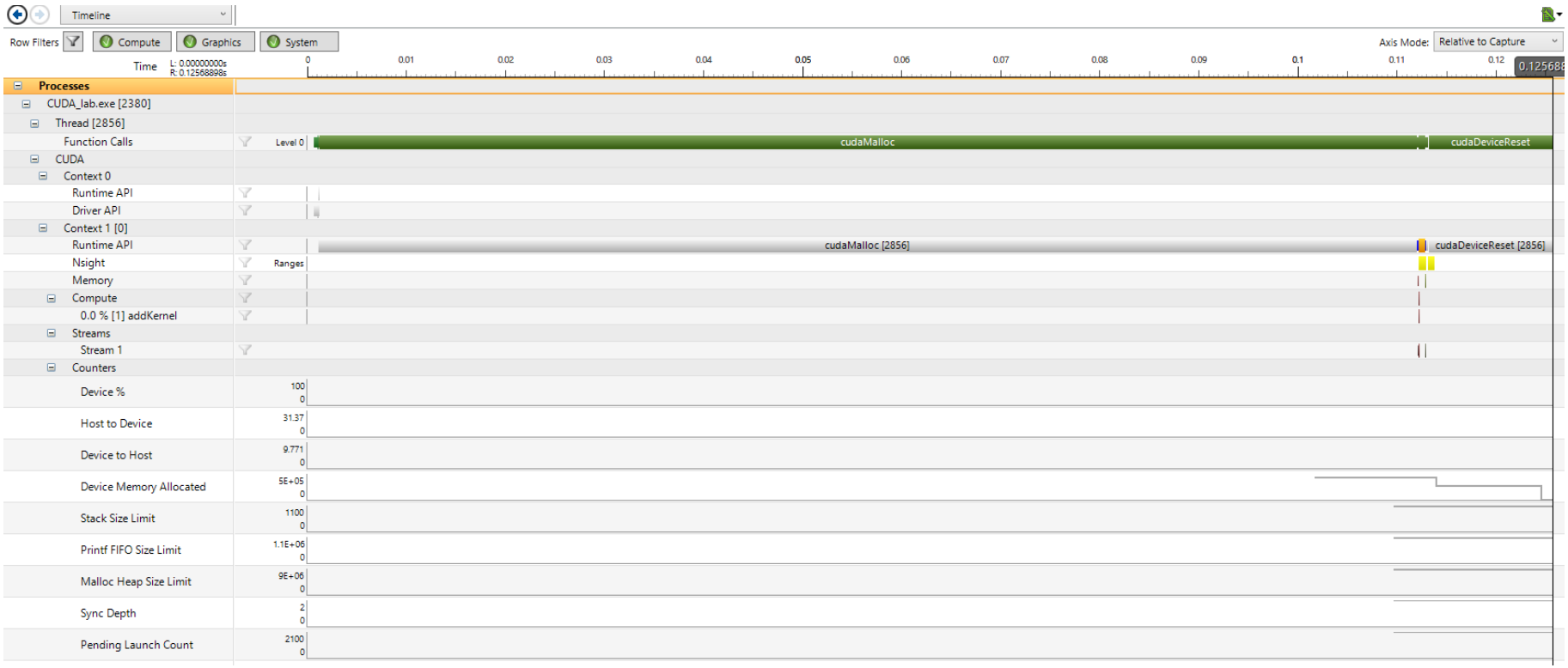
100 %

Autos

Name	Value	Type
a	0x42260200 1	__device__ const int* __parameter__
[0]	1	__device__ const int&
a[i]	1	__device__ const int&
b	0x42260400 10	__device__ const int* __parameter__
[0]	10	__device__ const int&
b[i]	10	__device__ const int&
c	0x42260000 0	__device__ int* __parameter__
[0]	0	__device__ int&
c[i]	0	__device__ int&
i	0	int

NSight

- Teljesítmény analízis



Visual Profiler

- Teljesítmény analízis és értékelés