



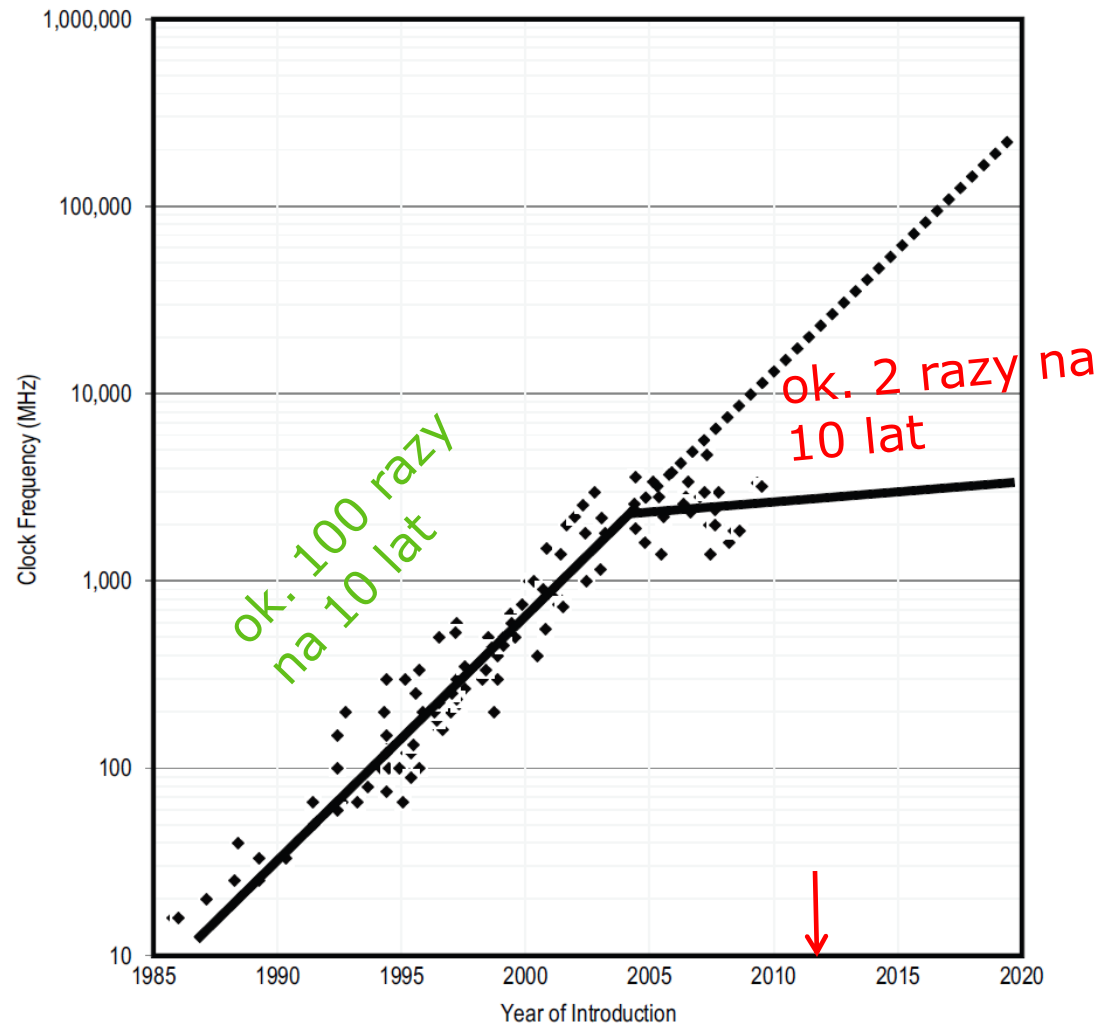
Uniwersytet
Wrocławski

CUDA jako platforma GPGPU w obliczeniach naukowych

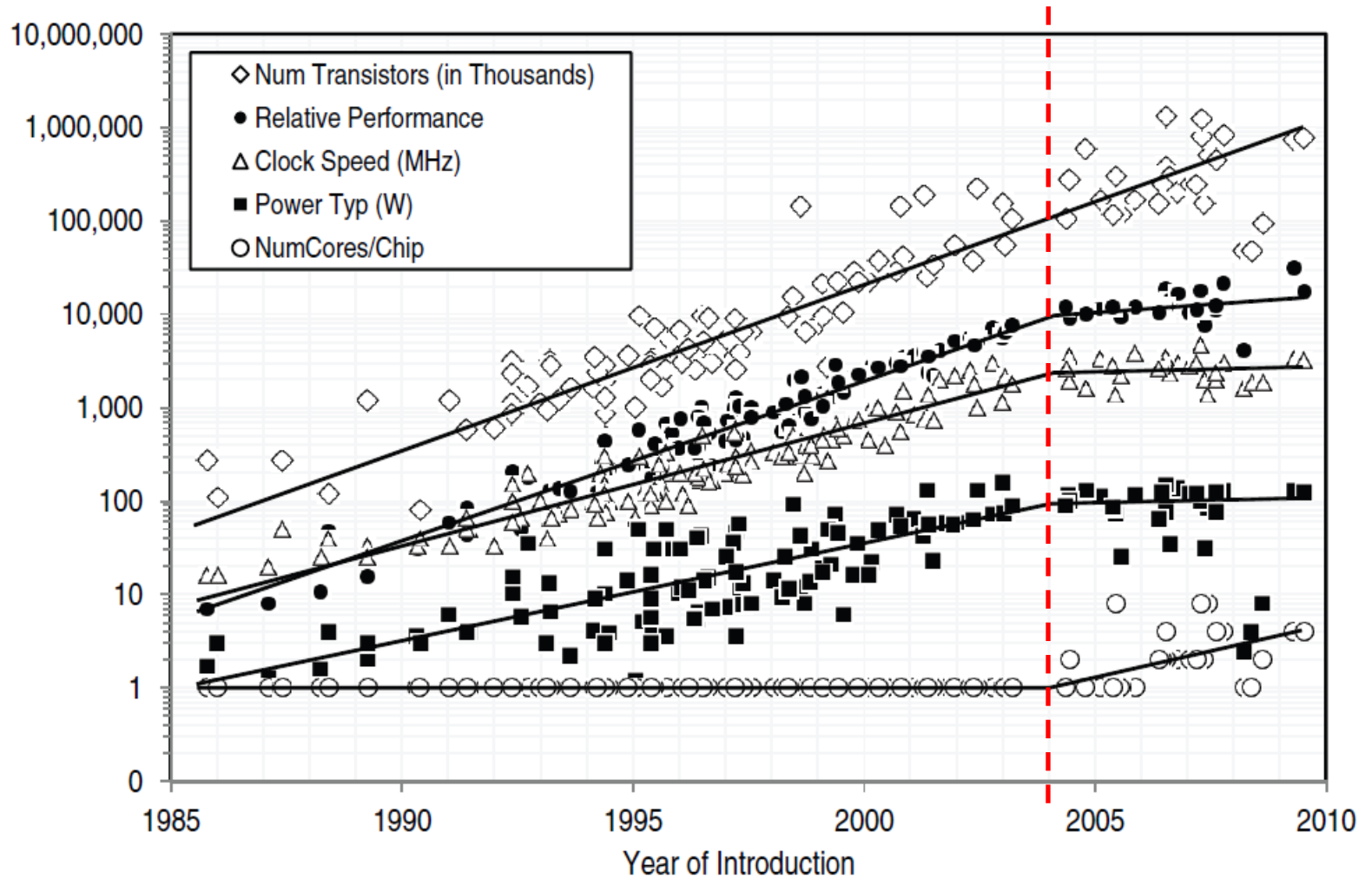
Seminarium Grupy Neutrinowej, 12.12.2011

Maciej Matyka, Zbigniew Koza
Instytut Fizyki Teoretycznej
Uniwersytet Wrocławski

Bariery sprzętowe (procesory)

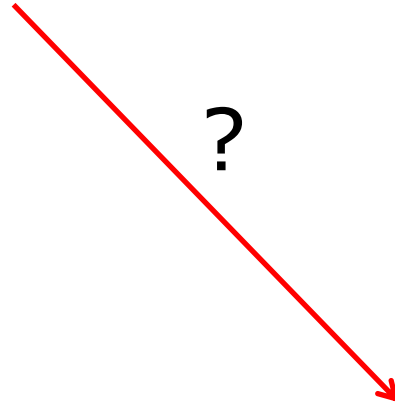


Rozwój 1985-2004 i dalej...?



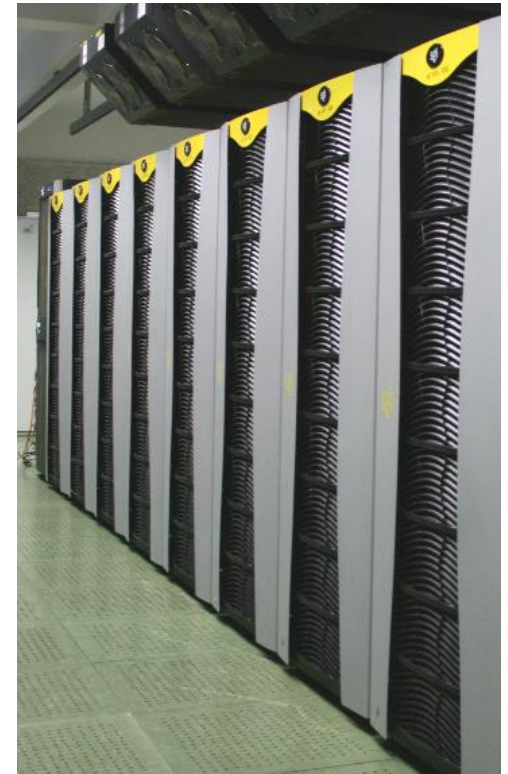


?



Superkomputer:

- Altix 3700, 128 procesorów (Intel Itanium 2)
- Używana np. w Cyfronet AGH
- **0.768 TFLOPS** mocy obliczeniowej
- Cena: **1/2 miliona zł !**



Karta graficzna:



- GeForce 285
(używana w pokoju 521 w IFT)
- 240 procesory CUDA
- **1.06 TFLOPS** mocy obliczeniowej (teoretycznie)
- Cena: **670zł**

- **1.38x** więcej mocy za **0.134%** ceny
- (+ energia + pomieszczenia + chłodzenie etc.)
- Zasada zachowania trudności, czyli coś za coś...
- Kartę trzeba jeszcze oprogramować!

GPGPU

General-purpose computing on graphics processing units

Gdzie umiejscowić GPGPU ?

- SISD – Single Instruction Single Data
- **SIMD – Single Instruction Multiple Data (MSIMD?)**
- SPMD – Single Process Multiple Data
- MISD – Multiple Instruction Single Data
- MIMD - Multiple Instruction Multiple Data

Niskopoziomowe API



- Vertex i Pixel Shaders (OpenGL)
- Direct Compute (DirectX)
- CUDA driver API
- **C runtime for CUDA**
- OpenCL
- Kompilatory PGI
- OpenACC



API wysokiego poziomu

Co to jest CUDA?

Compute **U**nified **D**evice **A**rchitecture

- Platforma programowania GPU (dla kart nVidii)
- Rozszerzenie języka C
- (działa również z C++)
- kompilator (nvcc)
- zbiór dyrektyw kompilatora i dodatkowych funkcji
- Np. słowa kluczowe:
 __device__, **__host__** wyróżniające rodzaj sprzętu
 na którym implementowana jest funkcja lub dana
- Np. funkcja:
 cudaMemcpy (...)
(kopiowanie pamięci CPU <-> GPU)

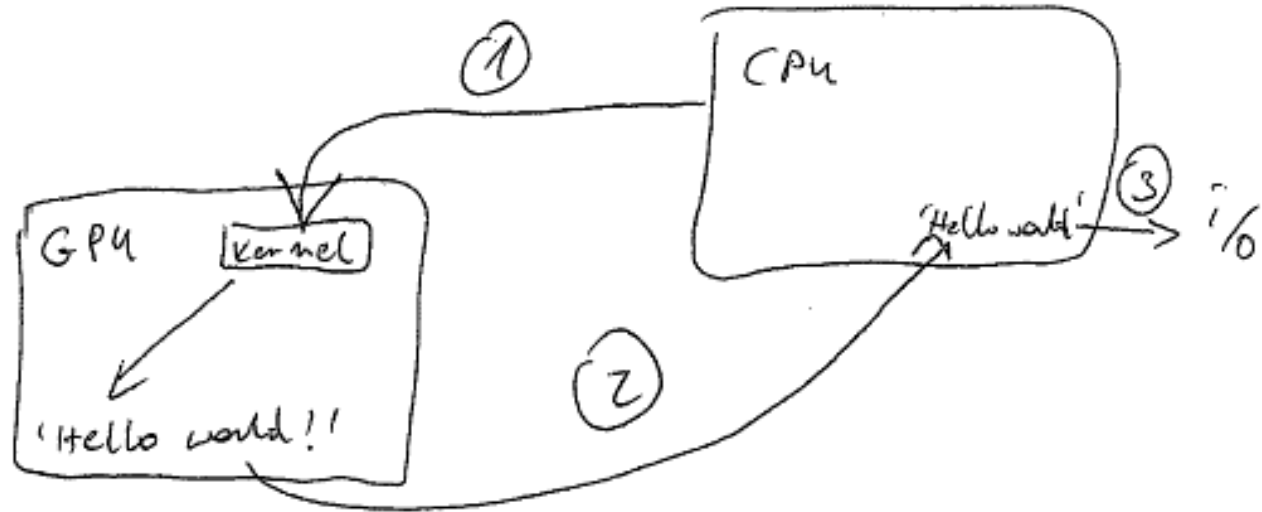


- Microsoft Visual C++ (2008, 2010)
 - można użyć darmowej wersji Express
- Sterownik (ze strony Parallel NSIGHT):
- CUDA Toolkit
- CUDA SDK
- nVidia Parallel NSIGHT HOST/MONITOR



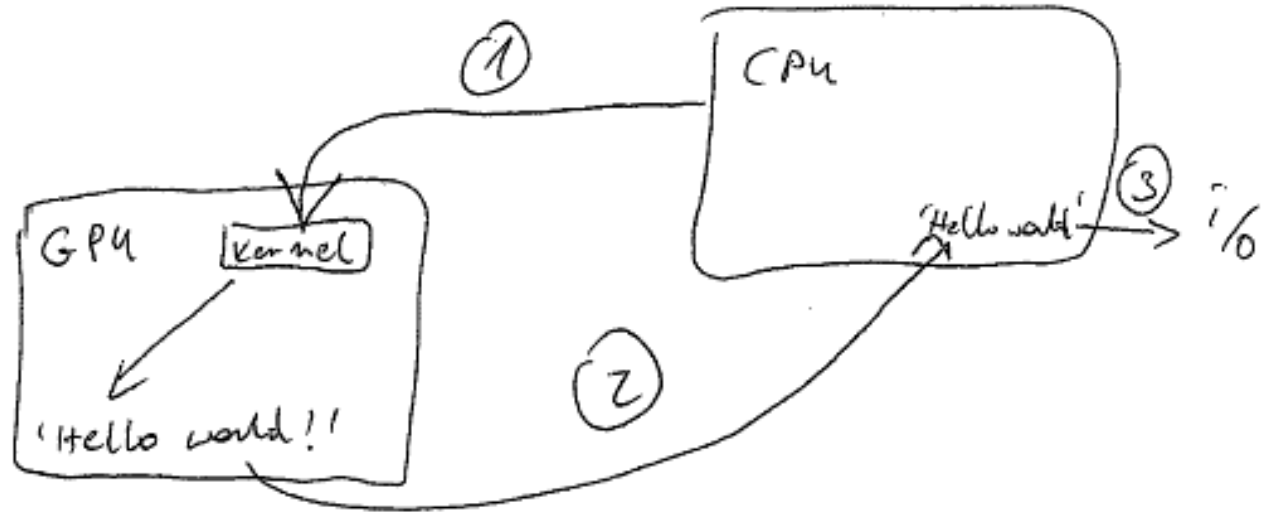
- Inaczej niż pod Windows
– nie używamy Parallel Nsight
- Sterowniki **nvidia-current** (repozytoria)
- Cuda Toolkit (biblioteki, pliki nagłówkowe, .so)
- Po zainstalowaniu kompilacja (plik program.cu):
 - > **nvcc program.cu**
 - > **./a.out**

- Pierwszy prosty program w CUDA



1. CPU wywołuje funkcję na GPU
 - Funkcja wstawia do pamięci GPU ciąg „Hello world!”
2. Kopiujemy dane z GPU do pamięci CPU
3. Dane wypisujemy przy pomocy funkcji i/o (cout etc.)

- Pierwszy prosty program w CUDA



1. CPU wywołuje funkcję na GPU
 - Funkcja wstawia do pamięci GPU ciąg „Hello world!”
2. Kopiujemy dane z GPU do pamięci CPU
3. Dane wypisujemy przy pomocy funkcji i/o (cout etc.)

Hello World

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

__device__ char napis_device[14];

__global__ void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    ...
    napis_device[11] = '!';
    napis_device[12] = '\n';
    napis_device[13] = 0;
}

int main(void)
{
    helloWorldOnDevice <<< 1, 1 >>> ();

    char napis_host[14];
    const char *symbol="napis_device";

    cudaMemcpyFromSymbol (napis_host,
        symbol, sizeof(char)*13, 0,
        cudaMemcpyDeviceToHost);

    printf("%s", napis_host);
}
```


Hello World

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

__device__ char napis_device[14];

__global__ void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    ...
    napis_device[11] = '!';
    napis_device[12] = '\n';
    napis_device[13] = 0;
}

int main(void)
{
    helloWorldOnDevice <<< 1, 1 >>> ();

    char napis_host[14];
    const char *symbol="napis_device";

    cudaMemcpyFromSymbol (napis_host,
        symbol, sizeof(char)*13, 0,
        cudaMemcpyDeviceToHost);

    printf("%s", napis_host);
}
```

Hello World

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

__device__ char napis_device[14];

__global__ void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    ...
    napis_device[11] = '!';
    napis_device[12] = '\n';
    napis_device[13] = 0;
}

int main(void)
{
    helloWorldOnDevice <<< 1, 1 >>> ();

    char napis_host[14];
    const char *symbol="napis_device";

    cudaMemcpyFromSymbol (napis_host,
        symbol, sizeof(char)*13, 0,
        cudaMemcpyDeviceToHost);

    printf("%s", napis_host);
}
```

Hello World

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

__device__ char napis_device[14];

__global__ void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    ...
    napis_device[11] = '!';
    napis_device[12] = '\n';
    napis_device[13] = 0;
}

int main(void)
{
    helloWorldOnDevice <<< 1, 1 >>> ();

    char napis_host[14];
    const char *symbol="napis_device";

    cudaMemcpyFromSymbol (napis_host,
        symbol, sizeof(char)*13, 0,
        cudaMemcpyDeviceToHost);

    printf("%s", napis_host);
}
```

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

__device__ char napis_device[14];

__global__ void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    ...
    napis_device[11] = '!';
    napis_device[12] = '\n';
    napis_device[13] = 0;
}

int main(void)
{
    helloWorldOnDevice <<< 1, 1 >>> ();

    char napis_host[14];
    const char *symbol="napis_device";

    cudaMemcpyFromSymbol (napis_host,
        symbol, sizeof(char)*13, 0,
        cudaMemcpyDeviceToHost);

    printf("%s",napis_host);
}
```

Hello World

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

__device__ char napis_device[14];

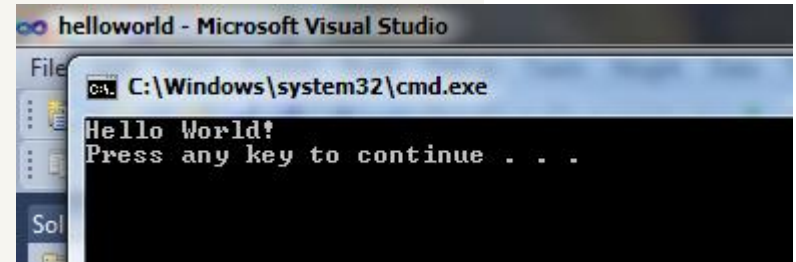
__global__ void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    ...
    napis_device[11] = ' ';
    napis_device[12] = 'W';
    napis_device[13] = 'o';
}

int main(void)
{
    helloWorldOnDevice <<< 1, 1 >>> ();

    char napis_host[14];
    const char *symbol="napis_device";

    cudaMemcpyFromSymbol (napis_host,
        symbol, sizeof(char)*13, 0,
        cudaMemcpyDeviceToHost);

    printf("%s", napis_host);
}
```



Deklaracja funkcji na GPU

- Funkcja uruchamiana na GPU to kernel (jądro)
- Zmienne na GPU – przedrostek **__device__**
- Preambuła jądra - przedrostek **__global__**

```
char napis_device[14];

void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    ...
    napis_device[11] = '!';
    napis_device[12] = '\n';
    napis_device[13] = 0;
}
```

CPU

```
__device__ char napis_device[14];

__global__ void helloWorldOnDevice(void)
{
    napis_device[0] = 'H';
    napis_device[1] = 'e';
    ...
    napis_device[11] = '!';
    napis_device[12] = '\n';
    napis_device[13] = 0;
}
```

GPU



- Wywołanie funkcji GPU:

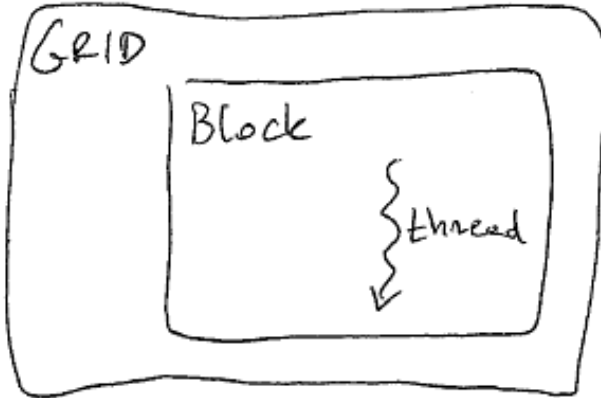
funkcja `<<< numBlocks, threadsPerBlock >>> (parametry);`

- `numBlocks` – liczba bloków w macierzy wątków
- `threadsPerBlock` – liczba wątków na blok

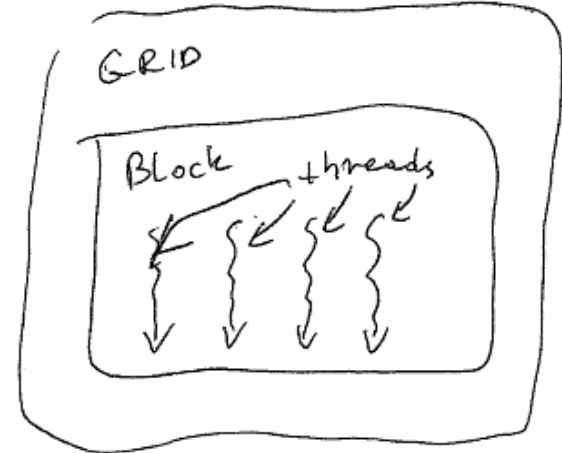
W naszym przykładzie `<<<1,1>>>` oznaczało uruchomienie jądra na jednym wątku który zawierał się w jednym jedynym bloku macierzy wątków.



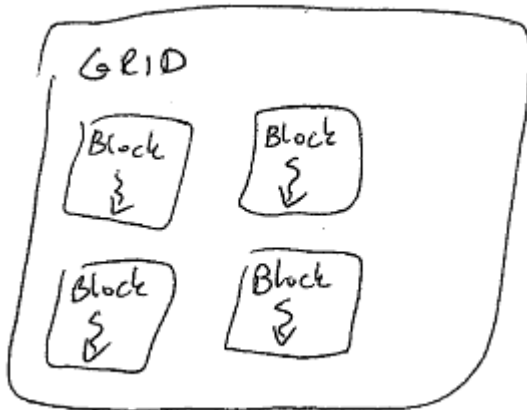
numBlocks, threadsPerBlock



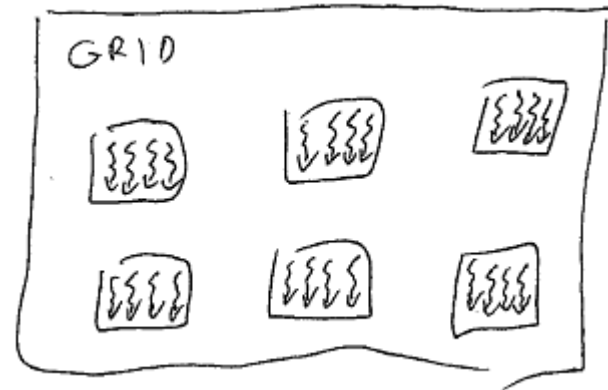
<<< 1, 1 >>>



<<< 1, 4 >>>



<<< 4, 1 >>>



<<< dim3(3,2,1), 4 >>>

CUDA Programming Guide 3.2

```
<<< dim3(3,2,1) , dim3(4,3,1) >>>
```

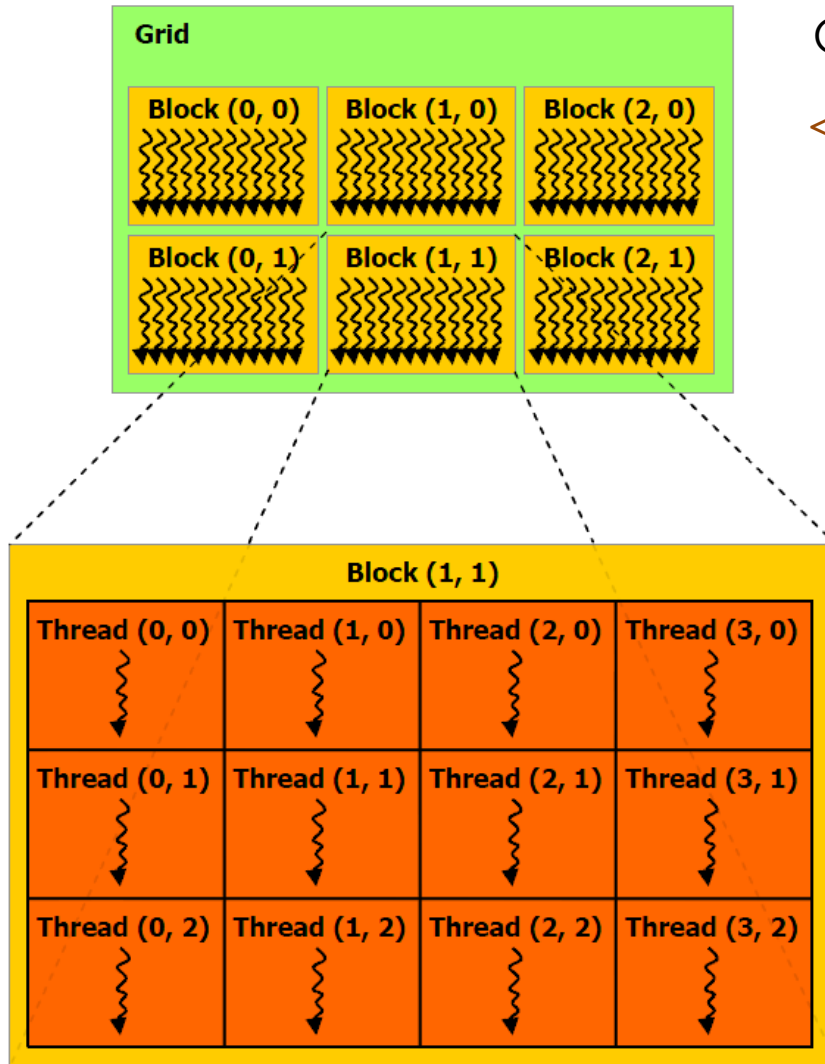
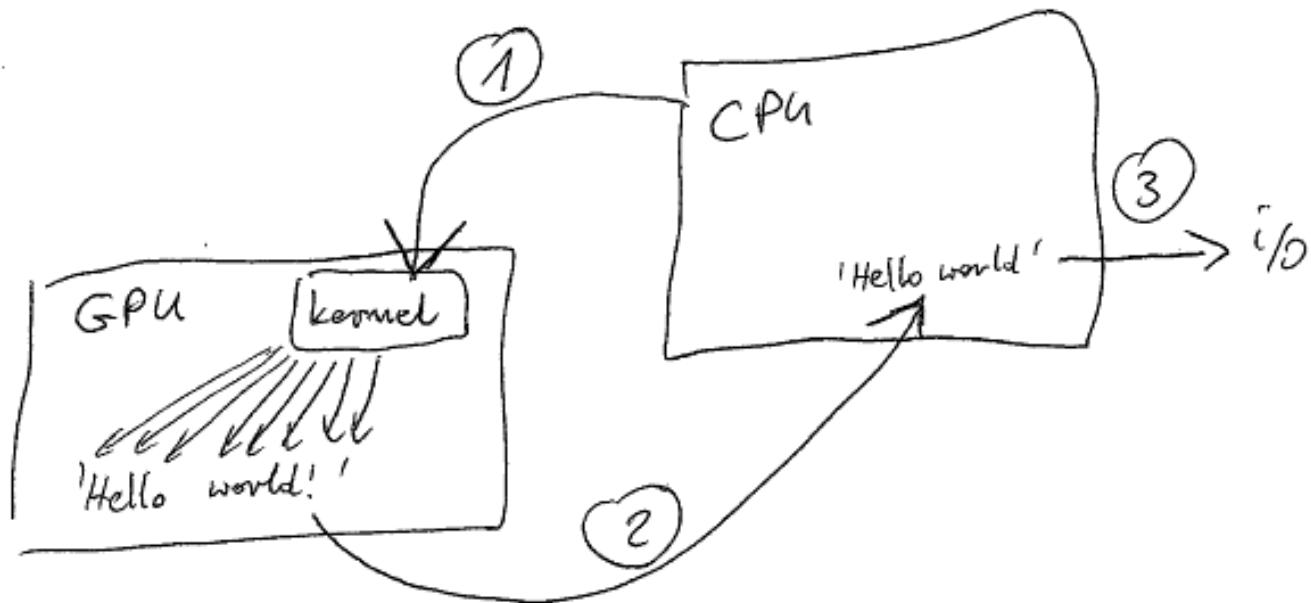


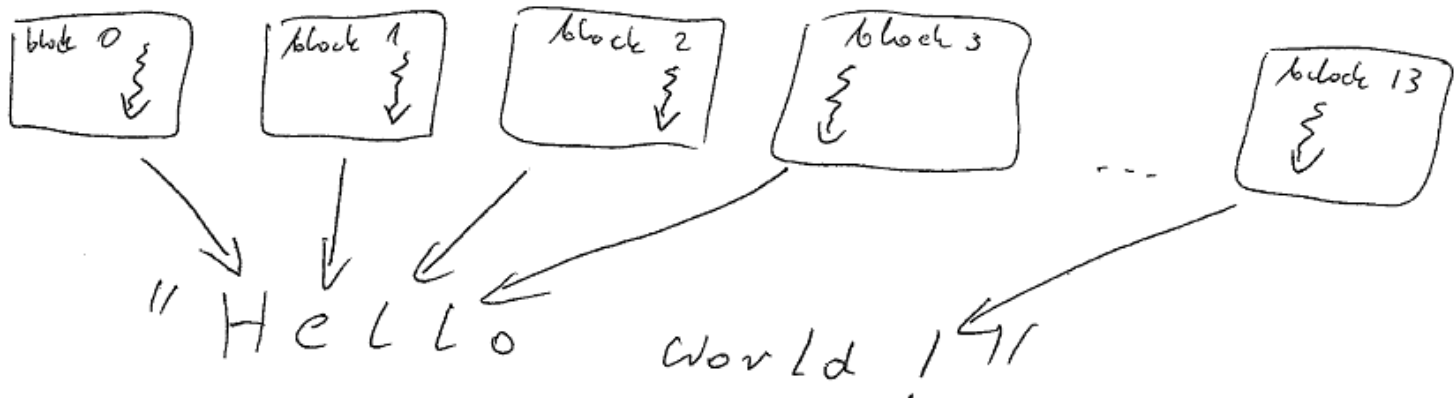
Figure 2-1. Grid of Thread Blocks



1. CPU wywołuje funkcję na GPU
 - Funkcja wstawia do pamięci GPU ciąg „Hello world!”
 - jedna litera na jeden wątek!

Hello World wielowątkowo

- jedna litera na jeden wątek



- stwórzmy 14 bloków po 1 wątku
- Każdy wątek „widzi” swój numer i numer bloku
- Numer bloku -> miejsce w tablicy do skopiowania

Hello World wielowątkowo

- Jądro dla hello world w wersji wielowątkowej:

```
__constant__ __device__ char hw[] = "Hello World!\n";
__device__ char napis_device[14];

__global__ void helloWorldOnDevice(void)
{
    int idx = blockIdx.x;
    napis_device[idx] = hw[idx];
}

...

helloWorldOnDevice <<< 14, 1 >>> ();
```

- idx
 - zawiera numer bloku w którym znajduje się wątek
 - mapowanie blok/wątek -> fragment problemu
 - wątek z idx-ego bloku kopiuje idx-ą literę napisu
 - Kopiowanie GPU-GPU
 - (bez sensu, ale chodzi nam o najprostszy problem)



Hello World wielowątkowo

- Jądro dla hello world w wersji wielowątkowej:

```
__constant__ __device__ char hw[] = "Hello World!\n";
__device__ char napis_device[14];

__global__ void helloWorldOnDevice(void)
{
    int idx = blockIdx.x;
    napis_device[idx] = hw[idx];
}

...

helloWorldOnDevice <<< 14, 1 >>> ();
```

- idx
 - zawiera numer bloku w którym znajduje się wątek
 - mapowanie blok/wątek -> fragment problemu
 - wątek z idx-ego bloku kopiuje idx-ą literę napisu
 - Kopiowanie GPU-GPU
 - (bez sensu, ale chodzi nam o najprostszy problem)



Hello World wielowątkowo

- Jądro dla hello world w wersji wielowątkowej:

```
__constant__ __device__ char hw[] = "Hello World!\n";
__device__ char napis_device[14];

__global__ void helloWorldOnDevice(void)
{
    int idx = blockIdx.x;
    napis_device[idx] = hw[idx];
}

...

helloWorldOnDevice <<< 14, 1 >>> ();
```

- idx
 - zawiera numer bloku w którym znajduje się wątek
 - mapowanie blok/wątek -> fragment problemu
 - wątek z idx-ego bloku kopiuje idx-ą literę napisu
 - Kopiowanie GPU-GPU
 - (bez sensu, ale chodzi nam o najprostszy problem)



- Jądro dla hello world w wersji wielowątkowej:

```
__constant__ __device__ char hw[] = "Hello World!\n";
__device__ char napis_device[14];

__global__ void helloWorldOnDevice(void)
{
    int idx = blockIdx.x;
    napis_device[idx] = hw[idx];
}

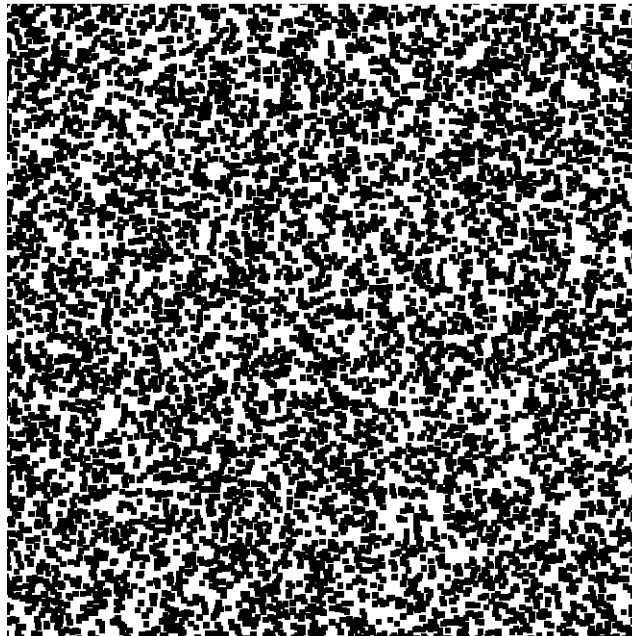
...

helloWorldOnDevice <<< 14, 1 >>> ();
```

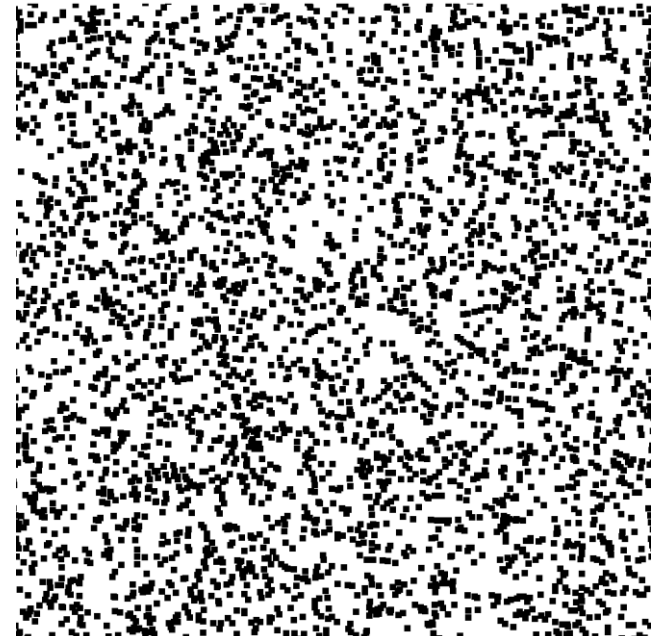
- idx
 - zawiera numer bloku w którym znajduje się wątek
 - mapowanie blok/wątek -> fragment problemu
 - wątek z idx-ego bloku kopiuje idx-ą literę napisu
 - Kopiowanie GPU-GPU
 - (bez sensu, ale chodzi nam o najprostszy problem)



W praktyce

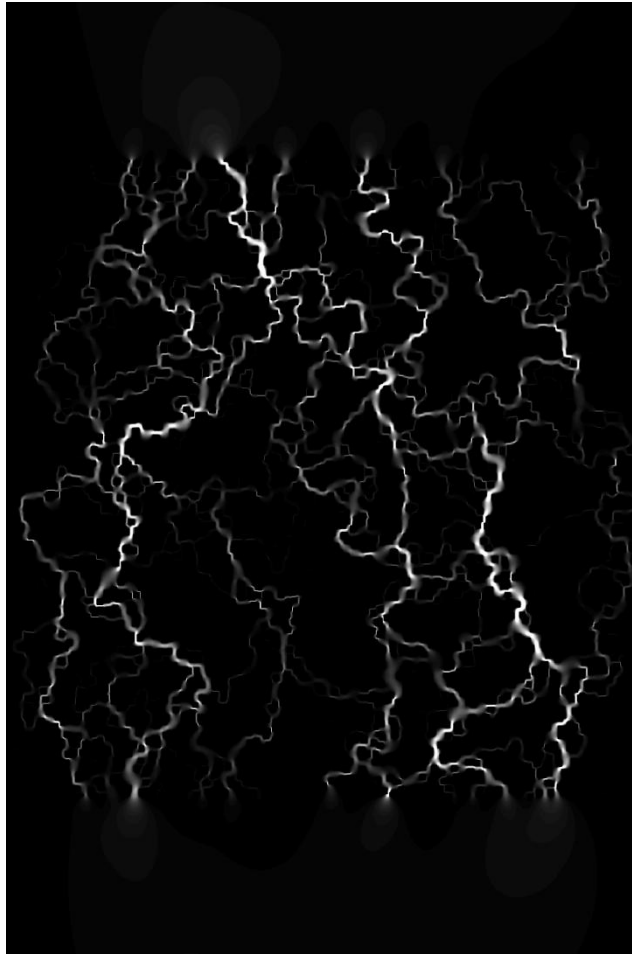


$P=0.4$

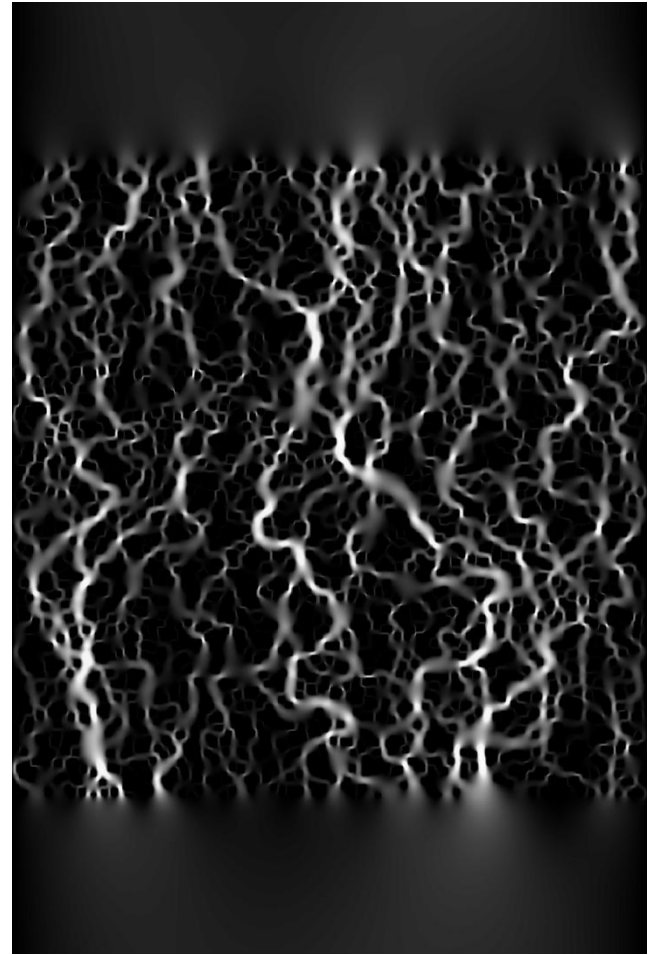


$P=0.7$

Ogromne układy (miliony komórek obliczeniowych).

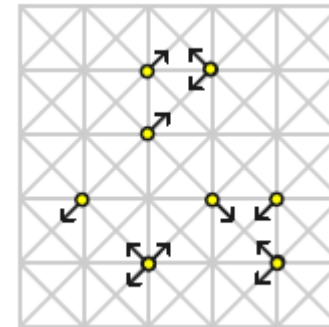
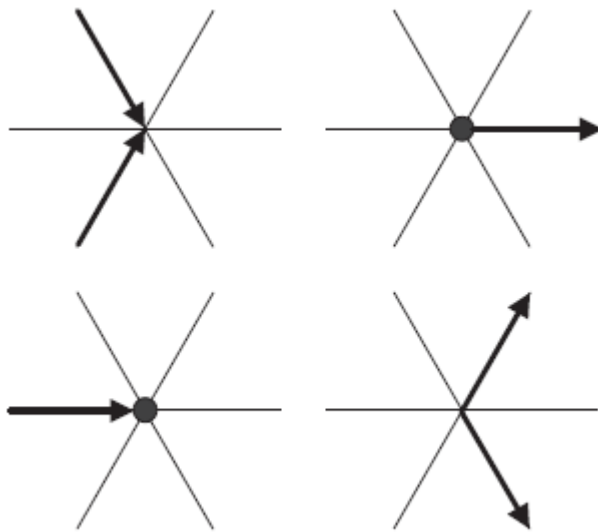


$P=0.4, T=1.7$



$P=0.7, T=1.28$

- Uproszczenie: model gazu sieciowego opisany przez U. Frisha, B. Hasslachera i Y. Pomeau gaz FHP



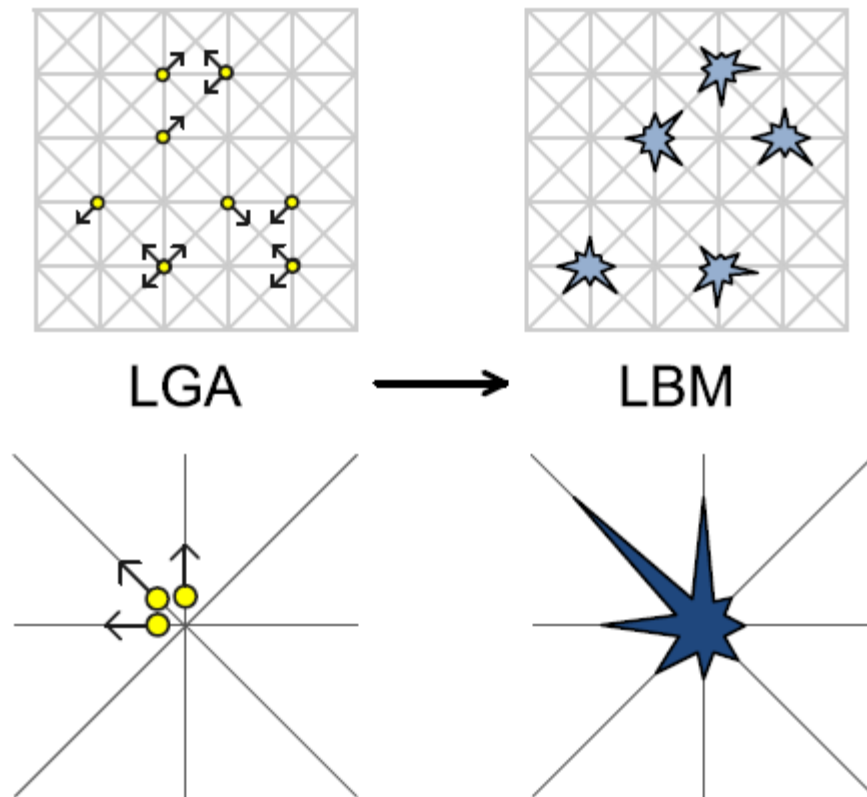
- 1) Transport
- 2) Kolizje

Rys. Kolizje w modelu FHP5

U. Frish, B. Hasslacher, Y. Pomeau 1986 *Lattice-Gas Automata for the Navier-Stokes Equation*, Phys. Rev. Lett. 56, 1505–1508.

Matyka, M. and Koza, Z., *Spreading of a density front in the Kuentz-Lavallee model of porous media* J. Phys. D: Appl. Phys. 40, 4078-4083 (2007)

- Historycznie wprowadzona jako rozwinięcie LGA



- Zmienne n_i $(0,1)$ zastąpione funkcją rozkładu $f_i \in [0, 1]$

Podstawowy algorytm LBM to dwa kroki:

- kolizji:

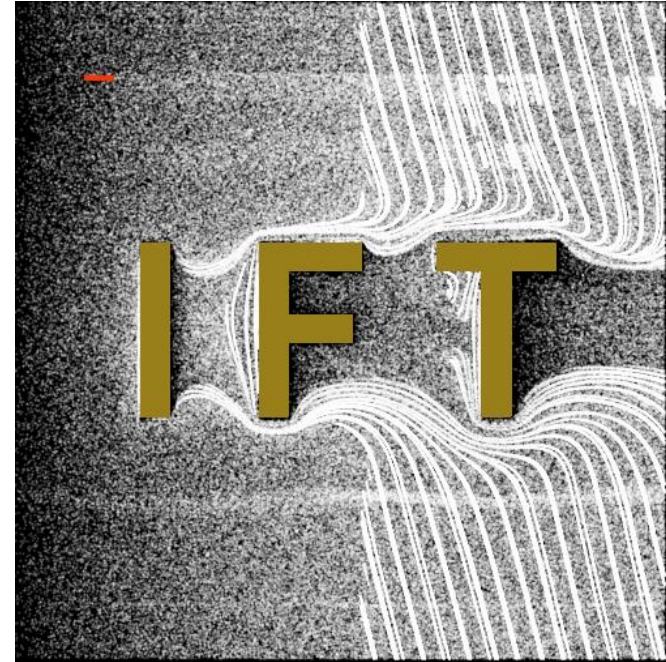
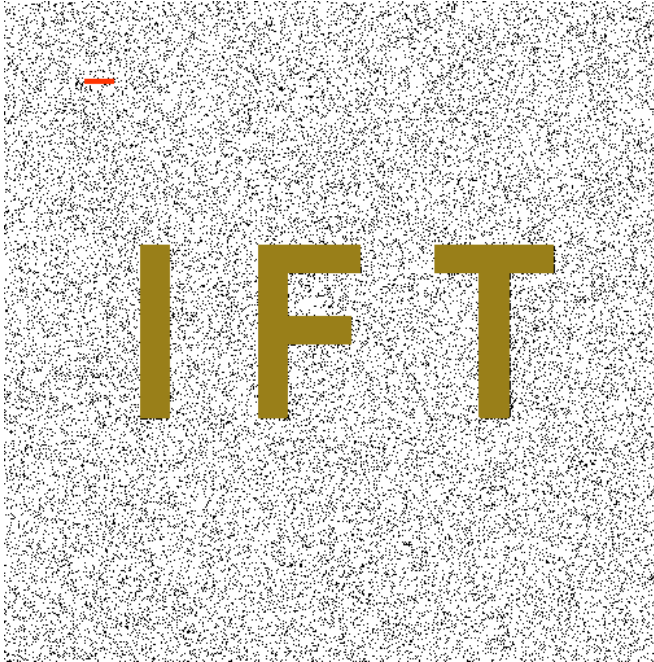
$$\tilde{f}_i(\mathbf{x}, t) = f_i(\mathbf{x}, t) - \frac{1}{\tau}(f_i(\mathbf{x}, t) - f_i^{eq}(\mathbf{x}, t))$$

- transportu:

$$f_i(\mathbf{x} + \mathbf{c}_i, t + 1) = \tilde{f}_i(\mathbf{x}, t).$$

Lokalność!

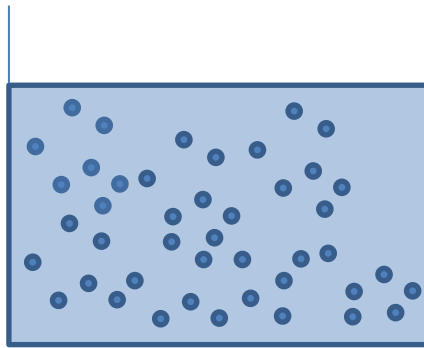
- 512 x 512



- CPU stoi....

GPU w tym czasie..

(przykład [cpu](#), [gpu](#))



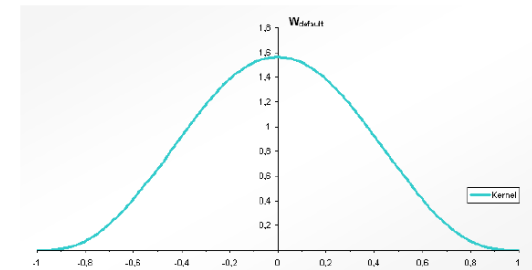
1. Lagrangian Fluid Dynamics, Using Smoothed Particle Hydrodynamics, Micky Kelger, 2006
2. <http://pl.wikipedia.org/wiki/SPH>

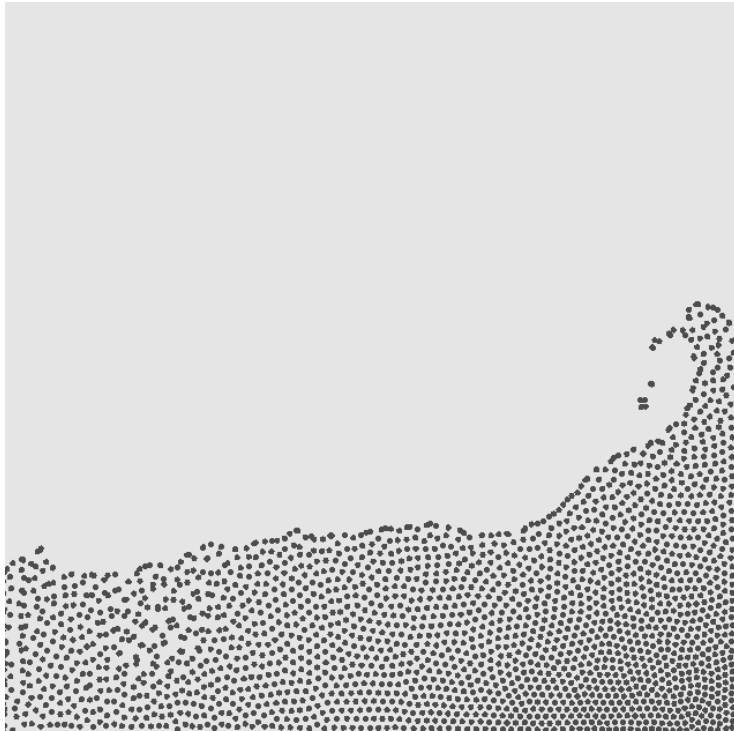
- Podejście **Lagrange'a** (ruchome punkty)
- Każda cząstka reprezentuje pewną objętość cieczy
- Wielkości fizyczne interpolowane po sąsiadach

$$A_i = \sum_{j=1}^N m_j \frac{A_j}{\rho_j} W_{ij}$$

- Gdzie W_{ij} jest tytułowym „rozmyciem” np.

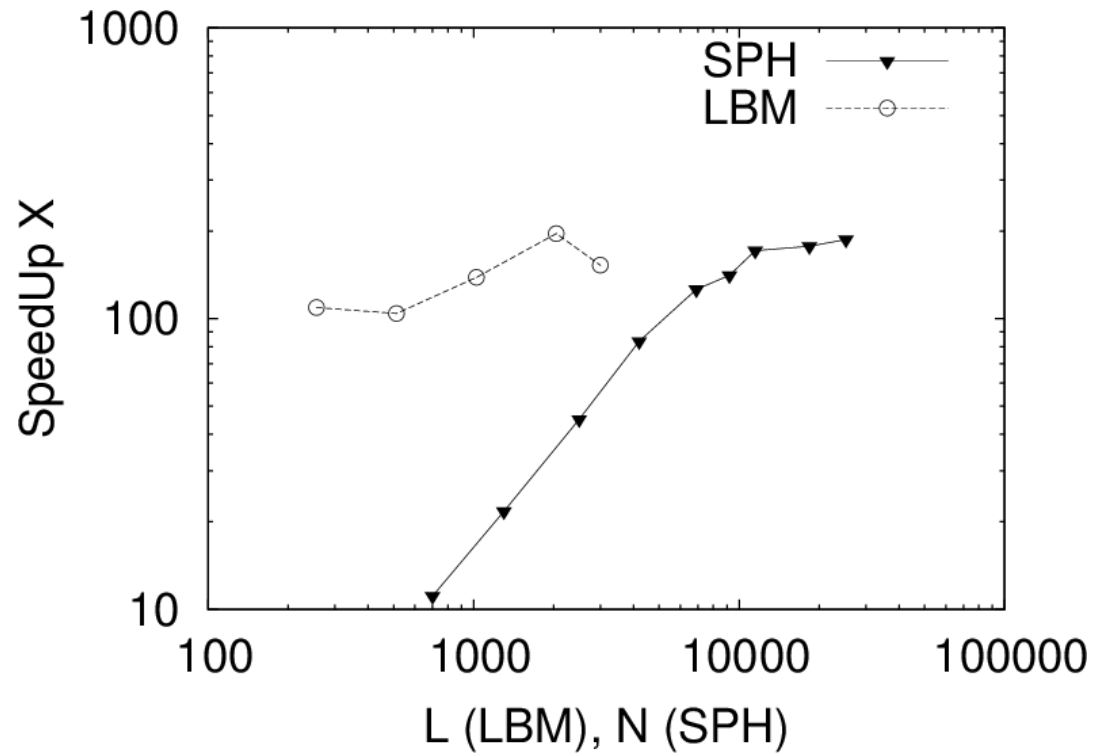
$$W_{default}(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - \|\mathbf{r}\|^2)^3 & 0 \leq \|\mathbf{r}\| \leq h \\ 0 & \|\mathbf{r}\| > h, \end{cases}$$





Rys. Algorytm SPH na CUDA
(GTX285).

(w działaniu)



Rys. Skalowanie algorytmów LBM i SPH na karcie GTX 285 względem jednego rdzenia CPU.

Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU

Victor W Lee[†], Changkyu Kim[†], Jatin Chhugani[†], Michael Deisher[†],
Daehyun Kim[†], Anthony D. Nguyen[†], Nadathur Satish[†], Mikhail Smelyanskiy[†],
Srinivas Chennupati^{*}, Per Hammarlund^{*}, Ronak Singhal^{*} and Pradeep Dubey[†]

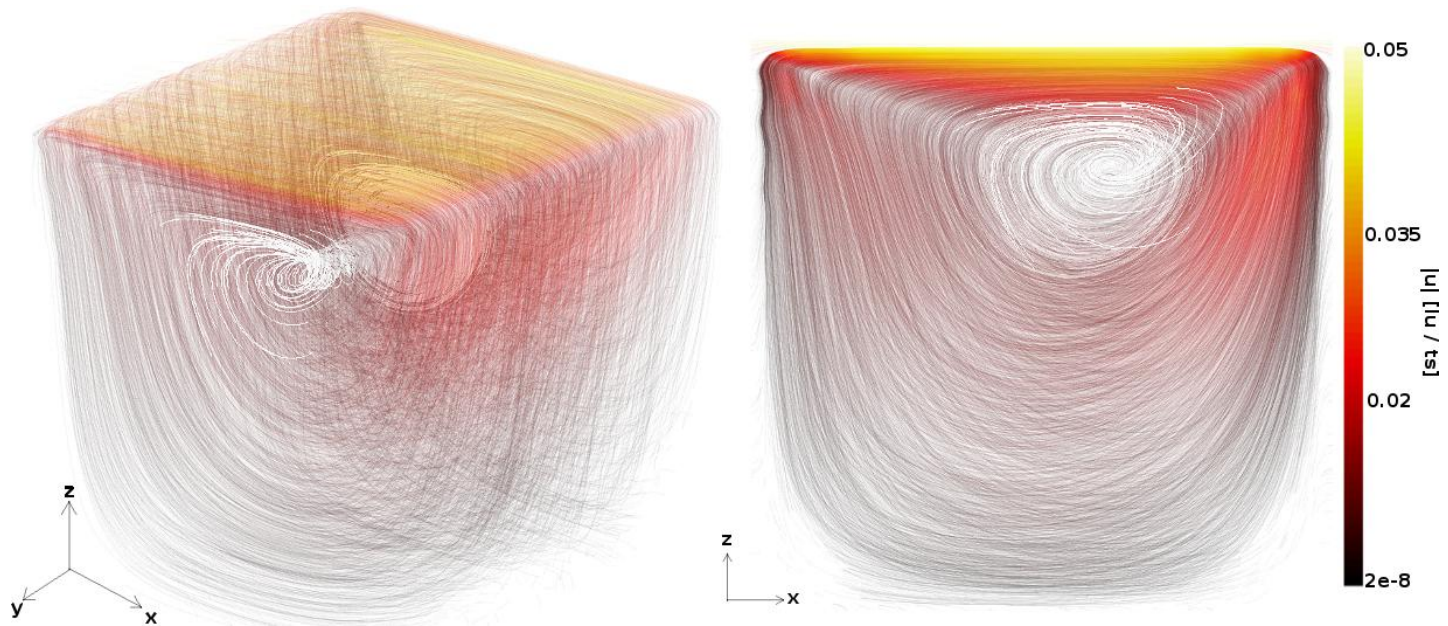
victor.w.lee@intel.com

[†]Throughput Computing Lab,
Intel Corporation

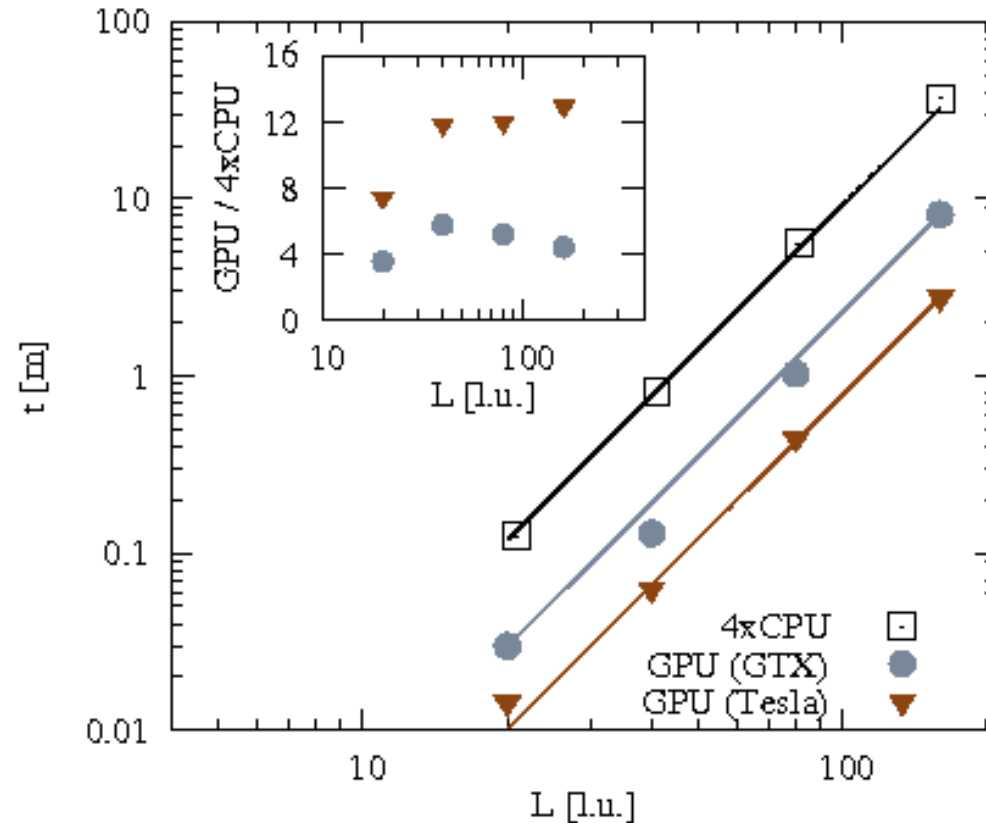
^{*}Intel Architecture Group,
Intel Corporation

„ (...) we perform a rigorous performance analysis and find that after applying optimizations appropriate for both CPUs and GPUs the performance gap between an Nvidia GTX280 processor and the Intel Core i7 960 processor narrows to only **2.5x** on average. (...)”

- Przypadek testowy: driven cavity 3d
- Porównanie czasów określonej liczby iteracji
- Kody: Palabos (MPI) i Sailfish (CUDA)



Rys: Wizualizacja przepływu przez trójwymiarową komorę wyznaczonego kodem **Sailfish** dla liczby Reynoldsa $Re=100$.



Czas wykonania 30000 kroków na 4xCPU i GPU (Gtx460 i Tesla).

Niskopoziomowe API



- Vertex i Pixel Shaders (OpenGL)
- Direct Compute (DirectX)
- CUDA driver API
- C runtime for CUDA
- OpenCL
- Kompilatory PGI
- OpenACC (Listopad 2011) ?



API wysokiego poziomu

- Grant MNiSW Grant N N519 437939
- Projekt Zielony Transfer (Vratis Sp. z o.o.) akceleracja CFD na GPU

Duda, A., Koza, Z. and Matyka, M.

Hydraulic tortuosity in arbitrary porous media flow
Phys. Rev. E 84, 036319 (2011)

Malecha, Z., Mirosław, L., Tomczak, T., Koza, Z., Matyka, M., Tarnawski, W., Szczerba, D., *GPU-based simulation of 3D blood flow in abdominal aorta using OpenFOAM*
Arch. Mech. Vol. 63 No. 2 pp. 137-161 (2011)

Matyka, M., Mirosław, L., Koza, Z.

Performance of open source implementations of the Lattice Boltzmann methods on CPU and GPU (polish), The computer-aided scientific research (KOWBAN) XVIII (2011)

Koza, Z., Matyka, M., Szkoda, S., Mirosław, L., Pola, J., *Application of GPGPU technology to computational science (polish)*, The computer-aided scientific research (KOWBAN) XVIII (2011)

Dziękuję za uwagę