Programowanie CUDA
informacje praktycznie i przykłady problemów obliczeniowych

Wersja z 14.12.2016

Podstawowe operacje na urządzeniu GPU

cudaSetDevice()

- Określenie GPU i ustanowienie „kontekstu” dla GPU wykorzystywanego przez wątek CPU. (Analog w GPU obiektu takiego jak proces dla CPU.) Alokacje pamięci DEVICE I i uruchomienia kernela będą realizowane na wybranym urządzeniu, z nim związane są strumienie i zdarzenia. Domyślnie wybrane jest urządzenie o devID =0.

cudaGetDevice(&devID)

- służy do sprawdzenia efektu wykonania cudaSetDevice(), zwraca ewentualnie błąd.

cudaDeviceReset()

- usuwa „kontekst” GPU używany przez wątek, wywołanie **niezbędne do zakończenie profilowania**.

Pomiar czasu obliczeń kernela

```
cudaEvent_t start;
error = cudaEventCreate(&start);
   if (error != cudaSuccess)
   {   fprintf(stderr, „Nie udało się utworzyć zdarzenia start (kod bledu %s)!\n“,  cudaGetErrorString(error));
      exit(EXIT_FAILURE);
   }
cudaEvent_t stop;
error = cudaEventCreate(&stop);    if (error != cudaSuccess) ....
error = cudaEventRecord(start, 0);    if (error != cudaSuccess)....  //zapisywanie zdarzenia  startowego w
      strumieniu zerowym
// uruchomienie kernela
error = cudaEventRecord(stop, 0);    if (error != cudaSuccess) ... //zapisywanie zdarzenia  końcowego w
      strumieniu zerowym
error = cudaEventSynchronize(stop);      if (error != cudaSuccess)...//oczekiwanie na zakończenie
 float msecTotal = 0.0f;
error = cudaEventElapsedTime(&msecTotal, start, stop);    if (error != cudaSuccess) ...
//porządki
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Zwracane wartości są mierzone w oparciu o zegar GPU i dlatego rozdzielczość pomiaru jest niezależna od sprzętu i systemu operacyjnego komputera host.

Informacja o błędach przetwarzania CUDA

Wszystkie wywołania funkcji CUDA (oprócz uruchomienia kernela) zwracają kod błędu typu cudaError_t

cudaError_t cudaGetLastError(void)

- Zwraca kod ostatniego błędu (braku błędu) – do wykorzystania dla wywołań asynchronicznych CUDA i wywołań kernela

char* cudaGetErrorString(cudaError_t code)

- Zwraca zakończony zerem ciąg znaków opisujący błąd.
- Użycie: printf("%s\n", cudaGetErrorString(cudaGetLastError()));

- Funkcje informacyjne

  cudaGetDeviceCount(),cudaGetDeviceProperties()

Błędy uruchomienia kernela

- Uruchomienie kernela nie zwraca kodu błędu.
- **Pobieranie błędu konfiguracji kernela -** Wywołania cudaPeekAtLastError() lub **cudaGetLastError()** powinny być zrealizowane dla uzyskania informacji o **błędach przeduruchomieniowych** (parametrów, konfiguracji uruchomienia). Dla zapewnienia, że błąd zwracany przez cudaPeekAtLastError() cudaGetLastError() nie pochodzi z wcześniejszych wywołań ważne jest wyzerowanie zmiennej błędu (ustawienie jej na cudaSuccess) lub wywołanie cudaGetLastError() (funkcja zeruje błąd) przed wywołaniem kernela.
- **Pobieranie błędu obliczeń kernela - po synchronizacji / zakończeniu kernela -** Wywołanie kernela jest asynchroniczne względem przetwarzania CPU i dlatego w celu odczytu aktualnych danych o błędzie należy wywołanie zsynchronizować (np. **cudaDeviceSynchronize())** z przetwarzaniem CPU przed wywołaniami pobrania błędu - cudaPeekAtLastError(), **cudaGetLastError()**, aby mieć pewność, że wywołanie kernela się zakończyło i pobierane informacje (błąd lub brak) dotyczą wywołania kernela.

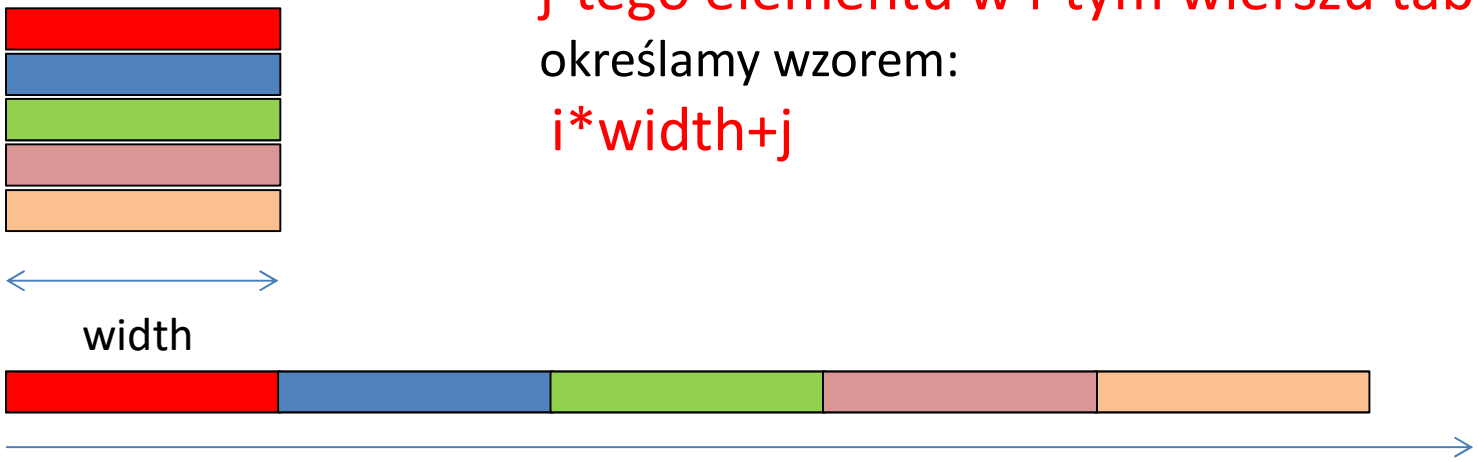
Mnozenie macierzy

Warianty procedur:

1. Jeden blok wątków
2. Wiele bloków wątków (1w/1w)
3. Wykorzystanie pamięci współdzielonej (1w/1w)
4. Zwiększenie ilości przetwarzania między synchronizacjami (1w/1w)
5. Wzrost ilości pracy dla wątku (skalowanie w dół przetwarzania) (1w/Nw)
6. Zrównoleglenie transferów danych host-GPU, GPU-host z przetwarzaniem GPU

Dostęp do elementów tablic

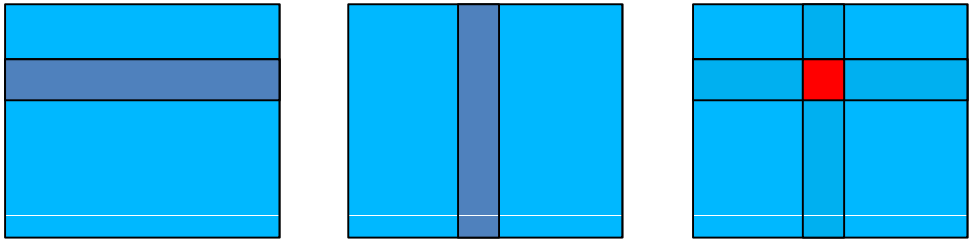
Tablica dwu wymiarowa:
Ze względu na lokację tablic wierszami, pozycję:
j-tego elementu w i-tym wierszu tablicy
określamy wzorem:
i*width+j



Kolejne elementy tablicy w przestrzeni adresowej pamięci karty

Wyznaczanie elementu macierzy

A * B = C



For (k=0, k<width,k++)

C[i*width+j]+=A[i*width+k]*B[k*width+j];

//czyli C[i][j]=A[i][k]*B[k,j];

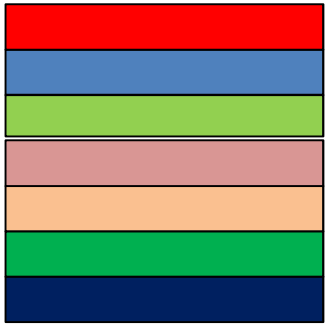
//kod obliczania dla jedengo wątku gdy liczy on jeden wynik

Tworzenie wiązek (warp) z wątków bloku - przykład

Wiązka do 32 wątków używa osnowy czyli Warp do obliczeń.
Stąd wiązka to nazywana jest również jako WARP.
Załóżmy blok wątków o wymiarach 7 na 7 dla CC 1.3

- liczba wątków bloku wynosi 49
- MAX liczba bloków na SM (cc 1.3) = 8
- MAX liczba wątków na SM (cc 1.3) = 1024 = 32x32
- 49 wątków to 2 WARP (zajmują równocześnie 2 osnowy)
- max zajętość SM blokami to 8 bloków (obowiązuje gdy nie brakuje innych zasobów)
- 8 bloków po 2 WARP to 16 WARP
- wtedy (teoretyczna – maksymalna) zajętość SM przez warp'y wynosi 16/32 = 50%,
- zajętość (teoretyczna – maksymalna) SM wątkami 392/1024 = 38 %
- Może zabraknąć wątków do zapewnienia ciągłości obliczeń (długo trwające dostępy do pamięci globalnej)

Blok wątków 7x7 (2 wymiarowy)



Przydział bloku wątków do WARP

Mnożenie macierzy PKG wer.1

```
// funkcja mnożenia macierzy GPU wer.1
__global__ void MatrixMulKernel_1(float* Ad, float* Bd, float* Cd, int WIDTH)
   { // indeksy obliczanego elementu
    int tx = threadIdx.x;                    int ty = threadIdx.y;      float C_local = 0;  //obliczany wynik
      for (int k = 0; k < WIDTH; ++k)
       {                                  float A_d_element = Ad[ty * WIDTH+ k];
// Ad[ty,k] transfer nieefektywny – (potencjalnie) ta sama wartość potrzebna jednocześnie  wątkom ½ warpa

                                          float B_d_element = Bd[k * WIDTH + tx];
// Bd[k,tx] transfer  efektywny  łaczone dostępy  – (potencjalnie) sąsiednie lokacje pamieci dla ½  warpa
         C_local += A_d_element * B_d_element;
         }
                                          Cd[ty * WIDTH + tx] = C_local;
      }
```

- Obliczenia za pomocą jednego bloku wątków (kluczowe są zmienne threadIdx.x, threadIdx.y)
- Lokacja zmiennych
- Każdy wątek oblicza jeden element wyniku – tutaj liczba wątków równa liczbie elementów macierzy (zadanie z laboratorium nie).
- Rozmiar bloku wątków równy rozmiarowi macierzy lub więcej pracy (dodatkowa pętla po obliczanych elementach) dla każdego wątku (ilość pracy zależna od stosunku wielkości tablicy i wielkości bloku)

Uruchomienie przetwarzania

- MM ver.1

```
// parametry konfiguracji uruchomienia kernela
 dim3 wymiaryBloku(WIDTH, WIDTH);
 dim3 wymiaryGridu(1, 1); // jeden blok
// uruchomienie obliczeń
 MatrixMulKernel_1<<< wymiaryGridu,
   wymiaryBloku >>>   (Ad, Bd, Cd,WIDTH);
```

Efektywność - MM ver.1

- Jeden blok - obliczenia wykonywane na jednym SM.
- W każdej chwili obliczany jest maksymalnie tylko jedna wiązka – 32 wątki (GTX260). Dla pozostałych wątków (z innych 32 el. wiązek) możliwe jest pobieranie operandów z pamięci globalnej GPU.
- Maksymalna liczba wątków na SM i wielkość bloku wątków ograniczona przez parametry CC (zdolności obliczeniowej) i zasoby GPU.
- Max wydajność rozwiązania - obliczenia – używamy tylko 1 SM:
  - $1{,}4 * 10^9$ (częstotliwość) 8 X 3 = **30 Gflop**
  - 8 oznacza liczbę rdzeni w SM (używane dla kodu)
  - 3 oznacza liczbę instrukcji na cykl w przeliczeniu na liczbę rdzeni ((8+16)/8)
- Max przepustowość pamięci dla karty: 112 GB/s (GTX260)
- Brak ograniczenia prędkości obliczeń przepustowością dostępu do pamięci.
- Ograniczenie prędkości obliczeń karty wykorzystaniem tylko jednego SM.
- Ograniczenie prędkości obliczeń SM czasem dostępu do pamięci i zbyt małą liczbą wiązek (max 16 wizązek – 512 wątków) równolegle realizujących dostępy do pamieci (potrzeba ok.100 warpów) aby rdzenie były zajęte obliczeniami gdyż CGMA =1.

Analiza ograniczenia prędkości obliczeń MM ver.1

- Współczynnik CGMA (compute to global memory access ratio) – stosunek liczby operacji do liczby dostępów do pamięci.
- Dla mnożenia macierzy 2 operacje (+,*) przypadają na 2 dostępy do pamięci globalnej (pobranie operandów) – CGMA =1
- CGMA wyrażone w bajtach - 2/8 opearacji / bajt

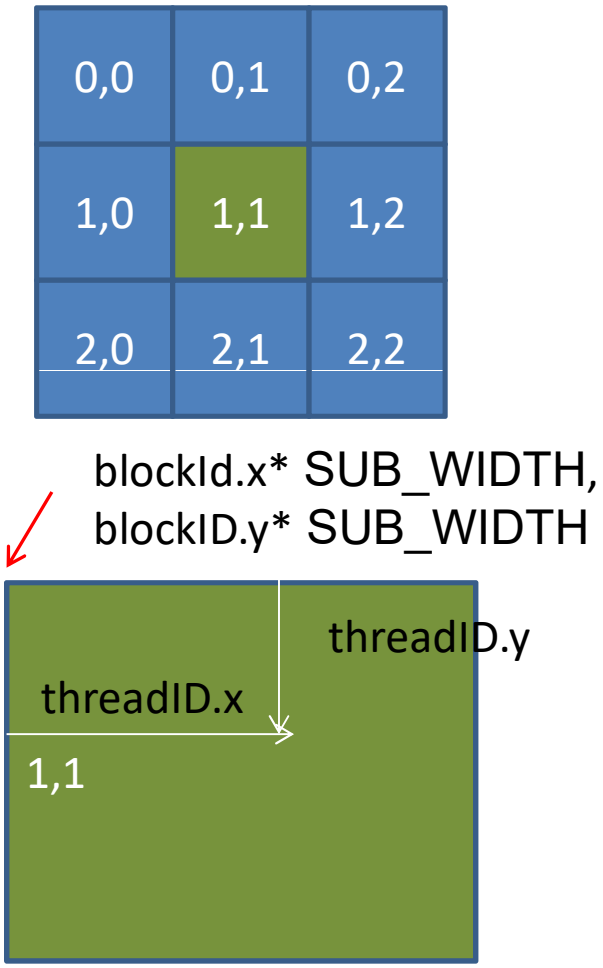
Model przetwarzania:

- Po zakończeniu kroku obliczeń dla jednej wiązki (2 operacje dla 32 watków ok. 4 cykli) przechodzi on do pobierania danych z pamięci globalnej.
- Kolejna praca tego warpu możliwa po odebraniu danych rzędu 100 cykli (czas dostępu do pamięci globalnej).
- Potrzeba 100/4 warpów realizujących obliczenia tego typu CGMA=1 dla zapewnienia ciągłości (efektywności obliczeń). (jest to niemożliwe – w tej wersji kodu - mamy maksymalnie 16 warpów z powodu 1 bloku watków)

Mnożenie macierzy wer.2

Określenie elementu dużej macierzy obliczanego przez wątek bloku wątków

- Zakładamy, że wymiar macierzy WIDTH jest podzielny przez wymiar podmacierzy SUB_WIDTH
- Kwadratowy blok wątków oblicza kwadratowy blok elementów macierzy wynikowej, każdy wątek wyznacza jeden element podmacierzy , SUB_WIDTH= blockDim.x= blockDim.Y
- Dla określenia lewego górnego elementu podtablicy obliczanej przez blok potrzebne są zmienne automatyczne środowiska - indeksy bloków blockID.x , blockID.y
  - blockID.x* blockDim.x, blockID.y* blockDim.y
- Dla wyznaczenia elementu w ramach bloku potrzebne są indeksy wątków w bloku:
  - threadID.x, threadID.y
- Indeksy elementów tablicy zatem dla wątku to:
  - blockID.x* blockDim.x + threadID.x,
  - blockIDd.y* blockDim.y + threadID.y
- Zrzutowanie odwołania na elementy w macierzy jednowymiarowej:
  - (blockID.y* blockDim.y + threadID.y)* WIDTH + (blockID.x* blockDim.x + threadID.x)



Mnożenie macierzy wer2

- Obliczenia za pomocą wielu bloków wątków o wielkości blockDim.x * blockDim.y
- Każdy wątek oblicza jeden element wyniku – liczba wątków równa liczbie elementów macierzy.
- Liczba bloków równa (rozmiar_macierzy/ blockDim.x)^2 - zależna od wielkości macierzy i wielkości bloku
- Obliczenia mogą być wykonywane na wielu SM, gdyż różne bloki nie muszą być uruchomione w tym samym SM.
- Liczba jednocześnie przetwarzanych warp zależy od:
  - liczby SM – ograniczenia sprzętu (GTX 260) jeden warp w jednym SM równoczesnie,
  - Liczby bloków - kolejny SM wymaga kolejnego bloku
  - Liczby aktywnych warp – im więcej warp można przydzielić do SM jednocześnie (ograniczenia zasobowe) tym większa możliwość ukrycia kosztu dostępu do pamięci – zachowania ciągłości obliczeń.

```
__global__ void MatrixMulKernel_2(float* Ad,  float* Bd,  float* Cd,  int WIDTH)
        {
        // wyznaczenie  indeksu wiersza/kolumny  obliczanego elementu  tablicy Cd
        int Row = blockId.y * blockDim.y + threadId.y;
        int Col = blockId.x * blockDim.x + threadId.x;
        float C_local= 0;
        // każdy wątek z bloku oblicza jeden element macierzy
        for (int k = 0; k < WIDTH; ++k)  C_local += A_d[Row][k] * B_d[k][Col];
Cd[Row][Col] = C_local;
        }
```

- A_d[Row][k] dostęp nieefektywny - nie wykorzystuje przepustowości łącza – jednocześnie potrzebna ta sama wartość
- B_d[k][Col] dostęp efektywny – sąsiednie wątki warpu czytają sąsiednie słowa z pamięci globalnej.
- Efektywny zapis do pamięci sąsiednich wartości.

MM wer. 2

Uruchomienie przetwarzania

Mnożenie macierzy wer2

// parametry konfiguracji uruchomienia kernela

```
dim3 wymiaryBloku(SUB_WIDTH, SUB_WIDTH);
dim3 wymiaryGridu(WIDTH / SUB_WIDTH, WIDTH / SUB_WIDTH);
```

(uwaga na liczbę wątków aby ich starczyło (SUFIT) i wszystkie realizowały właściwą pracę – czy wątek ze swoimi współrzędnymi jest w zakresie pracy do wykonania?)

// uruchomienie obliczeń

```
MatrixMulKernel_2<<< wym_Gridu, wym_Bloku >>>(Ad, Bd, Cd,WIDTH);
```

Ograniczenia na poziom równoległości przetwarzania

Ograniczenia wydajnościowe:

1. Dla wielu SM - poziom równoległości zależy od liczby bloków wątków przetwarzających grid.
2. Dla jednego SM – poziom równoległości i efektywność obliczeń w ramach jednego SM zależy od liczby wiązek (warps), które można równocześnie przydzielić do jednego SM. Jest ona ograniczona:
   - liczbą bloków w SM – 8 – problemem są małe bloki
   - liczba warps w SM – 32 - czyli max 1024 wątki (CC 1.3)
   - wymaganiami na liczbę rejestrów bloku wątków (wielkość bloku * liczba rejestrów potrzebnych wątkowi)
3. Im więcej bloków tym możliwe większe zrównoważenie pracy SM-ów.
4. Im więcej wątków w SM (bloków jednocześnie w SM) tym większe szanse zapewnienia ciągłości pracy jednostek wykonawczych.

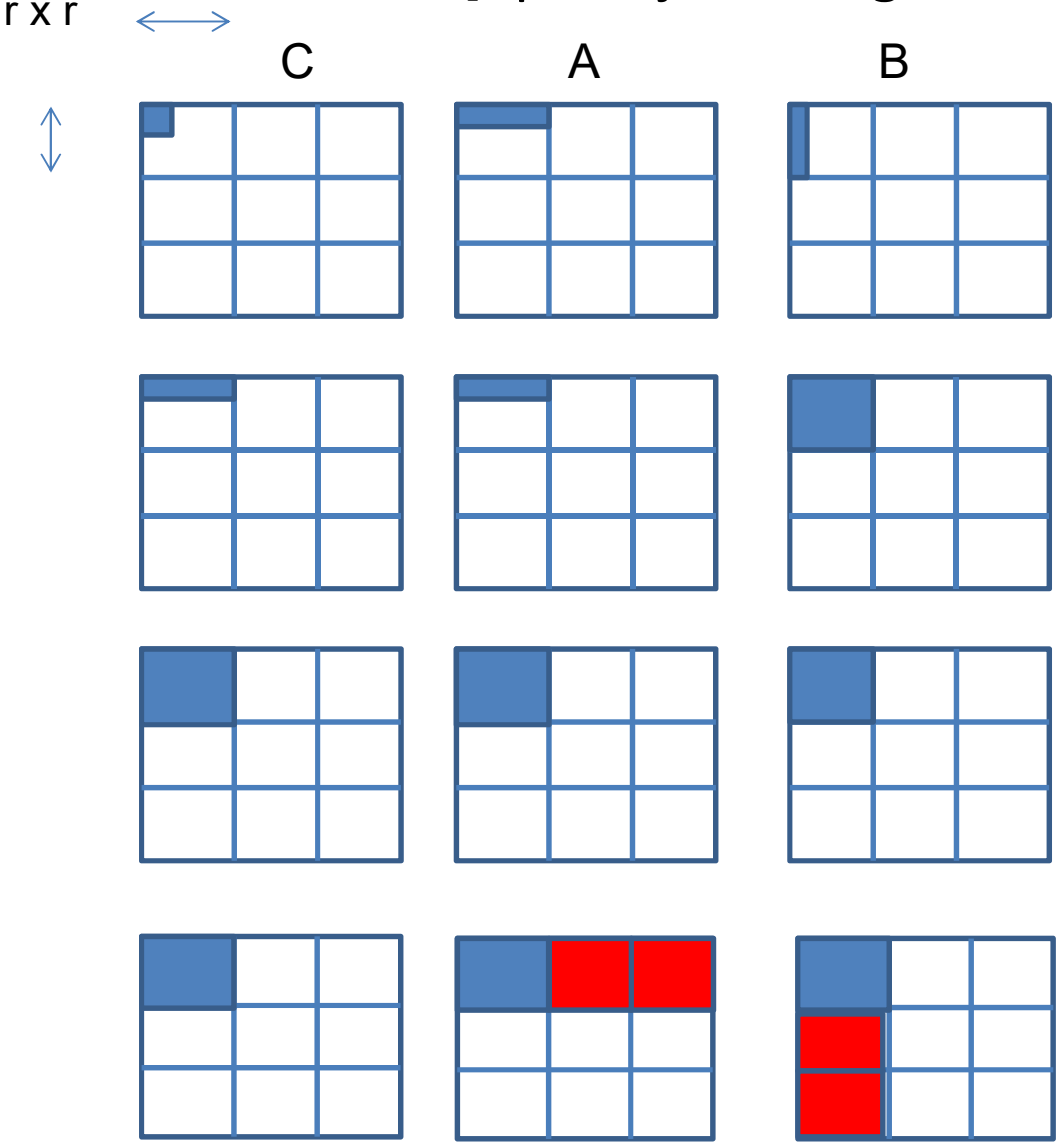
Wydajność rozwiązania - mnożenie macierzy wer2

Różnice względem wersji 1:

- Można wykorzystać potencjalnie wszystkie SM – wzrost maksymalnej prędkości obliczeń 27 x
- Można uzyskać większą maksymalną zajętość SM dla uruchomienia – np. 4 bloki 256 wątków dają maksimum 1024 watki na SM – maksymalna, bo zależna od wielkości instancji.
- Co najmniej dwukrotny (1024 zamiast 512) wzrost liczby wątków nie zapewni ciągłości obliczeń dla CGMA=1 – nadal dłużej trwa dostęp (pobieranie) do danych niż ich obliczenia.
- Nasz cel - zwiększyć CGMA – liczba operacji arytmetycznych stała (złożoność), trzeba zmniejszyć liczbę dostępów do pamięci globalnej, celem jest zapewnienie pracy rdzeni bez przestojów.

Z innego wykładu: Mnożenie macierzy – pamięć podręczna [operacje na fragmentach tablic]

r x r




```
for (int kk = k ; kk < k+r ; kk++)
      C[ii][jj] + = A[ii][kk] * B[kk][jj];
```

```
for (int jj = j ; jj < j+r ; jj++)
      for (int kk = k ; kk < k+r ; kk++)
            C[ii][jj] + = A[ii][kk] * B[kk][jj];
```

```
for ( int ii = i ; ii < i+r; ii++)
    for (int jj = j ; jj < j+r ; jj++)
      for (int kk = k ; kk < k+r ; kk++)
            C[ii][jj] + = A[ii][kk] * B[kk][jj];
```

```
for (int k = 0 ; k < n ; k+=r)
    for ( int ii = i ; ii < i+r; ii++)
      for (int jj = j ; jj < j+r ; jj++)
        for (int kk = k ; kk < k+r ; kk++)
            C[ii][jj] + = A[ii][kk] * B[kk][jj];
```

MM wer.3 - przebieg

- Zastosujemy to samo podejście jak w mnożeniu macierzy algorytmem zagnieżdżonych 6 pętli i optymalizacją wykorzystania pamięci podręcznej. Zamiast lokalności czasowej dostępu do pamięci podręcznej wykorzystujemy czasowo lokalnie dane w pamięć współdzielonej.
- Pobrania danych do pamięci współdzielonej realizują wątki kolektywnie przed obliczeniami . Konieczna synchronizacja zakończenia pobrania i zakończenia obliczeń, gdyż wątki nie są wykonywane równocześnie.
- ETAPY pracy:
  - Pobranie danych do pamięci współdzielonej bloku wątków – możliwość łączenia dostępów (przepisywanie bloków danych realizować można w wygodnej dla efektywności kolejności)
  - Każdy wątek wylicza wynik częściowy na podstawie danych w dostępnej mu pamięci współdzielonej.
  - Synchronizacja każdego bloku wątków przed wymianą danych w pamięci współdzielonej.
  - Kolejne kroki kolektywnego pobrania danych przez blok wątków do pamięci współdzielonej, a następnie uzupełnianie sum częściowych o wyniki mnożeń elementów z kolejno pobranych bloków.
  - Kolektywny zapis wyników (każdy wątek jeden wynik) do pamięci.

MM wer.3 - efektywność

- Zysk – mniej pobrań z pamięci globalnej, wielokrotne wykorzystanie danych dostępnych w pamięci podręcznej(rozmiar bloku)
- Strata - obliczenia konkretnego bloku wątków nie mogą być realizowane jednocześnie z pobieraniem danych dla tego bloku (synchronizacja miedzy etapami przetwarzania).
- Im większy blok tym większa krotność wykorzystania raz pobranych do pamięci współdzielonej danych - CGMA.
- Im większy blok tym niższy stopień zrównoleglenia pobrań danych i obliczeń w ramach jednego SM. LECZ: Różne bloki mogą realizować różne etapy obliczeń.

Mnożenie macierzy wersja 3 cz.1

```
global__ void MatrixMulKernel_3(float* Ad, float* Bd, float* Cd, int Width)
{
__shared__float Ads[SUB_WIDTH][SUB_WIDTH];
__shared__float Bds[SUB_WIDTH][SUB _WIDTH];
// określenie obliczanego przez wątek elementu macierzy (jak w poprzednim kodzie)
for (int m = 0; m < WIDTH/ SUB_WIDTH; ++m) {
        Ads[tx][ty] = Ad [Row][m*SUB_WIDTH + tx];  //kolejny element dla sąsiedniego wątku
        Bds[tx][ty] = Bd[m*SUB_WIDTH + ty][Col]; // używana kolumna – jakość pobrań ?
         __syncthreads();
         for (int k = 0; k < SUB_WIDTH; ++k)
                  C_local += Ads[tx][k] * Bds[k][ty];
         __syncthreads();
        }
Cd[Row][Col] = C_local;
}
```

Mnożenie macierzy przez bloki wątków w przy użyciu pamięci współdzielonej

Mnożenie macierzy wersja 3 – wyjaśnienia

// wątki wspólnie bloku ładują podmacierze do pamięci współdzielonej

// WĄTKI O SĄSIEDNICH ID ładują sąsiednie elementy z pamięci

//oczekiwanie na dane podmacierzy w pamięci współdzielonej - synchronizacja

//oczekiwanie na koniec obliczeń przed pobraniem nowych danych - synchronizacja

Efektywność MM wersji 3

- Im większy blok ładowany do pamięć współdzielonej (SHM) tym mniej razy dane pobierane i mniejsze wymagania na przepustowość pamięci globalnej.
- Dla tego kodu rozmiar bloku danych każdej macierzy wejściowej jest równy rozmiarowi bloku wątków.
- Rozmiar bloku wątków wpływa na wielkość potrzebnej pamięci SHM i możliwości uruchamiania bloków w SM.
- W wewnętrznej pętli każdy wątek bloku o rozmiarach BLOCK_SIZE x BLOCK_SIZE wykonuje BLOCK_SIZE operacji MUL_ADD i dwa pobrania danych macierzy wejściowej, CGMA = BLOCK_SIZE
- Ograniczeniem na prędkość tych obliczeń są:
  - synchronizacja pracy wątków – w bloku wątków faza pobrań danych i faza obliczeń dla bloku wątków nie są równoległe,
  - czas dostępu do pamięci globalnej.

Mnożenie macierzy wersja 4

Wersja 3

Pętla po blokach macierzy

{

Ładowanie kolejnego bloku danych do pamięci współdzielonej A;

Synchronizacja;

Obliczenia na pw A;

Synchronizacja;

}

Wersja 4

Pobranie pierwszego bloku danych z pamięci globalnej do A (do rejestru lub pamięci współdzielonej)

Pętla po blokach

{

Przepisanie danych z A do pamięci współdzielonej B; -- zwolnienie A

Synchronizacja;

Pobranie kolejnego bloku danych z pamięci globalnej do A;

Obliczenia na pamięci pw B;

Synchronizacja; -- zwolnienie B

}

Wzrost ilości operacji i większe wymagania zasobowe, lecz inna synchronizacja umożliwiająca nakładanie obliczeń i pobrania danych dla wątków jednego bloku.

Dalsze optymalizacje

- #pragma unroll - rozwinięcie pętli zmniejsza liczbę operacji (obsługujące pętle).
- Wzrost ilości pracy realizowanej przez wątek – obliczanie większej liczby wyników (w oparciu o częściowo te same dane).
- Obliczenia dla wielu tablic –przesłanie danych do obliczeń na kartę równoczesne z obliczeniami dla poprzednio odebranych danych.
- Ocena efektywności wprowadzonych rozwiązań:
  - CUDA_Occupancy_Calculator.xls
  - Nvidia Visual profiler

Sterowanie realizacją instrukcji przez wątki warpu w SM

- Problemem efektywnościowym związanym z rozgałęzieniami w kodzie jest rozbieżność przetwarzania.
  - Załóżmy, że różne wątki jednego warpu mają realizować inne ciągi instrukcji.
  - Realizacja: różne ścieżki przetwarzania są pokonywane dla każdej grupy wątków sekwencyjnie – spadek poziomu równoległości przetwarzania – spadek efektywności.
- Rozbieżność przetwarzania może się pojawić jako efekt wystąpienia w kodzie warunku uzależnionego od identyfikatora wątku np. :
- if (threadIdx.x >16)

  może tworzyć dwie ścieżki realizacji kodu dla wątków jednego warpu.
- if (threadIdx.x / rozmiar_warpu >1)

  może powodować rozbieżność przetwarzania w ramach bloku lecz nie powoduje rozbieżności przetwarzania w ramach warpu (inne przetwarzanie (0-31), inne 31-N (rozbieżność ta nie będzie problemem)

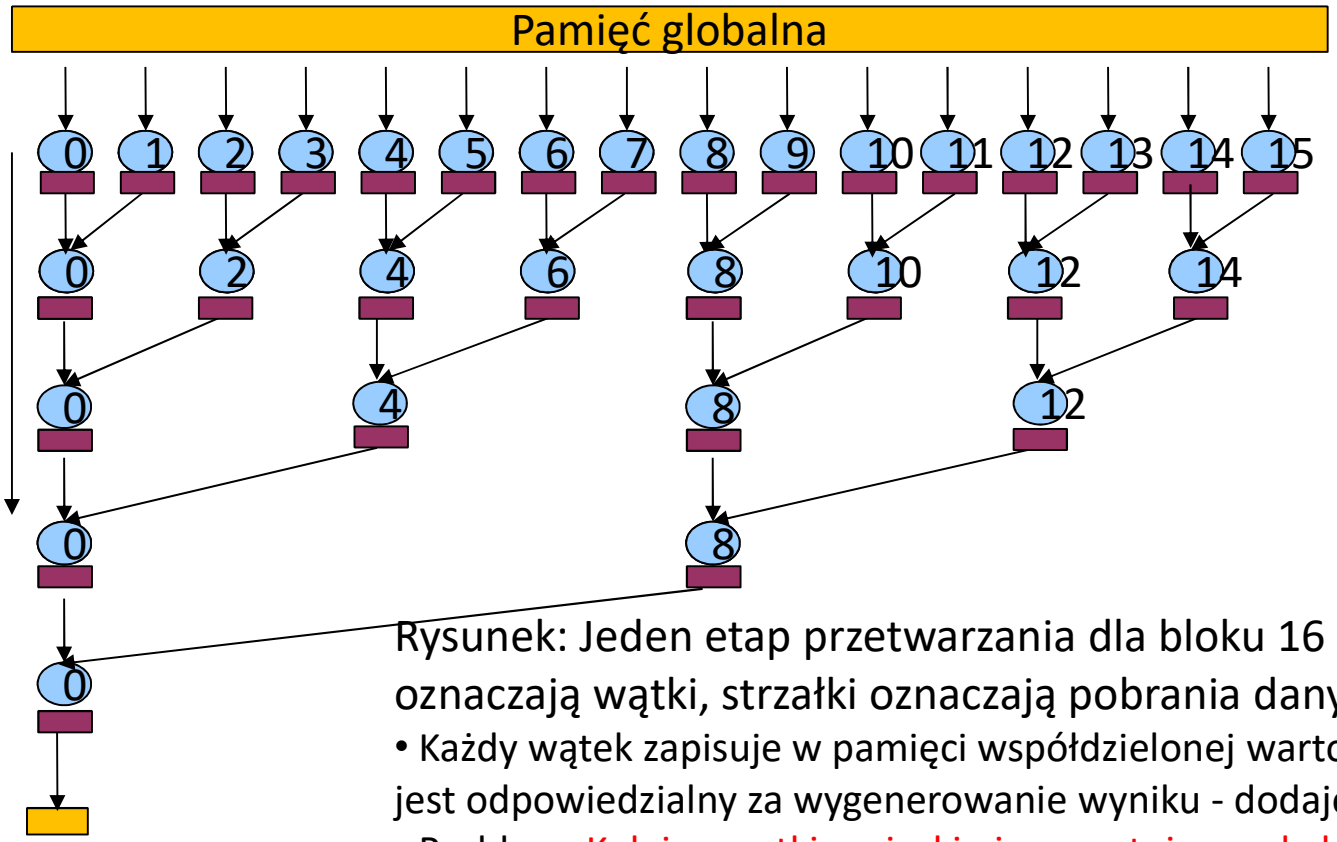
Równoległa redukcja - problem

- Problem polega na scaleniu elementów wektora do jednej wartości. Przykładem redukcji równoległej może być wyznaczenie **sumy elementów** wektora danych dostępnego w pamięci współdzielonej systemu równoległego.
- Przetwarzanie odbywa się na procesorach kart graficznych.
- Dane przechowywane są w pamięci globalnej.
- Wyniki pośrednie przechowywane są w pamięci współdzielonej bloku wątków lub w pamięci globalnej.
- Wynik obliczeń jest przekazywany do pamięci komputera host na potrzeby testów poprawności obliczeń.

Równoległa redukcja – etapy przetwarzania

1. Uruchomienie obliczeń w oparciu o niezbędną liczbę wątków wynikającą:
   - z liczby sumowanych elementów i
   - liczby sumowań przypadających na wątek.
2. Każdy blok wątków synchronizuje się w przetwarzaniu i generuje jeden wynik – sumę częściową.
3. Uruchomienie obliczeń w oparciu o niezbędną liczbę wątków wynikającą z liczby bloków poprzednio (w poprzednim etapie sumowania) uruchomionego gridu i liczby sumowań przypadających na wątek.
4. Powtarzanie procedury od kroku 2 do uzyskania jednego wyniku globalnego.
5. Ze względu na współdzielenie danych między wątkami w bloku (w kolejnych etapach sumowania) konieczna synchronizacja pracy wątków bloku po każdym etapie sumowania.

Równoległa redukcja - Struktura przetwarzania 1

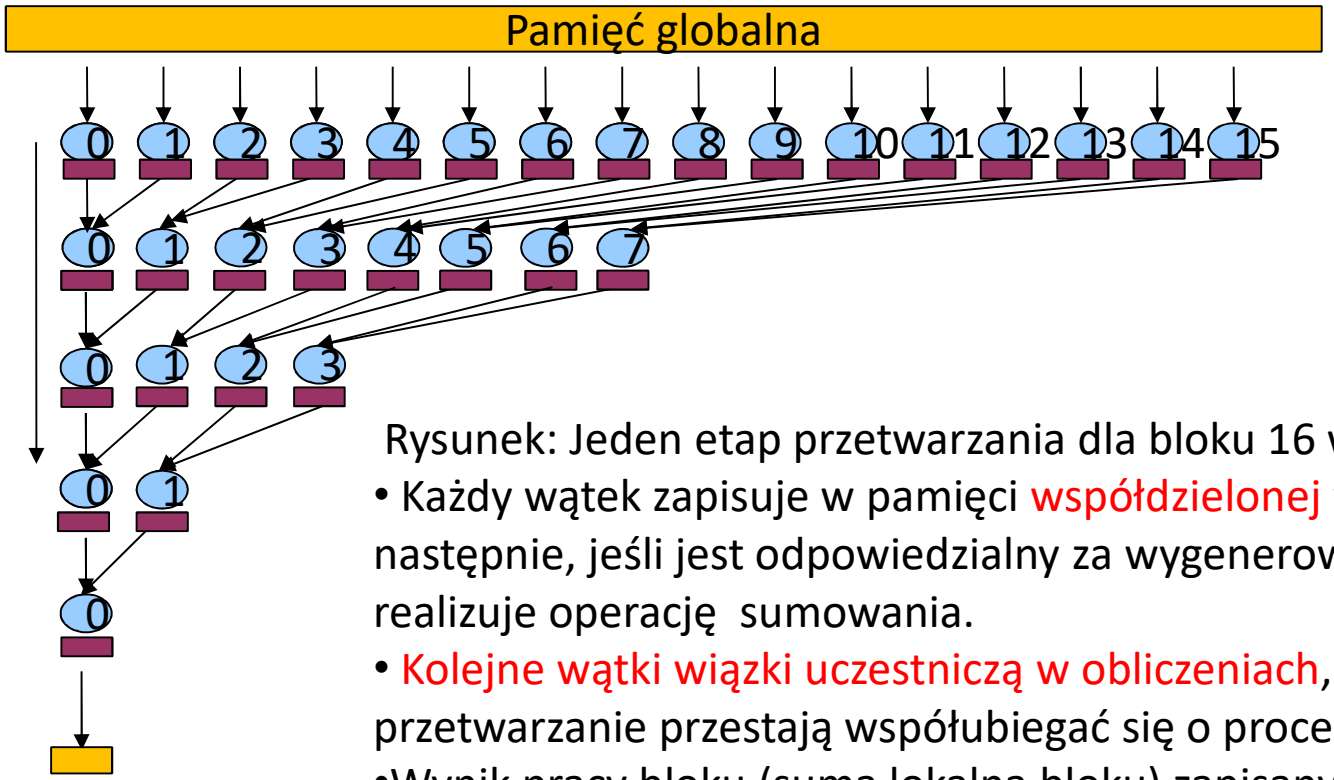


Zapis do pamięci globalnej

Rysunek: Jeden etap przetwarzania dla bloku 16 wątków, okręgi oznaczają wątki, strzałki oznaczają pobrania danych z pamięci.

- Każdy wątek zapisuje w pamięci współdzielonej wartość, a następnie, jeśli jest odpowiedzialny za wygenerowanie wyniku - dodaje
- Problem: Kolejne wątki z wiązki nie uczestniczą w kolejnych etapach obliczeń – wiązka zostaje uszeregowana do procesora, lecz coraz mniej wątków wykonuje II obliczenia – utrata efektywności przetwarzania.
- Wynik - pracy bloku (suma lokalna bloku) zapisany zostaje w pamięci globalnej dla wykorzystania w kolejnym etapie sumowania przez nowy grid.

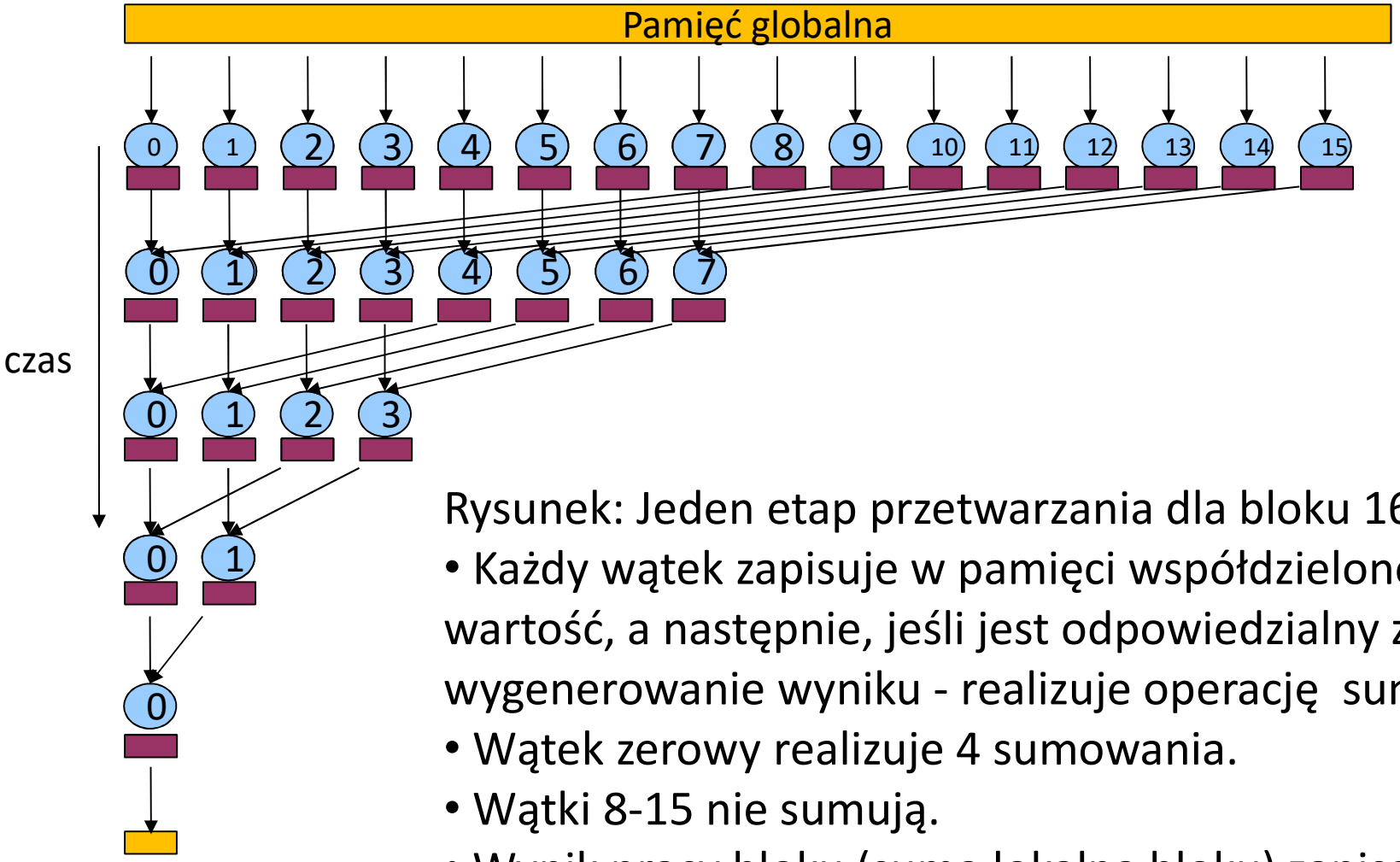
Równoległa redukcja - Struktura przetwarzania 2



Rysunek: Jeden etap przetwarzania dla bloku 16 wątków.

- Każdy wątek zapisuje w pamięci współdzielonej wartość, a następnie, jeśli jest odpowiedzialny za wygenerowanie wyniku realizuje operację sumowania.
- Kolejne wątki wiązki uczestniczą w obliczeniach, wiązki kończące przetwarzanie przestają współubiegać się o procesor.
- Wynik pracy bloku (suma lokalna bloku) zapisany zostaje w pamięci globalnej dla wykorzystania w kolejnym etapie sumowania przez nowy grid.
- Problem: równoczesne dostępy do oddalonych lokacji pamięci współdzielonej – możliwy konflikt dostępu do tych samych banków pamięci współdzielonej. (Pamięć globalna ?)

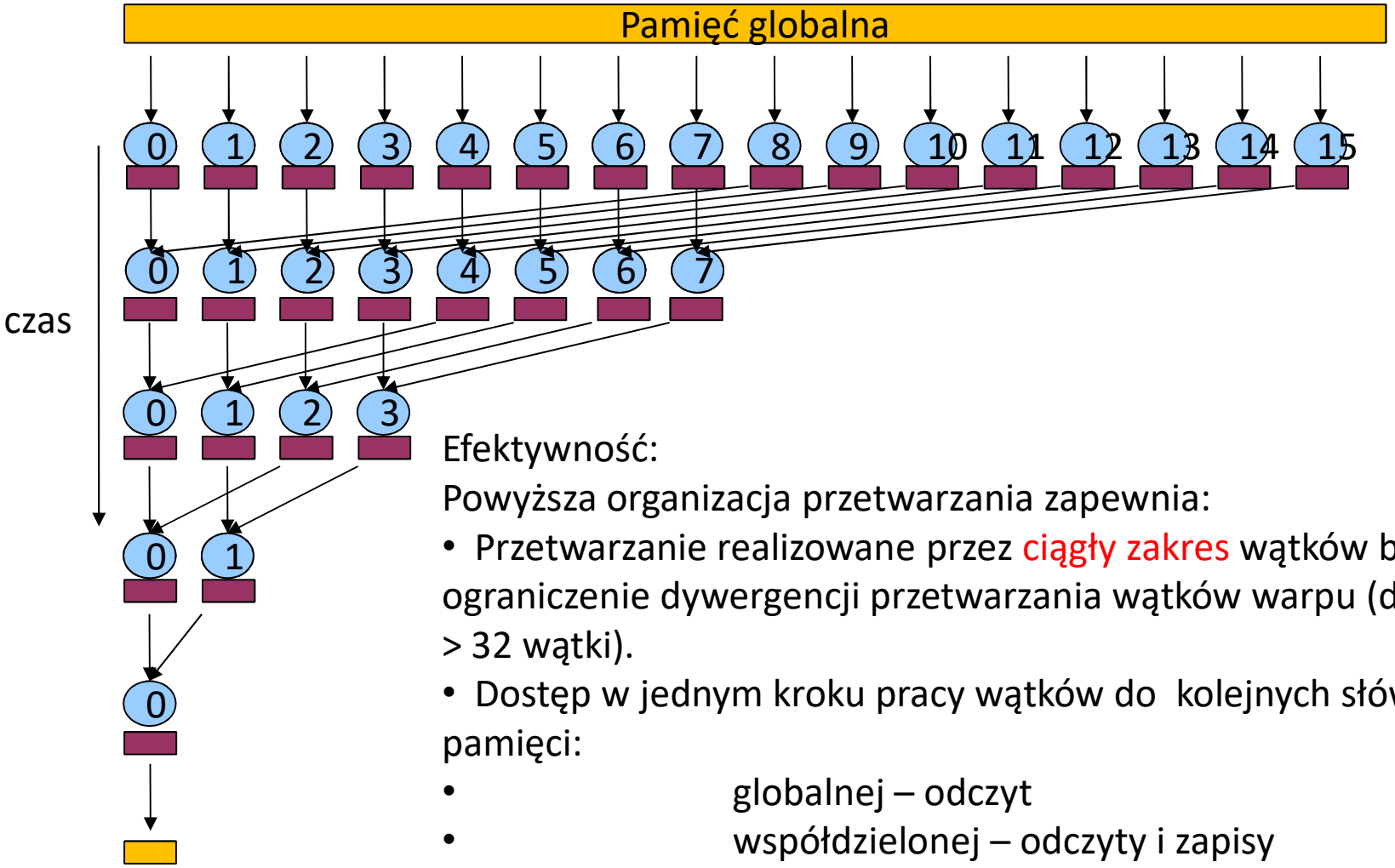
Równoległa redukcja - Struktura przetwarzania 3



Rysunek: Jeden etap przetwarzania dla bloku 16 wątków.

- Każdy wątek zapisuje w pamięci współdzielonej wartość, a następnie, jeśli jest odpowiedzialny za wygenerowanie wyniku - realizuje operację sumowania.
- Wątek zerowy realizuje 4 sumowania.
- Wątki 8-15 nie sumują.
- Wynik pracy bloku (suma lokalna bloku) zapisany …

Równoległa redukcja - struktura przetwarzania 3



Efektywność:

Powyższa organizacja przetwarzania zapewnia:

- Przetwarzanie realizowane przez ciągły zakres wątków bloku – ograniczenie dywergencji przetwarzania wątków warpu (dla bloków > 32 wątki).
- Dostęp w jednym kroku pracy wątków do kolejnych słów z pamięci:
  - globalnej – odczyt
  - współdzielonej – odczyty i zapisy
- Pozwala na efektywny dostęp poprzez różne banki pamięci współdzielonej.

Warianty struktury przetwarzania

- Korzystanie z pamięci globalnej lub pamięci współdzielonej wątków do zapisu wyników pośrednich.
- Realizacja sumowań pierwszego etapu przez wszystkie wątki bloku.
- Realizacja wielu sumowań przez wątki bloku – łączone pobieranie z pamięci globalnej i sumowanie z wartością w pamięci współdzielonej.

Przykładowy kod kernela wg struktury pierwszej przetwarzania

```
//sumowanie (w pamięci globalnej) przez blok wątków 2* blockDim.x  elementów
__global__ void block_sum(float *dane,  float *wyniki, const size_t rozmiar_danych)
{
  unsigned int i = 2*(blockIdx.x * blockDim.x + threadIdx.x);
                //identyfikator elementu zależny od  identyfikatora wątku i bloku (suma 2 elementów)
 for(unsigned int odstep =1; odstep< 2*blockDim.x; odstep *=2)
                //odstęp dla sumowanych elementów
 {
                 if (threadIdx.x %odstep ==0)
                                        //wykluczenie wątków w kolejnych etapach co 2,4,8,…
                                if (i+odstep< rozmiar_danych)                 //test rozmiaru danych
                                        dane[i] += dane[i+odstep];
                  __syncthreads();                                      //synchronizacja gotowości danych
   }
  if (threadIdx.x == 0)                                                  // wątek 0 zapisuje wynik
                {
                                wyniki[blockIdx.x] = dane[2*blockIdx.x * blockDim.x];
                }
}
```

Efektywność przetwarzania GTX 240
16MB – 4B *4M elementy

Instancja: 16MB – 4B *4M elementów – konieczne 3 uruchomienia kernela (512 wątków)

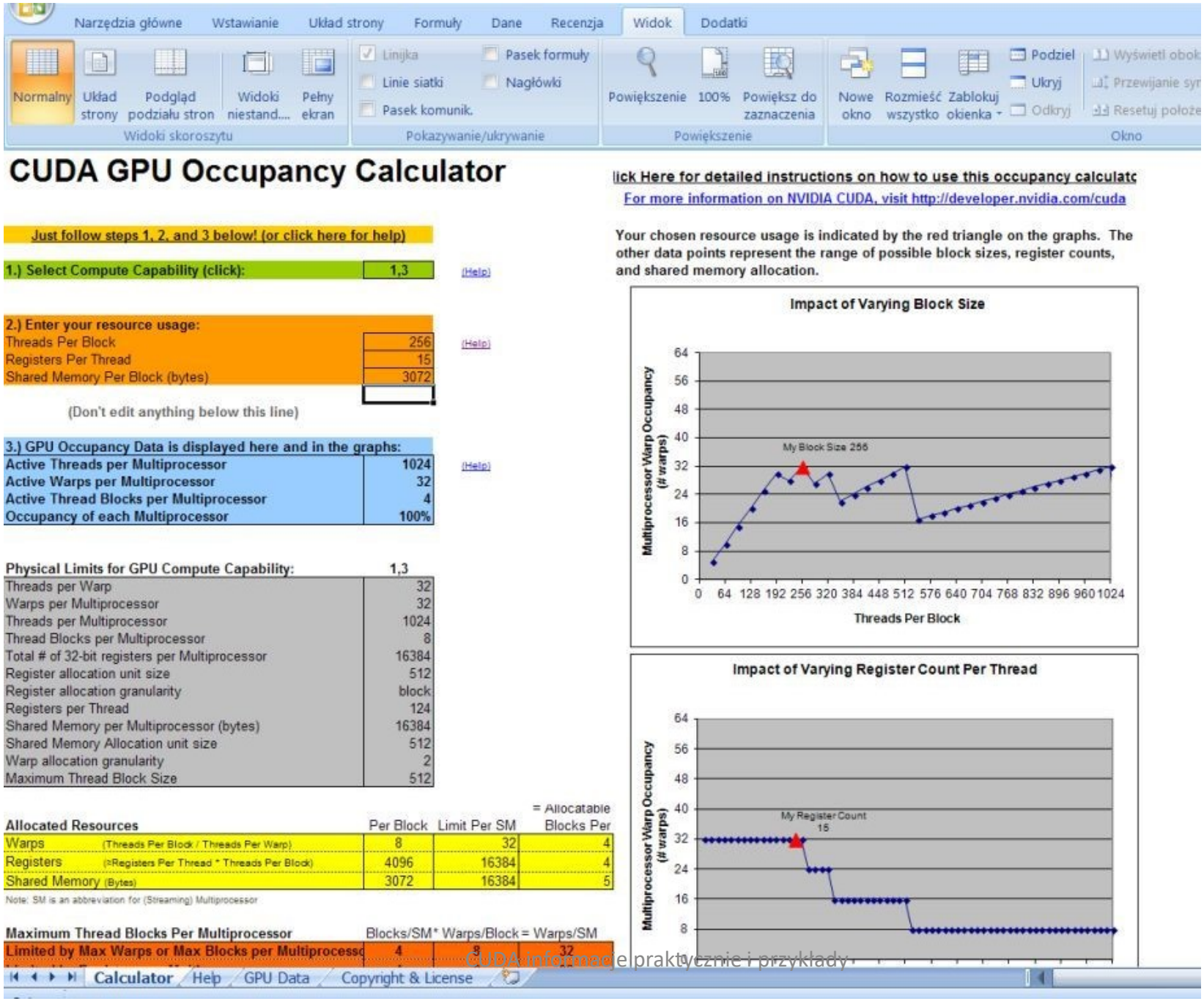
I uruchomienie kernela: Grid: 4K bloków, Blok: 512 wątków (2M)

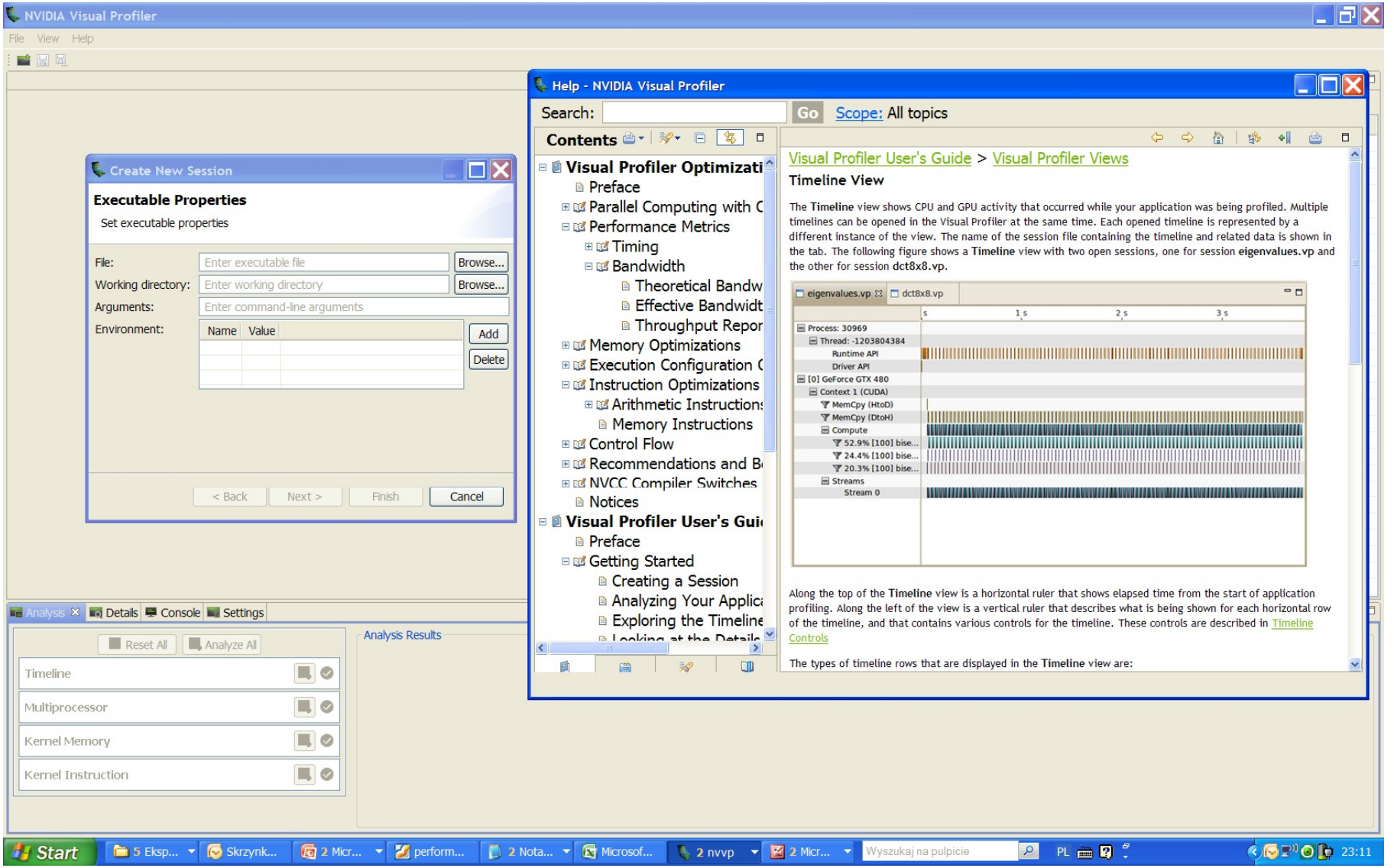
II uruchomienie kernela: Grid: 4 blok, Blok: 512 wątków (2K)

III uruchomienie kernela: 2 wątki (4)

Metoda 3S pozwala na 3,4 krotne przyspieszenie obliczeń

Wersja Struktury	Użyta pamięć	Czas obliczeń ms	Przepustowość Obliczeń GB/s	Wzrost efektywności względem 1G o
1	G	13,6	1,17	
1	S	12,6	1,27	9%
2	G	9,2	1,74	49%
2	S	8,6	1,86	59%
3	G	?	?	
3	S	3,1	5,16	340%


Narzędzia główne
Wstawianie
Układ strony
Formuły
Dane
Recenzja
Widok
Dodatki
Normalny
Układ strony
Podgląd podziału stron
Widoki niestand....
Pełny ekran
Widoki skoroszytu
Linijka
Linie siatki
Pasek komunik.
Pasek formuły
Nagłówki
Pokazywanie/ukrywanie
Powiększenie
100%
Powiększ do zaznaczenia
Powiększenie
Nowe okno
Rozmieść wszystko
Zablokuj okienka
Podziel
Ukryj
Odkryj
Okno
CUDA GPU Occupancy Calculator
lick Here for detailed instructions on how to use this occupancy calculatc
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda
Just follow steps 1, 2, and 3 below! (or click here for help)
1.) Select Compute Capability (click): 1,3 (Help)
2.) Enter your resource usage:
Threads Per Block 256 (Help)
Registers Per Thread 15
Shared Memory Per Block (bytes) 3072
(Don't edit anything below this line)
3.) GPU Occupancy Data is displayed here and in the graphs:
Active Threads per Multiprocessor 1024 (Help)
Active Warps per Multiprocessor 32
Active Thread Blocks per Multiprocessor 4
Occupancy of each Multiprocessor 100%
Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.
Impact of Varying Block Size
Multiprocessor Warp Occupancy (# warps)
My Block Size 256
Threads Per Block
Impact of Varying Register Count Per Thread
My Register Count 15
Physical Limits for GPU Compute Capability: 1,3
Threads per Warp 32
Warps per Multiprocessor 32
Threads per Multiprocessor 1024
Thread Blocks per Multiprocessor 8
Total # of 32-bit registers per Multiprocessor 16384
Register allocation unit size 512
Register allocation granularity block
Registers per Thread 124
Shared Memory per Multiprocessor (bytes) 16384
Shared Memory Allocation unit size 512
Warp allocation granularity 2
Maximum Thread Block Size 512
Allocated Resources Per Block Limit Per SM = Allocatable Blocks Per
Warps (Threads Per Block / Threads Per Warp) 8 32 4
Registers (=Registers Per Thread * Threads Per Block) 4096 16384 4
Shared Memory (Bytes) 3072 16384 5
Note: SM is an abbreviation for (Streaming) Multiprocessor
Maximum Thread Blocks Per Multiprocessor Blocks/SM * Warps/Block = Warps/SM
Limited by Max Warps or Max Blocks per Multiprocesso 4 8 32
Calculator
Help
GPU Data
Copyright & License


NVIDIA Visual Profiler
File View Help
Create New Session
Executable Properties
Set executable properties
File:
Enter executable file
Browse...
Working directory:
Enter working directory
Browse...
Arguments:
Enter command-line arguments
Environment:
Name Value
Add
Delete
< Back
Next >
Finish
Cancel
Help - NVIDIA Visual Profiler
Search:
Go
Scope: All topics
Contents
Visual Profiler Optimizati
Preface
Parallel Computing with C
Performance Metrics
Timing
Bandwidth
Theoretical Bandw
Effective Bandwidt
Throughput Repor
Memory Optimizations
Execution Configuration C
Instruction Optimizations
Arithmetic Instructions
Memory Instructions
Control Flow
Recommendations and B
NVCC Compiler Switches
Notices
Visual Profiler User's Gui
Preface
Getting Started
Creating a Session
Analyzing Your Applica
Exploring the Timeline
Visual Profiler User's Guide > Visual Profiler Views
Timeline View
The Timeline view shows CPU and GPU activity that occurred while your application was being profiled. Multiple timelines can be opened in the Visual Profiler at the same time. Each opened timeline is represented by a different instance of the view. The name of the session file containing the timeline and related data is shown in the tab. The following figure shows a Timeline view with two open sessions, one for session eigenvalues.vp and the other for session dct8x8.vp.
eigenvalues.vp
dct8x8.vp
Process: 30969
Thread: -1203804384
Runtime API
Driver API
[0] GeForce GTX 480
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
52.9% [100] bise...
24.4% [100] bise...
20.3% [100] bise...
Streams
Stream 0
Along the top of the Timeline view is a horizontal ruler that shows elapsed time from the start of application profiling. Along the left of the view is a vertical ruler that describes what is being shown for each horizontal row of the timeline, and that contains various controls for the timeline. These controls are described in Timeline Controls
The types of timeline rows that are displayed in the Timeline view are:
Analysis
Details
Console
Settings
Reset All
Analyze All
Timeline
Multiprocessor
Kernel Memory
Kernel Instruction
Analysis Results
Start
5 Eksp...
Skrzynk...
2 Micr...
perform...
2 Nota...
Microsof...
2 nvvp
2 Micr...
Wyszukaj na pulpicie
PL
23:11