Przetwarzanie równoległe

Laboratorium

Programowanie CUDA na NVIDIA GPU

Michał Kempka 105256
Jarasz Łojka 114816
(wtorek 13:30, nieparzyste)

Wersja: 1
Data przesłania: 02.02.2015 r.

1. Badana zagadnienie

Projekt polega na zbadaniu efektywności algorytmów sumowania (redukcji) elementów wektora przy użyciu platformy programistycznej CUDA. Do badania zostały użyte trzy podstawowe wersje algorytmu opisane w wymaganiach.

2. Opis użytej karty graficznej

- model: Nvidia GeForce GT 635M
- CUDA Driver Version / Runtime Version: 6.5/6.5
- CUDA Compute Capability: 2.1
- Liczba multiprocesorów: 2
- Liczba rdzeni CUDA: 96
- Częstotliwość taktowania zegara GPU: 950 MHz
- Częstotliwość taktowania zegara pamięci: 900MHz
- Szerokość szyny: 128 bitów
- Rozmiar całkowitej pamięci GPU: 2GB
- Rozmiar pamięci L2: 131072B (128KB)
- Rozmiar pamięci współdzielonej na blok: 49152B (48 KB)
- Maksymalna liczba wątków na multiprocesor: 1536
- Maksymalna liczba wątków na blok: 1024
- Liczba rejestrów dostępnych na blok: 32768

3. Opis badanych algorytmów

a) Wersja 1

Rozbieżność wątków i brak łączonych dostępów do pamieci (high branch divergence + non-coalesced memory access).

Kod:

```
template <typename T>
__global__ void reduce1(T *g_idata, T *g_odata)
{
	extern __shared__  T sdata[];

	unsigned int tid = threadIdx.x;
	unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
	sdata[tid] = g_idata[i];
	__syncthreads();

	for (unsigned int s = 1; s < blockDim.x; s *= 2)
	{
		if (tid % (2 * s) == 0) {
			sdata[tid] += sdata[tid + s];
		}
		__syncthreads();
	}
	if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Opis:
W prezentowanym kodzie każdy wątek iteruje po elementach tablicy większych (x2,x4,x8 …) od przypisanemu im elementowi i sumuje je. Każda iteracja eliminuje z pracy połowę wątków, że rozbieżność przetwarzania wątków jest duża i większość z nich nie bierze czynnego udziału w przetwarzaniu. Co więcej sąsiednie wątki (bliskie indeksy – prawdopodobnie ten sam warp/ half-warp) nie odwołują się do sąsiednich komórek pamięci co utrudnia łączony dostęp do pamięci.

a) Wersja 2

Mała rozbieżność wątków i brak łączonych dostępów do pamieci (high branch divergence + non-coalesced memory access).

Kod:

```
template <typename T>
__global__ void reduce2(T *g_idata, T *g_odata)
{
	extern __shared__  T sdata[];

	unsigned int tid = threadIdx.x;
	unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
	sdata[tid] = g_idata[i];
	__syncthreads();

	for (unsigned int s = 1; s < blockDim.x; s *= 2)
	{
		unsigned int index = 2 * s * tid;
		if (index < blockDim.x) {
			sdata[index] += sdata[index + s];
		}
		__syncthreads();
	}


	if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Opis:
Prezentowany kod poprawia sytuację rozbieżności z wersji 1 gdyż wątki zmieniają element, do którego zapisują co iterację przez co sąsiadujące wątki wykonują to samo zadanie. W każdej iteracji połowa wątków przestaje być potrzebna, lecz wszystkie te wątki są zgrupowane (nie przeplatane z wątkami pracującymi jak w wersji 1). Problem braku dostępów łączony pozostaje gdyż wątki odwołują się do oddalonych od siebie komórek.

a) Wersja 3

Niska rozbieżność przetwarzania i efektywne dostępny do pamięci.

Kod:

```
template <typename T>
__global__ void reduce3(T *g_idata, T *g_odata)
{
      extern __shared__  T sdata[];

      unsigned int tid = threadIdx.x;
      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
      sdata[tid] = g_idata[i];
      __syncthreads();

      for (unsigned int s = blockDim.x / 2; s>0; s >>= 1)
      {
            if (tid < s) {
                  sdata[tid] += sdata[tid + s];
            }
            __syncthreads();
      }
      if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Opis:
Wersja 3 od wersji 2 różni się tym, że wątki przechowują wyniki w sąsiednich komórkach zaś sumują nie sąsiednie komórki, lecz odpowiedniki z drugiej połowy aktualnie sumowanej tablicy. W efekcie sąsiednie wątki odczytują sąsiednie komórki pamięci co umożliwia łączone odczyty z pamięci.

4. Wyniki pomiarów

Wyniki uwzlgędniają charakterystyki czasowe sumowania wektora typu double o różnych rozmiarach i dla rozmiaru bloku 1024. Nie udało nam się zauważyć żadnych rozsądnych zależności w pozostałych metrykach dlatego nie zostały tu umieszczone.

Wersja	Czas [ms]
1	11.22
2	10.28
3	6.92

Tabela 1: Czas trwania dla wielkości instancji 2^{20}

Wersja	Czas [ms]
1	89.79
2	82.25
3	55.38

Tabela 2: Czas trwania dla wielkości instancji 2^{23}

Wersja	Czas [ms]
1	325.63
2	300.63
3	203.21

Tabela 3: Czas trwania dla wielkości instancji 2^{20}

5. Wnioski i spostrzeżenia

Jak widać w tabelach 1-3 wersja 1 działa szybciej niż wersja 2 zaś wersja 3 działą szybciej niż wersja 2. Jest to spowodowane oczywiście faktem, ze wersja 1 ma wysoką rozbieżność i brak dostępów łączonych, wersja 2 brak dostępów łączonych, a wersja 3 rozwiązuje te dwa problemy.

Wniosek praktyczny płynący z tych spostrzeżeń jest taki by w czasie projektowania algorytmu zadbać o to aby sąsiednie wątki odwoływały się do sąsiednich komórek pamięci oraz aby sąsiednie wątki wykonywały te same czynności (te same instrukcje) gdyż badana architektura wspiera łączone dostępy do pamięci (do komórek w innych bankach czyli o sąsiednich adresach) i wykonywanie tych samych instrukcji na różnych danych przez procesory w tym samej grupie (tutaj warp).

Należy pamiętać także by dobierać rozmiary bloków tak by były wielokrotnościami 32 (rozmiar warpu). W przeciwnym razie część wątków nie bedzie brała czynnie udziału w przetwarzaniu.