Raport zadanie Brandes Betweenness-Centrality

Karol Wołonciej 371869

1. Zastosowany algorytm:

Dostępne są 2 główne podejścia jedno vertex-based i drugie edge-based które mogą być preferowane w zależności od grafu jaki analizujemy i możliwe jest użycie kombinacji wykonując na początku sprawdzenie fragmentu grafu. Graf vertex-based może wykorzystywać kolejki by analizować tylko potrzebne wierzchołki (odpowiedniego poziomu).

Zastosowany algorytm to vertex-based bez kolejek (zawsze sprawdzamy wszystkie wierzchołki) gdzie jeden wątek liczy jeden wierzchołek i jego sąsiadów. Dodane modyfikacje z papera czyli wirtualne wierzchołki i Stride-CSR representation grafu. Pominięta została modyfikacja wprowadzająca tablice reach aby nie obliczać wierzchołków stopnia jeden (ponieważ na Gowalli był tylko 1 taki wierzchołek i nie spodziewam się by była ich istotna liczba w grafach testowych).

Zdecydowałem się finalnie zmienić algorytm na vertex-based bez kolejek ponieważ nie wiedziałem jak można by zastosować modyfikacje Stride-CSR representation grafu i virtual vertices.

2. Współbieżność:

Liczenie faz forward i backward zrównoleglający liczenie najkrótszych ścieżek nie okłada mocno karty dlatego liczone jest naraz wiele wierzchołków (każdy liczone przez jeden strumień). Każdy strumień ma osobne tablice takie jak d, delta, sigma, partial_bc. Synchronizacja wszystkich strumieni odbywa się po każdej iteracji pętli forward phase oraz backward phase i przed dodaniem częściowych wyników bc każdego strumienia do finalnej tablicy bc.

3. Szczegóły implementacji:

-Tablice dwuwymiarowe alokowane są jako tablica jednowymiarowa

-Strumienie nie są synchronizowane pojedynczo tylko razem cudaDeviceSynchronize()

-Kilka dodatkowych strumieni w celu asynchronicznej memset przed kolejną rundą faz strumieni

-Grid jest jednowymiarowy składający się wyłącznie z jednowymiarowych bloków

4. Modyfikowalne parametry:

Parametry które mają istotny wpływ na współbieżność to:

```
#define BLOCK_SIZE_X                          256
#define NUMBER_OF_STREAMS                     800
#define NUMBER_OF_NEIGBOURS_IN_STRIDE_GRAPH   8
```

`BLOCK_SIZE_X` - rozmiar blockDim.x jednowymiarowych bloków (eksperymentalnie jeden z szybszych).

`NUMBER_OF_STREAMS` - liczba strumieni, każdy liczący jeden wierzchołek w jednej iteracji. Wspólnie są w stanie niemal wypełnić kartę. Zwiększając tę wartość z 1 otrzymujemy duże przyśpieszenie na początku które jednak szybko spada. Dzięki temu karta jest cały czas możliwie zapełniona. Każdy strumień potrzebuje użyć O(N) pamięci więc ich maksymalna liczba będzie różna w zależności od grafu. Szacuje że optymalnie jest dać ok 50 (i tyle jest ustawione) (dla Gowalli program nie wywalał się do 1200 strumieni). Prawdopodobnie więcej strumieni wiąże się z overhead który ponownie zaczyna spowalniać obliczenia.

`NUMBER_OF_NEIGHBOURS_IN_STRIDE_GRAPH` - liczba sąsiadów wirtualnych wierzchołków.

Makra zdefiniowane są w settings.h

5. Optymalizacje: (z omówieniem i fragmentami asemblera)

Teraz przedstawię wszelkie optymalizacje które udało mi się wcisnąć (a które zazwyczaj zupełnie nic nie dały, programming guide kłamie).

-Używam cudaHostAlloc ponieważ chcemy oczywiście liczyć rzeczy asynchronicznie.

-Dla 4 bajtowych typów używam cudaMallocPitch w celu zaalokowania tablicy dwuwymiarowej (jako jednowymiarową) upewniając się że pamięć jest aligned do 512 bajtów.
Dla 8 bajtowych używam cudaMalloc sam się o to troszcząc.

-Daje gwarancje kompilatorowi że pamięć jest aligned do 512 bajtów w ten sposób:

```
__builtin_assume_aligned(curr_vertex_gpu, 512);
__builtin_assume_aligned(cont_gpu, 512);
__builtin_assume_aligned(l_gpu, 512);
__builtin_assume_aligned(GRAPH_R_gpu, 512);
__builtin_assume_aligned(GRAPH_C_gpu, 512);
__builtin_assume_aligned(OFFSET_gpu, 512);
__builtin_assume_aligned(VMAP_gpu, 512);
__builtin_assume_aligned(NVIR_gpu, 512);
__builtin_assume_aligned(streams_d_gpu, 512);
__builtin_assume_aligned(streams_sigma_gpu, 512);
__builtin_assume_aligned(streams_delta_gpu, 512);
```

Kilka strumieni wykonuje memset asychrocznie przed każdą rundą by wykonywać to jednocześnie.

Skoro nie używam shared memory (nad czym ubolewam ale nie wiem jak miałem to zrobić, korzystam bardzo z lokalnych rejestrów) to stosuje politykę by pamięć która jest rozdzielana na L1 cache i shared w proporcji normalnie 1 do 3 nakazuje by w całości poszła na L1 cache.

```
cudaFuncSetAttribute(next_round_preparation,
cudaFuncAttributePreferredSharedMemoryCarveout,
cudaSharedmemCarveoutMaxL1);
```

Robię unroll w updating_bc.h licząc globalne bc:

```
      #pragma unroll
      for (int i = 0; i < NUMBER_OF_STREAMS; i++) {
          bc_sum_elem += partials_bc[i * pitched_vertex_count +
threadId];
      }
```

Daje kompilatorowi kolejne gwarancje/expect ustawiając launch bounds mówiąc ile maksymalnie wątków będzie w bloku (bo ta liczba jest ustalona z góry oraz że chce przez prawie cały czas dać maksymalną możliwą przy tej ilości wątków aktywną liczbę bloków):

```
__launch_bounds__(BLOCK_SIZE_X, DESIRED_MIN_BLOCKS_PER_MULTIPROCESSOR)
```

Wszelkie parametry kerneli mówią kernelą że nie ma aliasów słowem __restrict__ co co pozwala kompilatorowi zoptymalizować takie same podwyrażenia bo wie że wskaźniki nie wskazują na siebie nawzajem i podwyrażenie nie będzie się w związku z tym zmieniać czego nie może sam założyć:

```
__global__
__launch_bounds__(BLOCK_SIZE_X, DESIRED_MIN_BLOCKS_PER_MULTIPROCESSOR)
void forward_step(int * __restrict__ GRAPH_R,
                  int * __restrict__ GRAPH_C,
                  int * __restrict__ OFFSET,
                  int * __restrict__ VMAP,
                  int * __restrict__ NVIR,
                  int * __restrict__ graph_data,
                  int * __restrict__ curr_vertex,
                  bool * __restrict__ cont,
                  int * __ restrict __ l,
                  int * __restrict__ d,
                  uint32_t * __restrict__ sigma,
                  float * __restrict__ time) {
```

Stałe dane jak ilość krawędzi czy wierzchołków mogłem dać do pamięci const, ale wycofałem się z tego ponieważ nie ma to absolutnie wpływu na nic dlatego że const może być cachowany w L1 i L2 a ja i tak zwalniam cały shared na cache L1 efektem tego patrząc na szybkość algorytmu prędkość jest taka sama ponieważ te dane i tak trafiają z globalnej pamięci do cache i tam pozostają. (const nie może być szybsza od shared/L1).

Sugeruje kompilatorowi również że dany if is expected to be true w dwóch krytycznych miejscach kroku forward i backward:

```
__builtin_expect (do_vertex, true);
  if (do_vertex) { ... }
```

Kompilator może te informacje wykorzystać do optymalizacji skoro wie że zazwyczaj należy w kodzie asemblera wykonać skok i ta instrukcja powinna zostać pobrana do pamięci.

Dalsza optymalizacja z której jestem dumny choć jest trywialna to:

```
  int threadId = blockIdx.x * BLOCK_SIZE_X + threadIdx.x;
```

Zamiast:

```
  int threadId = blockIdx.x * blockDim.x + threadIdx.x;
```

Ponieważ mnożenie jest kosztowne a wiemy że blockDim.x będzie zawsze taki sam i będzie potęgą dwójki więc można wstawić makro żeby przy kompilacji wstawił literał (tak kojarzę że makra tak działały) i wtedy kompilator widząc liczbę a nie zmienną która jest potęgą dwójki zamieni to na operację shiftu na bitach (więc na szczęście nie musimy sami liczyć o ile te bity trzeba przenieść i pisać tej operacji explicite). Podobnie byłoby dla modulo gdybym go

używał. Pobieżna analiza kodu asemblera cudy wskazuje że prawdopodobnie tak istotnie jest (operacja shl):

```
ld.param.u64    %rd7, [_Z12forward_stepPiS_S_S_S_S_S_PbS_S_PjPf_param_0];
ld.param.u64    %rd8, [_Z12forward_stepPiS_S_S_S_S_S_PbS_S_PjPf_param_1];
ld.param.u64    %rd9, [_Z12forward_stepPiS_S_S_S_S_S_PbS_S_PjPf_param_2];
ld.param.u64    %rd10, [_Z12forward_stepPiS_S_S_S_S_S_PbS_S_PjPf_param_3];
ld.param.u64    %rd11, [_Z12forward_stepPiS_S_S_S_S_S_PbS_S_PjPf_param_4];
ld.param.u64    %rd14, [_Z12forward_stepPiS_S_S_S_S_S_PbS_S_PjPf_param_5];
ld.param.u64    %rd12, [_Z12forward_stepPiS_S_S_S_S_S_PbS_S_PjPf_param_7];
ld.param.u64    %rd15, [_Z12forward_stepPiS_S_S_S_S_S_PbS_S_PjPf_param_8];
ld.param.u64    %rd13, [_Z12forward_stepPiS_S_S_S_S_S_PbS_S_PjPf_param_9];
ld.param.u64    %rd16, [_Z12forward_stepPiS_S_S_S_S_S_PbS_S_PjPf_param_10];
cvta.to.global.u64  %rd1, %rd16;
mov.u32     %r12, %ctaid.x;
shl.b32     %r13, %r12, 8;
mov.u32     %r14, %tid.x;
add.s32     %r1, %r13, %r14;
```

Kolejną optymalizacją która łączy się z ulepszeniem trzymania grafu jako Stride-CSR jest liczba sąsiadów wirtualnych wierzchołków. Zdecydowałem się nie stosować heurystyk by obliczyć jaka ta ilość mogłaby wynosić (na przykład średnia czy mediana) na podstawie ilości wierzchołków i krawędzi w grafie tylko ustawić na 8 ponieważ overhead jest już sensownie mały i musi to być wielokrotność dwójki ponieważ karta dla warpu (chyba warpu) pobiera dane z pamięci globalnej w rundach po 32 bajty i pewien transfer by się zmarnował (dzięki virtual vertices występuje memory coalescing). Jeśli chodzi o pamięć to profiler niestety chyba wskazywał na niewielkie użycie cachu ponieważ taka jest trochę natura tego algorytmu.

6. Dalsze możliwe optymalizacje:

Niestety na śmierć zapomniałem zebrać te strumienie w graf (zdefiniować raz workflow a nie za każdym razem). Myślę że miałoby to potencjał bo odpalam ogromną liczbę bardzo małych kerneli i jeśli z każdym wiąże się pewien koszt zdefiniowania obliczenia ten koszt może być istotny.

7. Uwagi:

Wyniki są w moim programie poprawne dla 7 grafów testowych i z dokładnością do błędów numerycznych dokładnie 2 razy większe niż wyniki dla tych samych grafów innych zaimplementowanych algorytmów liczących betweenness (np gephi).

Jeżeli wyniki będą dalej poprawne można dodać opcje optymalizacji poświęcające precyzje dla szybkości takie jak intrinsic less precision division i flushed denormalized numbers to zero: -ftz=true -prec-div=false -use_fast_math

Wyniki wypisuje w formie count << fixed i jest pełna wartość bez żadnych ‘e’.