Plotnikau Pavel

Semestrální práce z předmětu NI-GPU

Rok 2022/23

Úvod

Tato semestrální práce popisuje proces napsání algoritmů pro sestavení konvexní obálky pro CPU a GPU. K napsání CPU programu se používal programovací jazyk C++ a pro GPU program – jazyk CUDA. Oba programy používaly matematickou knihovnu GLM která slouží ke zjednodušení práce s vrcholy.

Sekvenční řešení

Popis algoritmu

Ke sestrojení konvexní obálky se používá variace algoritmu QuickHull:

1. K dispozici je buffer se všemi dostupnými vrcholy. Každý vrchol je označen jako nepatřící do konvexní obálky.
2. Z tohoto bufferu jsou vybrány dva vrcholy, které jsou globálním minimem a maximem vzhledem k ose X.
3. Oba vrcholy se označí jako součást obálky a hrana tvořená těmito dvěma vrcholy se zařadí do fronty.
4. Z fronty se vybere hrana a poté se hledá nejvzdálenější od ní vrchol z těch, které nesouvisejí s obálkou a jsou na opačné straně hrany než střed obálky.
5. Pokud takový vrchol neexistuje, algoritmus se vrátí ke kroku 4. V opačném případě bude aktuální hrana z fronty odstraněna a na konec fronty budou přidány dvě nové hrany s nalezeným vrcholem.
6. Zbývající vrcholy zahrnuté do nové konvexní obálky jsou poté označeny jako její součásti. Pokud po tomto kroku nezbyly žádné volné vrcholy, algoritmus dokončí svou práci, jinak se vrátí ke kroku 4.

Implementace

Před spuštěním hlavní smyčky se vytvoří buffer, do kterého se uloží indexy všech volných vrcholů. Přičemž dva vrcholy, z nichž je vytvořena první hrana, jsou umístěny na pravé straně této vyrovnávací paměti a všechny ostatní vrcholy jsou umístěny na levé straně. Do fronty jsou zařazeny dvě hrany s různým pořadím vrcholů, takže při první iteraci smyčky se zkontroluje horní polovina vrcholů a při druhé dolní polovina.

```
size_t min_vert_i = find_x_extrema_vert([](float ref, float next){ return ref > next; });
size_t max_vert_i = find_x_extrema_vert([](float ref, float next){ return ref < next; });
edge_queue.push_back({min_vert_i, max_vert_i});
edge_queue.push_back({max_vert_i, min_vert_i});

std::vector<size_t> vert_idxs(verts.size());
for (size_t i = 0; i < vert_idxs.size(); i++)
{
	vert_idxs[i] = i;
}
```

```
size_t vert_idxs_size = vert_idxs.size();
swap_indeces(vert_idxs, vert_idxs_size - 1, min_vert_i);
swap_indeces(vert_idxs, vert_idxs_size - 2, max_vert_i);
vert_idxs_size -= 2;
std::vector<float> distances(vert_idxs.size());
```

Po všech přípravách začíná hlavní cyklus, který lze rozdělit do čtyř částí:

- Po získání hrany z fronty se všem volným vrcholům přiřadí vzdálenost do této hrany. Přičemž vrcholům, které nejsou na správné straně hrany, bude přiřazena záporná vzdálenost.

```
for (size_t i = 0; i < vert_idxs_size; i++)
{
	size_t vert_i = vert_idxs[i];
	glm::vec2& T_vert = verts[vert_i];

	glm::vec3 AB_vec(B_vert - A_vert, 0.0f);
	glm::vec3 AT_vec(T_vert - A_vert, 0.0f);
	float A_cross = glm::cross(AB_vec, AT_vec).z;
	if (A_cross < 0)
	{
		distances[i] = -1;
		continue;
	}

	glm::vec3 T_proj = (glm::dot(AB_vec, AT_vec) / glm::dot(AB_vec, AB_vec)) * AB_vec;
	distances[i] = glm::length(AT_vec - T_proj);
}
```

- Poté se vyhledá vrchol, který je od hrany vzdálen nejvíce. Pokud je největší nalezená vzdálenost záporná, cyklus začne znovu s další hranou ve frontě, pokud nějaká existuje. V opačném případě bude aktuální hrana z fronty odstraněna a na její konec budou umístěny dvě nové hrany s nalezeným vrcholem.

```
size_t raw_C_vert_i = 0;
float farthest_distance = -1;
for (size_t i = 0; i < vert_idxs_size; i++)
{
	if (distances[i] > farthest_distance)
	{
		farthest_distance = distances[i];
		raw_C_vert_i = i;
	}
}

if (farthest_distance < 0)
{
	++edge_it;
	continue;
}

const size_t C_vert_i = vert_idxs[raw_C_vert_i];
const glm::vec2& C_vert = verts[C_vert_i];

edge_queue.push_back({A_vert_i, C_vert_i});
edge_queue.push_back({C_vert_i, B_vert_i});
edge_it = edge_queue.erase(edge_it);
```

- Dále se všem vrcholům, které mají kladnou vzdálenost do hrany, ale nejsou uvnitř nově vytvořeného trojúhelníku, přiřadí záporná vzdálenost, čímž se označí vrcholy, které nejsou součástí nové konvexní obálky.

```
for (size_t i = 0; i < vert_idxs_size; i++)
{
	size_t vert_i = vert_idxs[i];
	if (distances[i] > 0)
	{
		const glm::vec2& T_vert = verts[vert_i];

		glm::vec3 BC_vec(C_vert - B_vert, 0.0f);
		glm::vec3 CA_vec(A_vert - C_vert, 0.0f);

		glm::vec3 BT_vec(T_vert - B_vert, 0.0f);
		glm::vec3 CT_vec(T_vert - C_vert, 0.0f);

		float B_cross = glm::cross(BC_vec, BT_vec).z;
		float C_cross = glm::cross(CA_vec, CT_vec).z;

		if (!(B_cross >= 0 && C_cross >= 0))
		{
			distances[i] = -1.0f;
		}
	}
}
```

- Posledním krokem je seřazení pole indexů vrcholů tak, že vrcholy s kladnou vzdáleností hran budou považovány za součást nové konvexní obálky a budou přesunuty na pravou stranu pole a všechny ostatní, které nejsou součástí obálky, budou přesunuty na levou stranu. Pokud již nejsou žádné volné vrcholy, hlavní cyklus skončí a všechny hrany ve frontě se stanou součástí konvexní obálky.

```
int right_side = 0;
int left_side = vert_idxs_size - 1;
while (left_side >= right_side)
{
	if (distances[left_side] < 0 && distances[right_side] >= 0)
	{
		swap_indeces(vert_idxs, left_side, right_side);
		--left_side;
		++right_side;
	}
	else
	{
		if (distances[left_side] >= 0) { --left_side; }
		if (distances[right_side] < 0) { ++right_side; }
	}
}
vert_idxs_size = right_side;

if (vert_idxs_size == 0)
{
	break;
}
```

Paralelní řešení

Výpočet vzdálenosti u vrcholu

Nejprve je třeba vytvořit globální funkci, která jako argumenty přijímá pole vrcholů, pole vrcholových indexů, pole vzdáleností, oba vrcholy dané hrany a počet volných vrcholů.

Abych snížil množství použité paměti, rozhodl jsem se místo samotných vrcholů v hraně předávat jejich indexy. Dále aby se snížil počet přístupů do globální paměti, nulové vlákno v bloku zapíše tyto vrcholy do sdílené paměti. Všechna vlákna v bloku se pak synchronizují a pokračují ve výpočtu vzdálenosti.

Během výpočtu vzdálenosti bude každé vlákno postupně pracovat s několika vrcholy. Tím se optimalizuje počet použitých bloků a snižuje počet neaktivních vláken v bloku. Konečná funkce je následující:

```
__global__ void count_distances(
	glm::vec2* verts, uint32_t* vert_idxs, float* distances,
	uint32_t A_vert_i, uint32_t B_vert_i, uint32_t num_vert_idxs

)
{
	__shared__ glm::vec2 A_vert, B_vert;
	if (threadIdx.x == 0)
	{
		A_vert = verts[A_vert_i];
		B_vert = verts[B_vert_i];
	}
	__syncthreads();

	uint32_t blk_idx = ELEMS_PER_BLK * blockIdx.x;
	uint32_t max = fminf(blk_idx + ELEMS_PER_BLK, num_vert_idxs);
	for (uint32_t idx = blk_idx + threadIdx.x; idx < max; idx += BLK_DIM)
	{
		uint32_t vert_i = vert_idxs[idx];
		glm::vec2 T_vert = verts[vert_i];

		glm::vec3 AB_vec(B_vert - A_vert, 0.0f);
		glm::vec3 AT_vec(T_vert - A_vert, 0.0f);

		float A_cross = glm::cross(AB_vec, AT_vec).z;
		if (A_cross < 0)
		{
			distances[idx] = -1.0f;
		}
		else
		{
			glm::vec3 T_proj = (glm::dot(AB_vec, AT_vec) / glm::dot(AB_vec,
AB_vec)) * AB_vec;
			distances[idx] = glm::length(AT_vec - T_proj);
		}
	}
}
```

Získání vrcholu s maximální vzdáleností

První možností bylo napsat funkci, která najde maximální délku pomocí paralelní redukce. To by však vyžadovalo alokaci dalšího globálního bufferu a vícekrát spouštět kernel anebo pracovat pouze s jedním blokem vláken. Místo toho bylo rozhodnuto implementovat tuto logiku prostřednictvím atomické funkce.

Hlavní problém spočíval v tom, že standardní knihovna CUDA neposkytuje atomickou funkci pro hledání maxima u čísel s plovoucí desetinnou čárkou. Navíc bylo nutné kromě vzdálenosti přenášet také informace o indexu vrcholu. Proto bylo nutně napsat vlastní atomickou funkci, která by mohla nejen porovnávat čísla s desetinnou čárkou, ale i přenášet informaci o vrcholu, jehož distanci porovnáváme. Pro vytvoření nové atomické funkce je třeba použít funkci atomicCAS, která porovná předpokládanou starou hodnotu se starou hodnotou, která se právě nachází v paměťové buňce, a pokud se shodují, zapíše do buňky novou hodnotu. Tato funkce může přijímat hodnoty o velikosti až 64 bitů (unsigned long long int). To znamená, že je možné zakódovat jakoukoli informaci v rozsahu 8 bajtů a aktualizovat ji pomocí atomické funkce. V tomto případě to bude vzdálenost k vrcholu (float – 4 bajt) a informace o indexu vrcholu (uint32_t – 4 bajt). Konečný kód vypadá takto:

```
typedef union {
	struct { float distance; uint32_t raw_idx; } pair;
	unsigned long long int ulong;
} dist_idx_pair;

__device__ unsigned long long int atomicDistMax(unsigned long long int* addr, float
distance, uint32_t raw_idx)
{
	dist_idx_pair loc, loctest;
	loc.pair.distance = distance;
	loc.pair.raw_idx = raw_idx;
	loctest.ulong = *addr;
	while (loctest.pair.distance < distance)
	{
		loctest.ulong = atomicCAS(addr, loctest.ulong, loc.ulong);
	}
	return loctest.ulong;
}
```

Díky union vzniká struktura, ve které je dvojice potřebných 4-bajtových hodnot uložena do stejné buňky s 8-bajtovým integerem, což dovoluje přepínat mezi nimi v průběhu práce.
Funkce má pak velmi jednoduchou logiku. Před spuštěním hlavní smyčky je alokována paměťová buňka, do které se zapíše výsledek vyhledávací funkce. Před každým spuštěním této funkce je třeba buňku uvést do defaultního stavu.
Ve funkci samotné každé vlákno porovná několik vrcholů, snaží se mezi nimi najít maximum a zapíše ho do své lokální proměnné. Poté se pokusí zapsat svůj výsledek do sdílené proměnné bloku prostřednictvím dříve napsané atomické funkce. Poté, co se všechna vlákna pokusí zapsat svůj výsledek do sdílené proměnné, pokusí se nulové vlákno zapsat tento výsledek do globální paměti pomocí té samé atomické funkce. Výsledkem je následující kód:

```
__global__ void to_default_state(dist_idx_pair* g_res)
{
	g_res->pair.distance = -1.0f;
}

__global__ void find_max_distance(float* distances, uint32_t num_vert_idxs,
dist_idx_pair* g_res)
{
	__shared__ dist_idx_pair shr_res;
	if (threadIdx.x == 0)
	{
		shr_res.pair.distance = -1.0f;
	}
	__syncthreads();

	dist_idx_pair loc_res;
	loc_res.pair.distance = -1.0f;

	uint32_t blk_idx = ELEMS_PER_BLK * blockIdx.x;
```

```
	uint32_t max = fminf(blk_idx + ELEMS_PER_BLK, num_vert_idxs);
	for (uint32_t idx = blk_idx + threadIdx.x; idx < max; idx += BLK_DIM)
	{
		float tmp_dist = distances[idx];
		if (tmp_dist > loc_res.pair.distance)
		{
			loc_res.pair.distance = tmp_dist;
			loc_res.pair.raw_idx = idx;
		}
	}

	atomicDistMax(&shr_res.ulong, loc_res.pair.distance, loc_res.pair.raw_idx);
	__syncthreads();

	if (threadIdx.x == 0)
		atomicDistMax(&(g_res->ulong), shr_res.pair.distance, shr_res.pair.raw_idx);
}
```

Po ukončení kernelu zkopíruje kód hlavního programu výsledek do paměti CPU a podle toho, zda byl nalezen vrchol, pokračuje ve vykonávání programu nebo přejde k další iteraci hlavního cyklu.

Kontrola příslušnosti vrcholů ke konvexní obálce

Logika této funkce je velmi podobná logice popsané v části o výpočtu vzdáleností k vrcholům, proto ji zde nebudeme popisovat. Výsledná funkce vypadá takto:

```
__global__ void mark_inner_vets(
	glm::vec2* verts, uint32_t* vert_idxs, float* distances,
	uint32_t A_vert_i, uint32_t B_vert_i, uint32_t C_vert_i,
	uint32_t num_vert_idxs
)
{
	__shared__ glm::vec2 A_vert, B_vert, C_vert;
	if (threadIdx.x == 0)
	{
		A_vert = verts[A_vert_i];
		B_vert = verts[B_vert_i];
		C_vert = verts[C_vert_i];
	}
	__syncthreads();


	uint32_t blk_idx = ELEMS_PER_BLK * blockIdx.x;
	uint32_t max = fminf(blk_idx + ELEMS_PER_BLK, num_vert_idxs);
	for (uint32_t idx = blk_idx + threadIdx.x; idx < max; idx += BLK_DIM)
	{
		if (distances[idx] > 0)
		{
			uint32_t vert_i = vert_idxs[idx];
			glm::vec2 T_vert = verts[vert_i];

			glm::vec3 BC_vec(C_vert - B_vert, 0.0f);
			glm::vec3 CA_vec(A_vert - C_vert, 0.0f);

			glm::vec3 BT_vec(T_vert - B_vert, 0.0f);
			glm::vec3 CT_vec(T_vert - C_vert, 0.0f);
```

```
                        float B_cross = glm::cross(BC_vec, BT_vec).z;
                        float C_cross = glm::cross(CA_vec, CT_vec).z;

                        if (!(B_cross >= 0 && C_cross >= 0))
                        {
                                distances[idx] = -1.0f;
                        }
                }
        }
}
```

Třídění indexů vrcholů

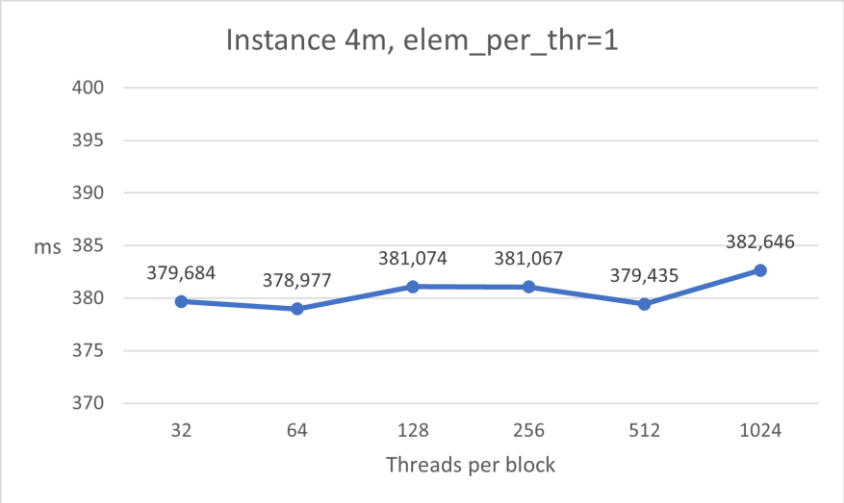
Tato část je nesmírně důležitá, protože snižuje počet vrcholů, se kterými bude algoritmus v budoucnu pracovat. Bohužel se mi v tuto chvíli nepodařilo optimalizovat tuto část tak, aby fungovala na GPU, což z ní v budoucnu dělá úzké hrdlo celého algoritmu.
Jedinou změnou, která zde byla přidána, je zkopírování bufferu vzdáleností do paměti CPU a zkopírování setříděného bufferu s indexy vrcholů zpět do paměti GPU.
Tuto část je však možné optimalizovat pomocí paralelní verze třídění. Tím by se také snížil počet přenosů dat mezi procesory, což by mělo pozitivní vliv na dobu provádění algoritmu.

Měření

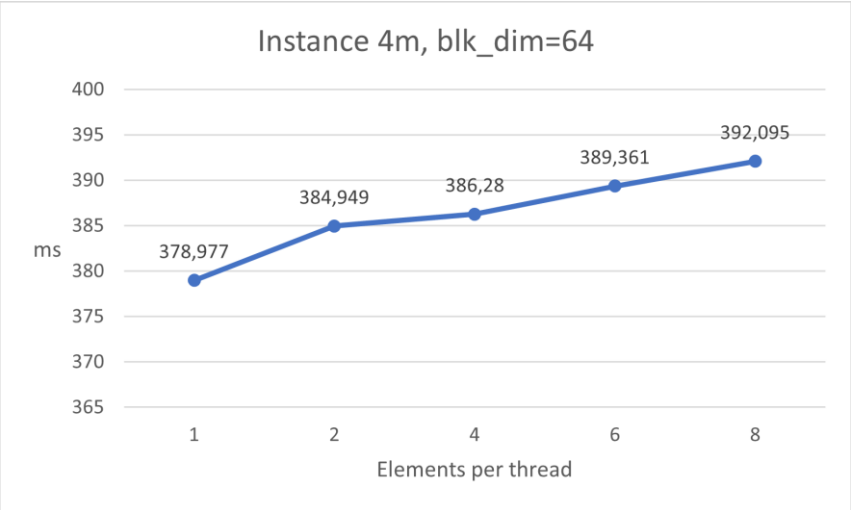
Pro závěrečná měření byly vygenerovány 4 instance vrcholů: 4 000, 40 000, 400 000 a 4 miliony vrcholů. Program má dva parametry, které přímo ovlivňují počet bloků používaných jádry:

- BLK_DIM – počet vláken v jednom bloku
- ELEMS_PER_THR – maximální počet prvků zpracovávaných jedním vláknem

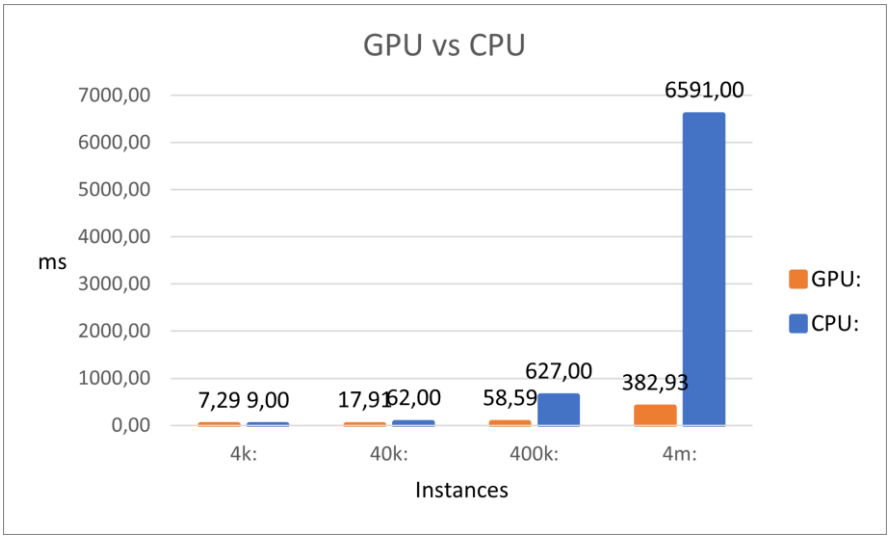
Nejprve je třeba zjistit, při jakém počtu vláken na blok poběží program nejoptimálněji. Počet položek na vlákno bude pevně stanoven na 1. Testování bude provedeno na instanci se 4 miliony vrcholů. Výsledky jsou následující:



Jak je vidět, ačkoli počet bloků klesá s rostoucím počtem vláken na blok, časový rozdíl není příliš velký a pohybuje se kolem 1-2 ms.
Dalším krokem je výběr optimálního počtu prvků na proces. Instance je stejná jako v předchozím experimentu a počet vláken na blok bude 64. Výsledky jsou následující:



Na grafu je vidět, že s rostoucím počtem prvků na vlákno se doba trvání hlavního cyklu postupně zvyšuje v průměru o cca 3-4 ms. Vzhledem k úzkému hrdlu ve formě dvou přenosů paměti mezi procesory a neparalelnímu zjednodušenému třídicímu algoritmu je poměrně obtížné určit příčinu tohoto chování algoritmu.
V závěrečném testu se porovná výkon GPU verze programu s CPU verzí na všech instancích. Parametry vybrané pro GPU verzi jsou následující: BLK_DIM=64, ELEMS_PER_THR=1.



Jak je vidět z grafu, rozdíl v rychlosti algoritmů se zvyšuje s rostoucí velikostí instance. Pro instanci velikosti 4 milionů vrcholů je tedy algoritmus pro grafickou kartu přibližně 17krát rychlejší než originální algoritmus.

Závěr

Algoritmus napsaný s využitím grafické karty ukázal výrazné zlepšení výkonu oproti původnímu algoritmu pro CPU. Výsledný program však není dokonalý a může být vylepšen. Kromě již zmíněné nevylepšené části hlavního cyklu je problémem také příprava bufferů před spuštěním. Ačkoli tato část algoritmu nebyla v žádné z verzí programu zapojena do závěrečných měření, může být také urychlena grafickou kartou.

Díky výše uvedeným vylepšením je tedy možné téměř kompletně přenést algoritmus na grafickou kartu, čímž se taky výrazně sníží počet paměťových přenosů a zůstane pouze kopírování výsledku hledání maxima z paměti GPU do paměti CPU.