Jihočeská univerzita v Českých Budějovicích
Přírodovědecká fakulta

Možnosti GPU Computingu v MATLABu

Bakalářská práce

Filip Hrubý

Školitel: doc. Dr.rer.nat. Jan Valdman

České Budějovice 2022

Bibliografické údaje:

Hrubý, F., 2022: Možnosti GPU Computingu v MATLABu. [GPU Computing capabilities in MATLAB. Bc. Thesis, in Czech.] - 39 p., Faculty of Science, University of South Bohemia, České Budějovice, Czech Republic.

Anotace

Cílem této práce je zjištění možností provádění výpočtů na GPU v MATLABu a jejich otestování v oblasti geometrického modelování a aplikace metody konečných prvků. Práce se nejdříve zabývá srovnáním testovacích prostředí, které jsou poté použité pro rychlostní testy. Dále popisuje sestavení trojúhelníkové sítě na CPU a výpočtu matice hmotnosti a tuhosti za použití CPU a GPU. Jsou porovnány časy sestavení a je ukázáno možné využití při výpočtu rovnice lineární elasticity.

Annotation

This thesis aims to investigate the possibilities of GPU computing in MATLAB with applications to geometric modeling and computations of the finite element method. First, we deal with test environments that are later used for the speed comparison. Furthermore, a triangular mesh is generated on CPU and mass and stiffness matrices are assembled on CPU and GPU. Evaluation times are compared and a the application in calculations of linear elasticity is demonstrated.

Prohlášení

Prohlašuji, že jsem autorem této kvalifikační práce a že jsem ji vypracoval pouze s použitím pramenů a literatury uvedených v seznamu použitých zdrojů.

V Českých Budějovicích
Dne: 5. prosince 2022

. . . .
Filip Hrubý

Klíčová slova

MATLAB, GPU, trojúhelníkové sítě, metoda konečných prvků, sestavení matice tuhosti, sestavení matice hmotnosti

Key worlds

MATLAB, GPU, triangle mesh, finite element method, stiffness matrix assembly, mass matrix assembly

Poděkování

Rád bych touto cestou poděkoval vedoucímu bakalářské práce panu doc. Dr.rer.nat. Janu Valdmanovi za jeho trpělivost, cenné rady a odborné vedení práce, jak při vysvětlování matematické části tak během vzniku písemné části práce a především za jeho ochotu kdykoliv pomoci.

Dále bych chtěl poděkovat panu Ing. Janu Feslovi, Ph.D. za poskytnutí výpočetního serveru a pomoc při jeho zprovoznění.

Také bych rád poděkoval své rodině a přátelům za přetrvávající podporu.

Obsah

1 Úvod 1

2 Testovací prostředí 2
2.1 Hardware (GPU, CPU) . . . 2
2.2 Software (CUDA, MATLAB) . . . 4

3 GPU programování v MATLABu 7

4 Struktura ukládání trojúhelníkových sítí 11
4.1 Struktura pole *coordinates* . . . 11
4.2 Struktura *elements* . . . 12
4.3 Struktura *coord* . . . 12
4.4 Uniformní zjemnění sítě . . . 13

5 Výpočet plochy rovinného obrazce 14
5.1 Gaussova metoda (shoelace method) . . . 14
5.2 Heronův vzorec . . . 14
5.3 Pomocí determinantu 2x2 nebo 3x3 matice . . . 15

6 Vytvoření trojúhelníkové/tetrahedrální sítě 16
6.1 Delaunayho triangulace ve 2D . . . 16
6.2 Triangulace pomocí PDE (Partial Differential Equation) Toolbox ve 2D . . . 16
6.3 Nahrání sítě z 3D modelu . . . 18

7 Vektorizace 20
7.1 Implementace uložení trojúhelníků ve 2D . . . 20

8 Metoda konečných prvků a její využití 23
8.1 Sestavení matice tuhosti . . . 23
8.1.1 Společná část sestavení matice tuhosti . . . 23
8.1.2 Část specifická pro Poissonovu rovnici . . . 24
8.1.3 Část specifická pro elasticitu . . . 25
8.1.4 Zrychlení sestavení matice tuhosti . . . 26
8.2 Sestavení matice hmotnosti . . . 28
8.3 Výpočet elasticity . . . 28
8.3.1 Zrychlení výpočtu úlohy elasticity . . . 31

9 Práce s vytvořeným frameworkem 33

10 Závěr 36

1. Úvod

V dnešní uspěchané době hledáme, kde ušetříme čas. Všechny záležitosti chceme mít co nejrychleji vyřešené, věci co nejrychleji k dispozici a procesy s nejkratší prodlevou. K záležitostem patří například stání ve frontě. U věcí je v dnešní době důležité například doručení on-line objednávky, kde doba a místo doručení je často rozhodujícím kritériem při výběru služby. A k procesům patří například zrychlení linky o počet kusů za minutu.

Vývoj dnešní techniky dospěl do stádia, kde není již mnoho důvodů pro zvýšení výpočetního výkonu počítače pro domácí nebo kancelářské využití. Vývoj se tedy v posledních letech více zaměřil na zvyšování výkonu GPU (grafic processing unit). To souvisí i s čím dál častější potřebou zpracovávat rychle velké množství dat. K tomu se dnes již běžně využívají grafické karty. Pod pojmem grafické karty si zajisté mnoho lidí vybaví hlavně herní a grafický průmysl, ale to už dlouho není pravda. Grafické karty se zejména využívají pro distribuované výpočty nebo pro v dnešní době populární těžení krypto měn. To, aby se grafické karty mohly využívat pro jiné věci než filmy a hry, umožnila technologie CUDA (Compute Unified Device Architecture) vyvinutá společností NVIDIA. CUDA podporuje různé jazyky a API, jako Fortran, OpenCL, C nebo MATLAB.

Možností, jak urychlovat výpočty, je v dnešní době několik. Kromě hardwarové akcelerace lze výpočty urychlovat pomocí matematické algoritmické optimalizace, paralelismem či vektorizací. Tedy upravit způsob matematického výpočtu tak, abychom prováděli výpočet optimalizovaný pro počítač a ne člověka nebo upravit algoritmus, aby prováděl méně kroků a každý krok byl atomický. Všechny tyto metody lze použít v interaktivním programovém prostředí a skriptovacím jazyku MATLAB. Je optimalizovaný pro analýzu dat, tvorbu algoritmů a modelů. Používá se především pro vědecké a vědeckotechnické účely. MATLAB také umožňuje bez hlubších znalostí jazyka CUDA využívat grafické karty NVIDIA pro své výpočty.

V tom případě je MATLAB ideálním prostředí pro kombinaci těchto metod a hardwarové akcelerace. Na jednoduchých skriptech porovnáme časové náročnosti jednotlivých matematických výpočtů například z oblastí geometrického modelování či výpočtu metody konečných prvků.

Čím přesnější výstup z metody konečných prvků požadujeme, tím více času a paměti je třeba k výpočtu. Metoda konečných prvků funguje na principu rozdělení objektu na konečný počet bodů, se kterými se dále pracuje. Pro rozdělení 2D, ale i 3D objektu na body se využívá takzvané meshování. Tedy vytvoření sítě bodů, jenž popisuje daný objekt. Čím hustější síť je (čím více bodů má), tím přesnější popis objektu dostaneme a poté i přesnější výsledek výpočtu.

V této práci se tedy budu zabývat tím, jak využít možnosti GPU computingu v MATLABu ke zrychlení sestavení matic tuhosti a hmotnosti. Tyto matice následně využijeme v metodě konečných prvků. Vytvořený framework v MATLABu s kódy je k nalezení na:
Matlab-optimalization-with-GPU.

2. Testovací prostředí

Jelikož je cílem vytvořit univerzální framework pro rychlé sestavení matice tuhosti a hmotnosti za pomocí grafických karet NVIDIA, otestujeme výsledky na více zařízeních, tím získáme co nejobecnější data.

2.1 Hardware (GPU, CPU)

Nejen software, ale i hardware je důležitý pro rychlejší výpočty. V této kapitole si povíme o něco hardwaru, na kterém se testovaly výstupy a zásadní rozdíly mezi prací GPU a CPU.

CPU

Centrální procesorová jednotka je hlavní výpočetní prvek počítače. Operační systém, ovladače, aplikace nebo vlastní programy běží za pomoci CPU, není-li řečeno jinak. Proto si vysvětlíme základní parametry procesoru, abychom si mohli lépe představit jeho výpočetní sílu.

Jádra: Každé procesorové jádro může vykonávat operace samostatně nebo jádra mohou spolupracovat paralelně [1].

Vlákna: Vlákno je malá sada instrukcí, která má být naplánována a provedena procesorem nezávisle na nadřazeném procesu [2].

Frekvence: Procesor zpracovává každou sekundu mnoho instrukcí z různých programů. Frekvence měří počet cyklů, který procesor vykoná za sekundu [3].

Posílená frekvence: Frekvence měří optimální počet cyklů, posílená frekvence (boost) měří maximální počet cyklů, kterou může CPU dosáhnou při velkém zatížení [3].

Mezipaměť: Je množství paměti, kterou disponuje CPU. Využívá se ke snížení času přístupu do operační paměti PC, pomocí ukládaní mezi výpočtů a opakovaně využívaných dat [4].

Testování se uskutečnilo na dvou rozdílných procesorech. Jeden procesor určen pro notebooky a jeden pro servery.

	Intel(R) Core(TM) i7-8565U	AMD EPYC 7313
Platforma	Notebook	Server
Jádra	4	16
Vlákna	8	32
Frekvence	1.80 GHz	3.00 GHz
Posílená frekvence	až 4.60 GHz	3.70 GHz
Mezipaměť	8 MB	128 MB

Tabulka 2.1: Procesory užité k testování.

GPU

Grafický procesor je jedním z klíčových typů výpočetní techniky jak pro osobní a firemní počítače, tak pro herní konzole nebo výpočetní clustery. GPU je určen pro paralelní zpracování. Využívá se v řadě oblastí, mezi které patří grafika a vykreslování videa. Ačkoliv jsou proslaveny pro své schopnosti v herním průmyslu, stávají se populárnější ve spojení s umělou inteligencí.

Na počátku byly grafické procesory navrženy k urychlení vykreslování 3D grafiky. Postupem času se staly flexibilnějšími a programovatelnějšími, což rozšířilo jejich možnosti. Například jsou zajímavější vizuální efekty, nebo vytváří realistické scény s pokročilými technikami osvětlení, stínování či odrazu. Výkon grafických procesorů se začal také využívat v oborech vysoce výkoných výpočtů (HPC), hlubokého učení a těžení krypto měn [5].

CUDA jádra: Jedno CUDA jádro je podobné CPU jádru, ale není tak rozvinuté. Místo toho jsou implementována ve velkém počtu a navržena tak, aby zvládalo více výpočtů najednou [6].

Frekvence: GPU zpracovává každou sekundu mnoho instrukcí z různých programů. Frekvence měří počet cyklů, který GPU vykoná za sekundu [3].

Posílená frekvence: Frekvence měří optimální počet cyklů, posílená frekvence (boost) tedy měří maximální počet cyklů, kterou může dosáhnou při velkém zatížení [3].

RAM: Operační paměť grafické karty v našem případě znační, kolik paměti může využít k výpočtům na GPU.

	NVIDIA GeForce GTX 1650 Max-Q	NVIDIA A30 PCIe
Architektura	Ampere (2020-2022)	Turing (2018-2021)
Platforma	Notebook	Server
CUDA jádra	1024	3584
Frekvence	1020 MHz	930 MHz
Posílená frekvence	1245 MHz	1440 MHz
RAM	4 GB	24 GB

Tabulka 2.2: Grafické karty užité k testování.

Rozdíl mezi CPU a GPU

Grafický procesor obsahuje mnoho jader a využívá je k paralelním výpočtům. GPU rozdělí složitý problém na tisíce nebo milióny menších samostatných úloh, které pak najednou vyřeší. Naproti tomu CPU je rychlý, všestranný a provádí velkou řadu úloh vyžadující značnou součinnost. Z architektonického hlediska se procesor skládá jen z několika jader s velkým množstvím mezipaměti, která dokáže zpracovat několik softwarových vláken najednou. Oproti tomu GPU se skládá ze stovek jader, která mohou zpracovávat tisíce vláken současně [7].

CPU	GPU
Několik jader	Mnoho jader
Nízká latence	Vysoká propustnost
Optimalizováno pro sériové zpracování	Optimalizováno pro paralelní zpracování
Může provádět několik operací najednou	Může provádět tisíce operací najednou

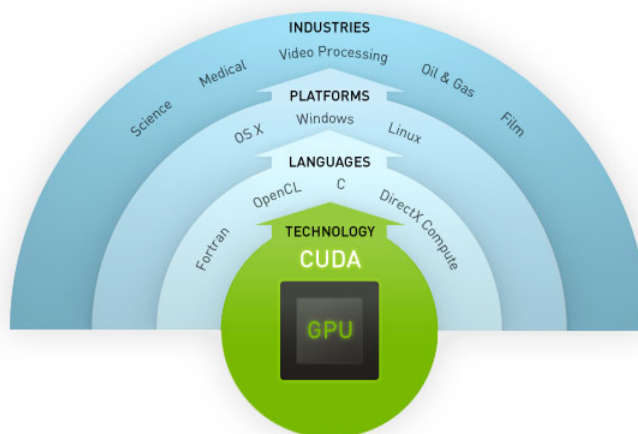
Tabulka 2.3: CPU vs GPU [7].

2.2 Software (CUDA, MATLAB)

Pod pojmem software rozumíme veškeré počítačové programy a soubory. I sebelepší hardware (fyzický počítač) nemůže provést operaci bez příslušného software, ale i sebelepší hardware nepomůže, když je software špatně napsaný. Pro zjištění zrychlení za pomocí GPU potřebujeme mít na všech testovacích stanicích jednotnou verzi softwaru. Kromě stejného kódu musíme mít i stejnou verzi MATLAB a ovladačů CUDA. V našem případě se všechny testy odehrávaly v MATLAB verze 9.12.0.1956245 (R2022a) Update 2 s ovladači CUDA 11.4.

CUDA

Je paralelní výpočetní platforma a programovací model vytvořený společností NVIDIA. Pomáhá vývojářům urychlit jejich aplikace využitím výkonu GPU. Kromě zrychlování vysoce výkoných výpočtů (HPC) se CUDA velice rozšířila ve spotřebitelských a průmyslových ekosystémech. Například farmaceutické společnosti využívají CUDA při objevování nových slibných léčebných postupů a automobilky ji využívají k vylepšení autonomního řízení. [8]



Obrázek 2.1: Technologie CUDA. Převzato z [9].

Z Obrázku 2.1 je patrné, že CUDA podporuje různé jazyky a API, jako Fortran, OpenCL, C, DirectX Compute, ale i MATLAB. O úroveň výše lze vidět, že CUDA je podporovaná na všech běžných operačních systémech [9].

IDE MATLAB

Neboli vývojové prostředí, je určeno pro práci s maticovou matematikou a jazykem MATLAB. Velkou výhodou prostředí MATLAB je velké množstvím profesionálně vytvořených, testova-

ných a dokumentovaných knihoven tzv. toolbox, z různých matematických a informatických odvětví. Toolboxy se jednoduše dají doinstalovat přímo v prostředí.

Seznam některých toolboxů [10]:

- **Deep learning toolbox** obsahuje framework pro návrh a implementaci hlubokých neuronových sítí s algoritmy a před-trénovanými modely. Dá se využít na konvoluční neuronové sítě a sítě s dlouhou krátkodobou pamětí k provádění klasifikace a regrese a další funkce.
- **Image Processing Toolbox** poskytuje rozsáhlou sadu standardních algoritmů a aplikací na zpracování, analýzu, vývoj a vizualizaci algoritmů. Může provádět segmentaci a vylepšení obrazu, redukci šumu, geometrickou transformaci.
- **Mapping Toolbox** nabízí algoritmy a funkce pro transformaci geografických dat a vytváření mapových zobrazení. Umožňuje vizualizovat data v geografickém kontextu a transformovat data z různých zdrojů do konzistentního geografického souřadnicového systému.
- **Financial Toolbox** přináší funkce pro matematické modelování a statistickou analýzu finančních dat. Můžete analyzovat, zpětně testovat a optimalizovat investiční portfolia s ohledem na obrat a další.

Seznam využitých toolboxů:

- **Parallel computing toolbox** umožňuje řešit výpočetně náročné problémy pomocí vícejádrových a grafických procesorů. Obsahuje vysokoúrovňové konstrukce jako jsou paralelní smyčky `for`,speciální typy polí a paralelizovat aplikace s využitím GPU bez CUDA programování. Tento toolbox je základem pro GPU programování v MATLAB.
- **Partial Differential Equation Toolbox** poskytuje funkce pro řešení stavební mechaniky, přenosu tepla a obecných parciálních diferenciálních rovnic (PDE). Tento toolbox byl využit pro vytváření tetrahedrálních sítí viz. Obrázek 6.1 (vpravo).

Profiler

Profiler je velmi užitečný nástroj pro časovou optimalizaci kódu. Tento nástroj umožňuje vývojáři lokalizovat místa v kódu, kde program při běhu stráví nejvíce času. Poté, co zjistíte, které funkce spotřebovávají nejvíce času, je můžete vyhodnotit z hlediska možného zlepšení výkonu [11]. Příklad výstupu je uveden v Obrázku 2.2. Červená místa označují podle intenzity nejvíce náročné části programu. Odpovídající časy v sekundách jsou uvedeny v prvním červeném sloupci vlevo a počet volání v druhém modrém sloupci. Výhodou je, že je rovnou integrován ve vývojovém prostředí MATLAB.

```
Time     Calls  Line
                 77  function e = energy(v,eps)
< 0.001   340    78      if nargin==1   % global energy only
4.181     170    79          v elems = v(mesh.elems2nodes);               % values on elements
2.609     170    80          F elems = evaluate F 2D scalar(mesh,v elems);   % all gradients
2.176     170    81          densities elems = densities(v elems,F elems);
0.361     170    82          e = sum(mesh.areas.*densities elems);
< 0.001   170    83      else   % local gradient energies using the epsilon perturbation vector
0.006     170    84          e = zeros(numel(dofsMinim),size(eps,2));
11.220    170    85          v patches = v(patches.elems2nodes);
< 0.001   170    86          for comp=1:size(eps,2)
2.772     340    87              v patches eps = v patches + eps(comp)*patches.logical;
15.213    340    88              F patches = evaluate F 2D scalar(patches,v patches eps);
12.610    340    89              densities patches = densities(v patches eps,F patches);
0.995     340    90              e patches = patches.areas.*densities patches;
2.489     340    91              cumsum all e = cumsum(e patches);
2.629     340    92              e(:,comp) = [cumsum all e(indx(1)); diff(cumsum all e(indx))];
0.001     340    93          end
< 0.001   340    94      end
3.326     340    95  end
```

Obrázek 2.2: Příklad výstupu MATLAB profileru. Převzato z [12].

Jazyk MATLAB

Patří k programovacím jazykům, který umožňují bez větší přípravy numericky řešit matematické úlohy, podobný jazyk je například Octave. Výhodou je velká řada předdefinovaných matematických a programátorských funkcí, které může programátor ihned využít k řešení dané úlohy. Na rozdíl od klasických programovacích jazyků, jako jsou C/C++, Java, C# a další, programátor nemusí deklarovat datový typ proměnné. Chceme-li za užití běžného programovacího jazyka do reálné proměnné (např. double) uložit druhou odmocninu ze záporného čísla, pak dostaneme chybu, za to MATLAB spočítá správný výsledek a uloží komplexní číslo [13].

3. GPU programování v MATLABu

V předchozí sekci jsme si pověděli něco málo o MATLABu v této sekci si vysvětlíme jak pracovat s GPU při výpočtech. MATLAB poskytuje možnost GPU programování na grafických kartách NVIDIA, využitím Parallel Computing Toolbox. Umožňuje vytvářet pole přímo na grafické kartě. Pokud se již pracuje s takovými poli, pak v MATLABu již existuje mnoho nativně implementovaných funkcí, které při zavolání automaticky pracují na GPU. Kdykoli zavoláme některou z těchto funkcí s alespoň jedením vstupním argumentem `gpuArray`, výpočet funkce poté poběží na GPU. Pokud má funkce výstupní argument pole, pak toto pole bude `gpuArray`. Ve stejném volání funkce můžeme kombinovat vstupní argumenty, které jsou klasické proměnné, pole využívající jak `gpuArray` a pole uložená v operační paměti [14].

gpuDevice

Než začneme s urychlováním výpočtů za pomocí GPU, je třeba ověřit, že tuto možnost máme. K tomu nám slouží funkce `gpuDevice` , kterou můžeme použít ke kontrole vlastností zařízení GPU. Pokud máme přístup k více GPU, použijeme funkci `gpuDevice` pro výběr konkrétního zařízení GPU, na kterém chceme provádět kód [15].

```
CUDADevice with properties:

                      Name: 'NVIDIA A30'
                     Index: 1
         ComputeCapability: '8.0'
            SupportsDouble: 1
             DriverVersion: 11.4000
            ToolkitVersion: 11.2000
        MaxThreadsPerBlock: 1024
          MaxShmemPerBlock: 49152
        MaxThreadBlockSize: [1024 1024 64]
               MaxGridSize: [2.1475e+09 65535 65535]
                 SIMDWidth: 32
               TotalMemory: 2.5437e+10
           AvailableMemory: 2.4895e+10
       MultiprocessorCount: 56
              ClockRateKHz: 1440000
               ComputeMode: 'Default'
      GPUOverlapsTransfers: 1
    KernelExecutionTimeout: 0
          CanMapHostMemory: 1
           DeviceSupported: 1
           DeviceAvailable: 1
            DeviceSelected: 1
```

Obrázek 3.1: Výstup funkce *gpuDevice*.

gpuArray

Objekt `gpuArray` představuje pole uložené v paměti GPU. Takové pole se dá vytvořit například následovně:

```
1 % example_1_gpuArrayDeclaration.m
2 X = [1 2 3 4 5];
3 X_GPU = gpuArray(X);
```

X je klasické pole uložené v operační paměti počítače. Pokud bude pole *X* metodě `gpuArray` předáno v argumentu metoda pole vezme a vytvoří ho na GPU. Nově vytvořené pole na GPU je poté vráceno výstupním parametrem metody.

Možnosti vytvoření objektu gpuArray

Jak již bylo ukázáno v sekci gpuArray, jednou z možností jak vytvořit pole na GPU je volání metody `gpuArray`. Existují, ale i další možnosti jak `gpuArray` vytvořit. V seznamu níže jsou zvýrazněné funkce, které umožňují vytvářet pole s daty přímo na GPU. Těmto funkcím stačí přidat argument v této podobě: `'gpuArray'`.

Seznam některých funkcí, které generují data přímo na GPU:

- **zeros** Funkce generuje matici naplněnou nulami o zadaných rozměrech.
- **ones** Funkce generuje matici naplněnou jedničkami o zadaných rozměrech.
- **eye** Funkce generuje jednotkovou (identickou) matici o zadaných rozměrech

Funkce gather

Tato funkce umožňuje převést `gpuArray` zpět do operační paměti počítače. Ne všechny funkce jsou schopné využít `gpuArray`, proto je čas od času vhodné využít funkci `gather` na získání zpět do operační paměti a vykonání dané funkce [16, 17].

```
1 % example_1_gpuArrayDeclaration.m
2 X = [1 2 3 4 5];
3 X_GPU = gpuArray(X);
4 Z = gather(X_GPU);
```

Pre-alokace

Změna velikosti pole je nákladná, protože zahrnuje dealokaci nebo alokaci paměti a kopírování hodnot při každé změně velikosti. Před alokováním matic, můžeme dosáhnout poměrně výrazného zrychlení [16].

```
1 % example_2_memoryAlocation.m
2 tic % zmena velikosti
3 a = 1; a(2) = 2;
4 a(3) = 3; a(4) = 4;
5 toc
6 tic % pred-alokace
7 b = zeros(4,1);
8 b(1) = 1;  b(2) = 2; b(3) = 3; b(4) = 4;
9 toc
```

Výsledek urychlení je pak zřejmý:

```
Elapsed time is 0.006489 seconds.
Elapsed time is 0.004104 seconds.
```

To samé platí i pro GPU pole.

```
% example_3_memoryAlocationGPU.m
tic % zmena velikosti
a = 1; a = gpuArray(a);
a(2) = 2; a(3) = 3; a(4) = 4;
toc
tic % pred-alokace
b = zeros(4,1, 'gpuArray');
b(1) = 1;  b(2) = 2; b(3) = 3; b(4) = 4;
toc
```

```
Elapsed time is 0.035863 seconds.
Elapsed time is 0.007357 seconds.
```

Proto budeme často nejprve alokovat pole, které bude dostatečně velké pro uložení dat, a tím bude program zrychlen, jelikož bude bez realokace paměti [16].

Režie při použití GPU

Při použití GPU nesmíme zapomenout na režii přenosu dat. Za běžných podmínek MATLAB využívá k výpočtům CPU a operační paměť počítače. Ale jelikož je GPU připojeno k CPU pomocí PCI sběrnice, pak se data musí z operační paměti přesunout po sběrnici do paměti GPU [16].

```
% example_4_resourcesCosts.m
n = 100; % pocet prvku
X = linspace(1, 10, n);
tic; Y = sin(X); toc;
tic;
X_GPU = gpuArray(X);
Y_GPU = sin(X_GPU);
toc; % ziskani dat z GPU zpet na operacni pamet PC
```

```
>> resourcesCosts
Elapsed time is 0.000474 seconds.
Elapsed time is 0.000551 seconds.
```

Z příkladu je vidět, že je GPU pomalejší nebo jsou rychlosti s CPU srovnatelné. Pokud bude počet prvků n větší například $n = 100\,000\,000$, pak jsou časy následující:

```
>> resourcesCosts
Elapsed time is 1.091723 seconds.
Elapsed time is 0.560803 seconds.
```

Z toho plyne, že pro menší počty n použití GPU nepomůže s rychlostí a dost možná rychlost výpočtu sníží, ale pro velké počty prvků GPU razantně sníží čas výpočtu, pokud se počítá přímo v paměti GPU. U těchto časů také musíme uvážit náročnost operace. Výpočet *sin* není tolik složitý proto jsou časy většinou shodné, ale nebýt tam režie přenosu pak se GPU rychlejší.

Řídká matice

Většina výsledných struktur jako matice tuhosti a hmotnosti jsou značně velké (v závislosti na počtu elementů). Pro další práci s nimi je zmenšíme. Na zmenšení využijeme řídké matice

(sparse matrix). O převod do tvaru řídké matice se postará příkaz `sparse`, tomuto příkazu v argumentu předáme matici, kterou chceme převést [18].

Pro práci s řídkými maticemi (velké matice, jejichž většina prvků jsou nuly), je dobré zvážit použití "řídkého maticového tvaru"v aplikaci MATLAB. Pro velmi velké matice vyžaduje řídká forma matice méně paměti, protože ukládá pouze nenulové prvky, a je rychlejší, protože eliminuje operace s nulovými prvky. Na řídkou matici lze aplikovat všechny vestavěné operace MATLABu. Operace s řídkou maticí se provádějí automaticky a vracejí výsledky ve tvaru řídké matice [16, 18].

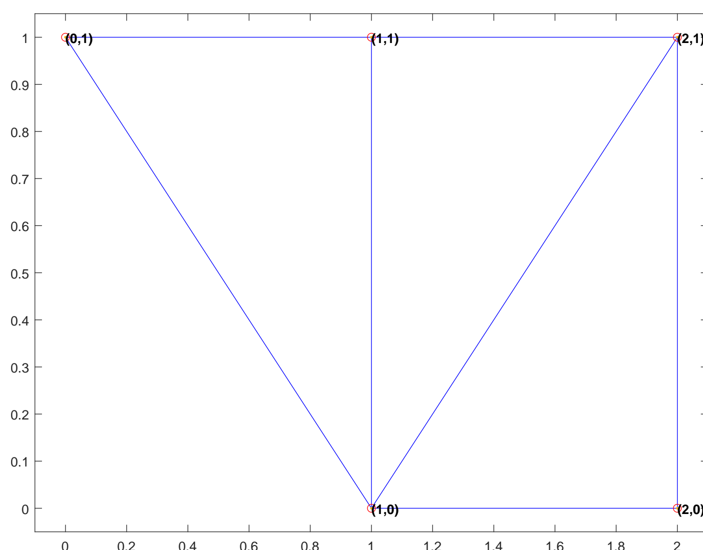
```
Name        Size               Bytes   Class       Attributes
K       14739x14739          7290748   gpuArray    sparse
Name        Size               Bytes   Class       Attributes
K       14739x14739       1737904968   gpuArray
```

V příkladu výše jsou uvedeny výpisy příkazu `whos` pro proměnou *K* představující matici tuhosti v elasticitě, která bude probírána později. V prvním řádku je uložena jako řídká matice, která zabírá přibližně 6.95 MB. V druhém řádku je stejná matice převedená příkazem `full` na plnou matici s mnohem větší paměťovou náročností a to přibližně 1657 MB. Pro matice se kterými budeme později pracovat, je využití struktur řídkých matic přímo podmíněné, bez nich by nám velice rychle došla paměť.

4. Struktura ukládání trojúhelníkových sítí

V této sekci definujeme použité struktury pro ukládání trojúhelníkových sítí, se kterou budeme v průběhu zbytku práce pracovat [19, 20]. Jednotlivé struktury lze poměrně jednoduše zobecnit i pro čtyřstěnné sítě ve třech dimenzích.

Pro uložení souřadnic trojúhelníků byl využit zjednodušený systém polí se snadným vyhledáváním a úsporou paměti. Podobný systém, ale s více informacemi, využívá i PDE Toolbox. Na jednoduché trojúhelníkové síti na Obrázku 4.1 si vysvětlíme princip tohoto systému a jak se s takovým systémem pracuje.



Obrázek 4.1: Jednoduchá trojúhelníková síť se třemi trojúhelníky.

4.1 Struktura pole *coordinates*

Pole *coordinates* má dvě dimenze (matice). V první dimenzi je uložena dvojice souřadnic [x,y]. Pomocí druhé dimenze pak přistupujeme k jednotlivým složkám dvojice. Tímto způsobem jsou uloženy všechny body sítě. Záznam na prvním indexu v Tabulce 4.1 máme bod [1,0], který můžeme vidět na Obrázku 4.1.

Index	X	Y
1	1	0
2	1	1
3	0	1
4	2	0
5	2	1

Tabulka 4.1: Vizualizace pole *coordinates* pro síť na Obrázku 4.1.

4.2 Struktura *elements*

Je také pole s dvěma dimenzemi (matice), ve kterém jsou uchovány informace o vrcholech jednotlivých trojúhelníků. Na každém indexu jsou uloženy vrcholy jednoho trojúhelníku. Souřadnice vrcholů jsou uloženy ve formátu indexů řádek z Tabulky 4.1. Například na prvním indexu v 4.2 máme hodnoty 1, 2, 3, které odkazují na indexy 1, 2, 3 v Tabulce 4.1. Tudíž získáme trojici bodů tvořící trojúhelník, který je spolu s ostatními dvěma trojúhelníky znázorněný na Obrázku 4.1.

Index	a	b	c
1	1	2	3
2	1	2	5
3	1	4	5

Tabulka 4.2: Vizualizace pole *elements* pro síť na Obrázku 4.1.

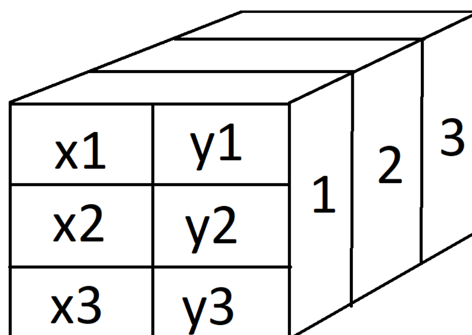
4.3 Struktura *coord*

Tuto strukturu budeme využívat hlavně při sestavování matic tuhosti a hmostnosti. Struktura *coord* je trojrozměrná matice a vznikne spojením struktur *coordinates* a *elements*. Nejdříve do dvourozměrného pole uložíme konkrétní souřadnice všech vrcholů trojúhelníku, které můžeme vidět v Tabulce 4.3 pro trojúhelník označený indexem 1 z Tabulky 4.2.

Vrchol	x	y
1	1	0
2	1	1
3	0	1

Tabulka 4.3: Vizualizace 2D pole, které se součástí *coord* pro síť na Obrázku 4.1.

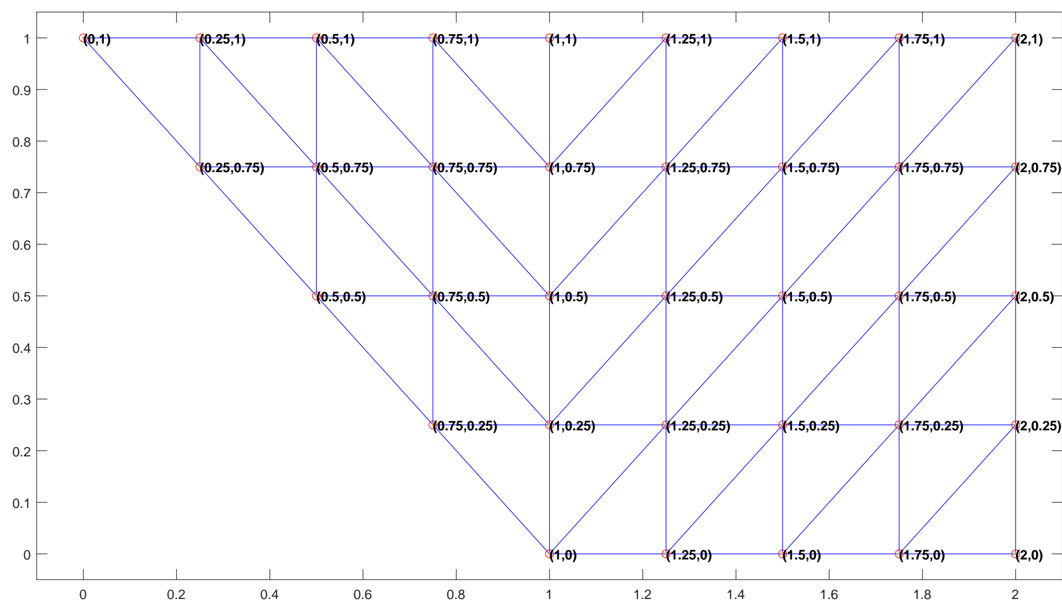
Takto vzniklé pole poté uložíme do pole *coord* na první index (tím nám vznikne 3D pole) a pokračujeme na další trojúhelník. Vzniklé 3D pole poté vidíme na Obrázku 4.2.



Obrázek 4.2: Vizualizace 3D pole *coord*.

4.4 Uniformní zjemnění sítě

Trojúhelníkové sítě je možno snadno dále zjemňovat. Na Obrázku 4.3 můžeme vidět dvojnásobné zjemnění sítě z Obrázku 4.1. Síť má stejné rozměry a plochu, ale tvoří ji více trojúhelníků a jejich bodů. V určitých případech může takové zjemnění zvýšit přesnost výpočtu, nebo usnadnit práci při testování kódu na rychlost podle počtu prvků.



Obrázek 4.3: Příklad zjemnění sítě.

5. Výpočet plochy rovinného obrazce

Abychom mohli přibližně vypočítat plochu rovinného obrazce, musíme si aproximovat pomocí hranice o počtu prvků n. Počtem prvků se určuje jemnost hranice pro následné výpočty. Existuje více metod na výpočet plochy, které se dají porovnat v rychlosti výpočtu.

5.1 Gaussova metoda (shoelace method)

Gaussova metoda je matematický algoritmus výpočtu plochy jednoduchého mnohoúhelníku. Vrcholy mnohoúhelníku jsou popsány kartézskými souřadnicemi v rovině. Hlavní myšlenkou metody je křížové násobení odpovídajících souřadnic pro nalezení oblasti obklopující mnohoúhelník a odečtení okolních lichoběžníků pro zjištění oblasti uvnitř.

$$2S = \sum_{i=1}^{n} (x_i(y_{i-1} - y_{i+1})) \tag{5.1}$$

V MATLABu lze zapsat například takto:

```
1  % example_5_gaussMethod.m
2  fx = @(x)2*(cos(x).*(1+sin(x))); fy=@(y)sin(y);  % hranice
3  t = (linspace(0,n-1,n)*(2*pi/n));
4  x = fx(t); y = fy(t);                          % body hranice
5  S = sum(x.*(([y(2:end), y(1)]) - ([y(end), y(1:end-1)])))/2
```

Pole x a y obsahují souřadnice na osách X a Y získané z dané funkce.

5.2 Heronův vzorec

V další části budeme vytvářet trojúhelníkovou síť. Tak trochu předběhneme a využijeme ji na další tři možné výpočty plochy. Jeden typ je s použitím Heronova vzorce spočítáme obsahy trojúhelníků.

$$S = \sqrt{s(s-a)(s-b)(s-c)} \tag{5.2}$$

kde

$$s = \frac{a+b+c}{2}$$

je poloviční obvod trojúhelníku. Tyto plochy poté sečteme a dostaneme plochu obrazce.

Níže naleznete jednoduchou implementaci v MATLAB. Napřed se získají souřadnice vrcholů každého trojúhelníku zvlášť a poté se spočítá

```
1  % example_6_heron.m
2  plocha = 0;
3  for i = 1:size(elements,1)
4      n1=elements(i,1);
5      n2=elements(i,2);
6      n3=elements(i,3);
7      ver1 = coordinates(n1,:);
8      ver2 = coordinates(n2,:);
9      ver3 = coordinates(n3,:);
10     vecA = ver2-ver1;  vecB = ver3-ver1; vecC = ver2-ver3;
11     a = norm(vecA); b = norm(vecB); c = norm(vecC);
12     s = (a+b+c)/2;
13     %Heronuv vzorec
```

```
14      plocha = plocha + sqrt(s.*(s-a).*(s-b).*(s-c));
15  end
```

5.3 Pomocí determinantu 2x2 nebo 3x3 matice

Matice 2x2

Z bodů vzniklé sítě vytvoříme vektory. Spočítáme determinant matice 2x2, která vznikne složením dvou libovolných vektorů z bodů trojúhelníku. Obdélník který je tvořen dvěma vektory, následně vypočítáme determinant vektorů a tím získáme plochu daného obdélníku. Vzniklou plochu vydělíme dvěma, abychom dostali potřebnou poloviční plochu, která odpovídá ploše daného trojúhelníku.

$$S = \det \begin{pmatrix} a_1 & a_2 \\ b_1 & b_2 \end{pmatrix} /2$$

Tento postup opakujeme pro všechny trojúhelníky a následně tyto plochy sečteme. Tímto výpočtem získáme plochu celého obrazce.

```
1  % example_7_det2x2.m
2  plocha = 0;
3  for i = 1:size(elements,1)
4      n1=elements(i,1); n2=elements(i,2); n3=elements(i,3);
5      ver1 = coordinates(n1,:);
6      ver2 = coordinates(n2,:);
7      ver3 = coordinates(n3,:);
8      a = ver2-ver1; b = ver3-ver1;
9      plocha = plocha + (det([a;b])/2);
10 end
```

Matice 3x3

Jelikož známe všechny tři vrcholy trojúhelníku, můžeme vypočítat obsah metodou, která využívá 3x3 matici. Matici sestavíme následovně.

$$S = \det \begin{pmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{pmatrix} /2$$

V prvním sloupci jsou souřadnice vrcholů trojúhelníku na ose x, v druhém sloupci jsou souřadnice osy y a poslední sloupec je doplněný jedničkami. Po získání matice spočítáme její determinant, který vydělíme dvěma, tím dostaneme plochu jednoho trojúhelníku. Po sečtení ploch všech trojúhelníků získáme plochu celého obrazce.

```
1  % example_8_det3x3.m
2  plocha = 0;
3  for i = 1:size(elements,1)
4      plocha = plocha + det([coordinates(elements(i,:),:),[1;1;1;]])/2;
5  end
```

6. Vytvoření trojúhelníkové/tetrahedrální sítě

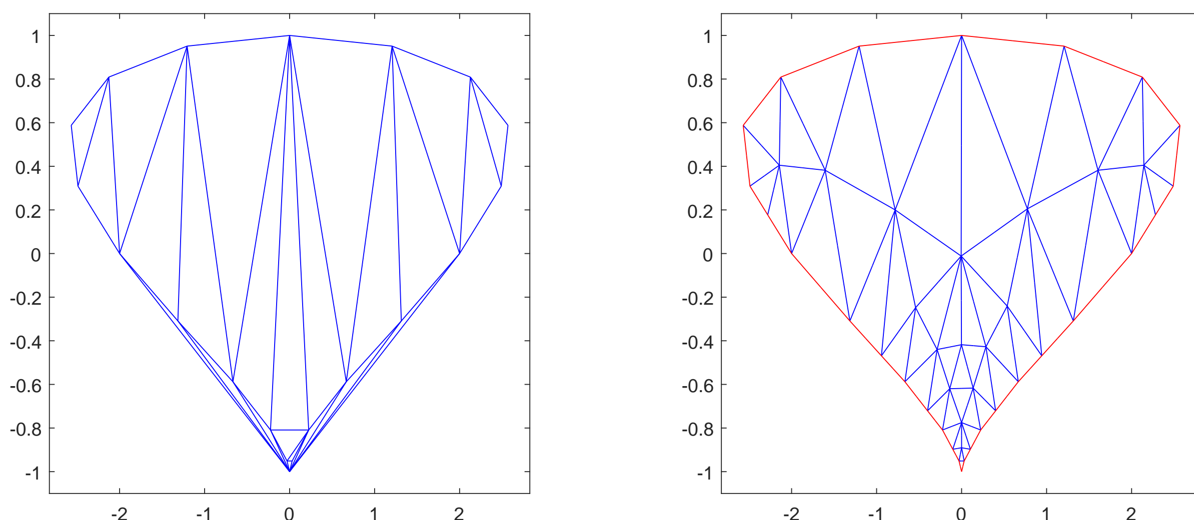
Po vytvoření hranice obrazce je třeba vytvořit trojúhelníkovou síť se kterou budeme nadále pracovat. Vytvoření sítě je poměrně složitý problém. S tím nám pomůže MATLAB, který má vestavěné funkce pro tvorbu takovýchto sítí.

6.1 Delaunayho triangulace ve 2D

Jedna z možností, jak vytvořit trojúhelníkovou síť je pomocí Delaunayho triangulace. Funkce `delaunayTriangulation`po zadání hranice vytvoří trojúhelníkovou síť daného obrazce, jejíž příklad je ukázán na Obrázku 6.1 (vlevo). K vygenerování sítě slouží následující kód:

```
1 % example_9_meshDelaunay.m
2 n=20; % Pocet uzlu
3 % Funkce, ktera definuje hranici oblasti
4 fx = @(x)2*(cos(x).*(1 + sin(x))); fy = @(y)sin(y);
5 T = (linspace(0, n-1, n)*(2*pi/n));
6 x = fx(T); y = fy(T);
7 DT = delaunayTriangulation(reshape(x,n,1),reshape(y,n,1));
8 triplot(DT);
```

Ačkoliv taková Delaunayho triangulace stačí na výpočet plochy, pro využití sestavení matice tuhosti a následného zobrazení je nevyužitelná. Trojúhelníky v ní obsahují totiž velmi malé úhly, které způsobují špatné numerické vlastnosti výsledné matice tuhosti.



Obrázek 6.1: Delaunayho triangulace (vlevo) a PDE triangulace (vpravo).

6.2 Triangulace pomocí PDE (Partial Differential Equation) Toolbox ve 2D

Triangulace pomocí PDE je pro sestavení matice tuhosti a následné zobrazení vhodnější. PDE totiž doplní body na hranici o body uvnitř obrazce. Na Obrázku 6.1 (vpravo) je příklad takové triangulace.

Vytvoření sítě pomocí z triangulace

Pro jednoduché vytvoření trojúhelníkové sítě, se kterou budeme dále pracovat, bude využita následující funkce:

```
% meshGenerator.m
function [coordinates, elements, dirichlets] = meshGenerator(x,y)
    model = createpde; n = length(x);
    geometryDescription = [2, n, x, y]'; % Popis obrazce body hranice
    sf = 'object'; ns = char('object'); ns = ns';
    geometry = decsg(geometryDescription, sf, ns);
    geometryFromEdges(model,geometry);
    P = [x; y]'; [minDist,maxDist]=minAndMaxDist(P);
    mesh = generateMesh(model,'Hmax',maxDist, 'Hmin',minDist);
    elements = mesh.Elements(1:3,:)'; % ulozeni elementu z mesh
    coordinates = mesh.Nodes';
    dirichlets = convhull(coordinates);
    dirichlets = intersect(dirichlets,elements);
end
```

Funkce má jako vstupní parametry souřadnice na osách x a y. Výstupní parametry pak jsou coordinates (body na hranici), elements (jednotlivé vrcholy trojúhelníků) a dirichlets (vrcholy trojúhelníků, které leží na hranici) pro pozdější upevnění membrány k hranici obrazce.

Pro vytvoření sítě je třeba inicializovat PDE model a vytvořil popis geometrického obrazce. Popis geometrického obrazce (proměnná geometryDescription) je pole, kde se na prvním řádku nachází identifikátor, o jaký objekt se jedná, pro nás je to číslo 2, které znamená, že popisujeme mnohoúhelník. Druhá řádka obsahuje číslo n počet souřadnic popisující mnohoúhelník. Od třetí řádky do řádky *n+3* po sobě jdoucí souřadnice na ose x a od řádky *n+4* do *2n+4* jsou po sobě jdoucí souřadnice na ose y. Jelikož funkce pro vytvoření matice dekomponované geometrie `decsg`, dokáže pracovat s více množinami je třeba do parametru *sf* (set formula) zadat výčet všech množin a vztahy mezi jednotlivými množinami. V tomto případě se jedná o název *object* pro pojmenování obecného vstupního objektu. Pod parametrem *ns* (name-space matrix) se skrývá matice čísel s dvojnásobnou přesností. Počet sloupců odpovídá počtu tvarů použitých ke konstrukci geometrie. Po vygenerování matice dekomponované geometrie bude možno s její pomocí vytvořit 2D geometrii. Při předání matice dekomponované geometrie v parametru funkce `geometryFromEdges` společně s na začátku inicializovaným modelem do kterého pak funkce uloží výsledky bude výsledkem model zadaného objektu potřebný pro vytvoření sítě. Pro vytvoření sítě pomocí `generateMesh` je třeba dodat minimální a maximální délku hrany jednoho trojúhelníku (Hmin a Hmax). Pokud se Hmin a Hmax nezadá pak je možné, že funkce vyhodí výjimku pro určité hranice. Z toho důvodu je jistější si Hmin a Hmax zadat, pokud ale však není známo jaké jsou minimální či maximální délky hran trojúhelníků lze využít funkci `minAndMaxDist`, která nalezne minimální a maximální délku hran trojúhelníku.

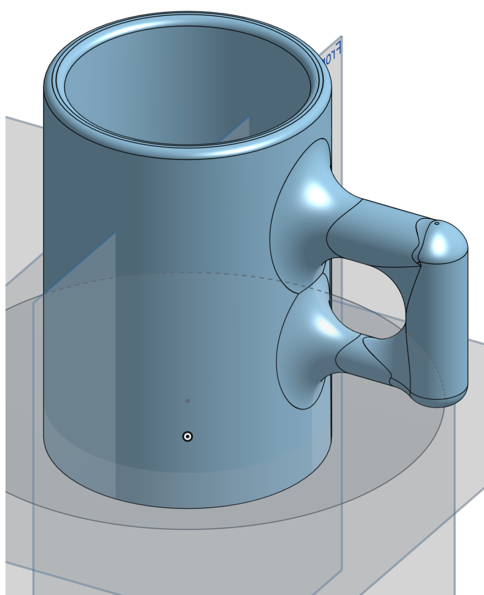
```
% minAndMaxDist.m
function [minDist, maxDist] = minAndMaxDist(points)
    minDist = intmax;
    maxDist = 0;
    for i = 1 : length(points)
       A = points(i, :);
       for j = i+1 : length(points)
          B = points(j, :);
          dist = norm(A-B);
          if minDist > dist
              minDist = dist;
          end
```

```
13            if maxDist < dist
14                maxDist = dist;
15            end
16        end
17    end
18 end
```

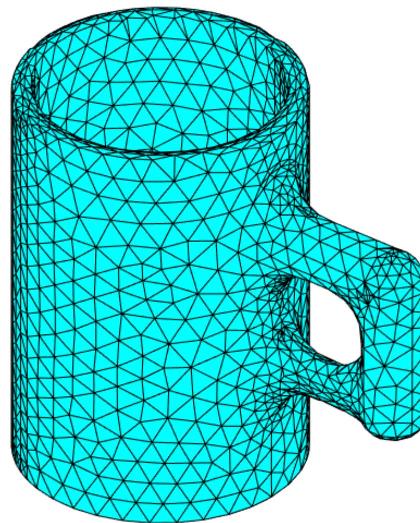
Vzdálenost mezi body dvěma body v rovině lze spočítat pomocí MATLAB funkce norm, která vrací Eukleidovskou vzdálenost vektoru.

6.3 Nahrání sítě z 3D modelu

Další možností jak získat trojúhelníkovou síť je nahrání z již existujícího 3D modelu. V tomto textu budeme využívat modely ve formátu STL. Takové modely se dají vytvořit v CAD programech jako je například onshape. Jedním z takto vytvořených modelů, je například tento hrnek, se kterým budeme nadále pracovat.



(a) Modelu hrnku v náhledu CAD.



(b) Tetrahedrální síť hrnku z STL souboru v MATLAB.

Obrázek 6.2: Model hrnku zobrazen ve dvou prostředích.

Formát STL

Často využíván v 3D tisku a počítačovém projektování (CAD). STL je zkratkou pro stereolitografii, ale má i další vysvětlení, které je pro naše použití vystihují lépe a to je "Standard Triangle Language".

Každý soubor se skládá ze série propojených trojúhelníků, jež popisují geometrii povrchu 3D modelu. Čím více trojúhelníků tím složitější model a větší rozlišení [21].

Získání sítě z STL do MATLABu

Pro získání sítě ve tvaru, se kterým budeme nadále pracovat (*elements*, *coordinates*), byla vytvořena funkce v MATLABu. Funkce má jeden vstupní parametr jméno souboru. Pro načtení

dat do PDE objektu ze souboru určeného vstupním parametrem, využijeme PDE toolbox. Následně pomocí PDE objektu vygenerujeme síť, ze které získáme potřebné údaje.

```
% meshFromSTL.m
%% Vraci pole coordinates a elements ziskane z STL modelu
function [coordinates, elements] = getDataFromSTL_simple(name)
    model = createpde;
    importGeometry(model, name);
    mesh = generateMesh(model);
    pdemesh(mesh)
    elements = mesh.Elements(1:4,:)';
    coordinates = mesh.Nodes';
end
```

Výstupem funkce jsou tedy pole *elements*, *coordinates* a vykreslení trojúhelníkové 3D sítě pomocí PDE z Obrázku 6.2 *b*.

7. Vektorizace

Vektorizace je sloučením dvou typů matematických aritmetických operací. Maticové operace definované zákony lineární algebry a operace s poli prováděné prvek po prvku. Práci s polem o prvcích rozšíříme na práci s polem o matice, kde prvky pole jsou matice a operace jsou definovány pravidly lineární algebry [20].

Vzhledem k tomu, že MATLAB má vektorovou/maticovou reprezentaci svých dat, vektorizace může pomoci zrychlit běh kódů MATLABu. Klíčem k vektorizaci je minimalizovat použití smyček (např. `for`) [16].

7.1 Implementace uložení trojúhelníků ve 2D

Trojúhelníky, které získáme z triangulace ve formátu *elements*, *coordinates* vektorizujeme následovně. Napřed je třeba získat indexy jednotlivých vrcholů z *elements*. Na druhé řádce skriptu získáme indexy každé vrcholu zvlášť. Každý vrchol pak bude mít svoje pole, kde budou tyto indexy.

Na řádkách čtyři až šest z indexů obdržených v předchozím kroku, získáme stejným způsobem souřadnice jednotlivých vrcholů.

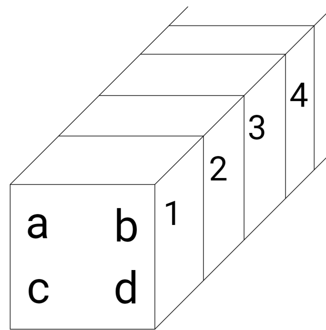
Dále ze souřadnic získáme dva vektory u a v. A před-připravíme si 3D matici 2x2xn, kde n je počet trojúhelníků.

Jednotlivé složky vzniklých vektorů pak uložíme do nově vzniklého pole *uv* po složkách.

```
1  % example_10_vectorization.m
2  n1=elements(:,1); n2=elements(:,2); n3=elements(:,3);
3
4  ver1 = coordinates(n1,:);
5  ver2 = coordinates(n2,:);
6  ver3 = coordinates(n3,:);
7  u = ver2-ver1; v = ver3-ver1;
8
9  ne = size(elements,1); %pocet elementu
10 uv = zeros(2,2,ne);
11
12 uv(1,1,:) = u(:,1); %a
13 uv(1,2,:) = u(:,2); %b
14 uv(2,1,:) = v(:,1); %c
15 uv(2,2,:) = v(:,2); %d
16
17 a = uv(1,1,:); b = uv(1,2,:);
18 c = uv(2,1,:); d = uv(2,2,:);
19 matrix3D = [a, b; c, d];
```

Obrázek 7.1: Vektorizace 2D trojúhelníků

Na Obrázku 7.2 je znázorněna grafická vizualizace vzniklého pole *uv*. S takovou 3D maticí můžeme nadále pracovat rychleji, jelikož MATLAB pracuje efektivněji s maticemi než s cykly.



Obrázek 7.2: Vizualizace vzniklé 3D matice.

Využití vektorizace pro výpočet plochy

S takto vzniklou 3D maticí můžeme snadno a rychle dále pracovat. V této sekci si ukážeme jak vektorizovanou matici využít pro výpočet obsahu obrazce pomocí metod uvedené v sekci 5.

Vektorizace výpočtu obsahu pomocí 2x2 determinantu

Pro výpočet obsahu pomocí 2x2 determinantu krásně využijeme 3D matici vzniklou v předchozí sekci. Jelikož úvodní kroky pro vektorizaci výpočtu obsahu pomocí determinantu 2x2 jsou identické těm, které jsme dělali ve skriptu `vectorization.m` není třeba znovu ukazovat stejný kód. Zavoláme skript `vectorization.m`, aby nám sestavil 3D matici a následně pouze využijeme vzniklé dlouhé vektory *a, b, c, d* k výpočtu determinantu 2x2 matice. Determinant se spočte pro každý trojúhelník zvláště, poté pouze vzniklé determinanty sečteme a získáme obsah obrazce.

```
1 % example_11_vectorize_det2x2.m
2 vectorization.m
3 plocha = sum(((a.*d)-(c.*b))/2);
4 fprintf("Plocha pomoci vektorizace a determinantu 2x2: %f \n", plocha)
```

Takto jednoduše vypočteme obsah trojúhelníkové sítě za použití determinantu 2x2 bez smyčky `for`.

Vektorizace výpočtu obsahu pomocí 3x3 determinantu

Pro výpočet obsahu pomocí determinantu 3x3 nejprve vytvoříme 3x3xn matici s jedničkami ve třetím sloupci jak bylo již vysvětleno. Následně pro každou matici vypočteme determinant pomocí Sarrusova pravidla. Nakonec sečteme determinanty všech matic výsledek vydělíme dvěma a dostaneme plochu obrazce.

```
1 % example_12_vectorize_det3x3.m
2 n1=elements(:,1); n2=elements(:,2); n3=elements(:,3);
3 ver1 = coordinates(n1,:);
4 ver2 = coordinates(n2,:);
5 ver3 = coordinates(n3,:);
6
7 xy = ones(3,3,ne);
8 xy(1,:,:) = [ver1(:,1),ver1(:,2), ones(ne,1)]';
9 xy(2,:,:) = [ver2(:,1),ver2(:,2), ones(ne,1)]';
10 xy(3,:,:) = [ver3(:,1),ver3(:,2), ones(ne,1)]';
11
```

```
12 % Vypocet determinantu pomoci Sarrusova pravidla
13 plocha = sum(xy(1,1,:).*xy(2,2,:).*xy(3,3,:)...
14             +xy(1,2,:).*xy(2,3,:).*xy(3,1,:)...
15             +xy(1,3,:).*xy(2,1,:).*xy(3,2,:)...
16             -xy(3,1,:).*xy(2,2,:).*xy(1,3,:)...
17             -xy(3,2,:).*xy(2,3,:).*xy(1,1,:)...
18             -xy(3,3,:).*xy(2,1,:).*xy(1,2,:))/2;
19 fprintf("Plocha pomoci vektorizace a determinantu 3x3: %f \n", plocha)
```

Takto jednoduše vypočteme obsah trojúhelníkové sítě za použití determinantu 3x3 bez smyčky `for`.

Vektorizace Heronova vzorce

Při ukázce vektorizace a sestavení 3D matice jsme pracovali pouze s vektory *a* a *b*, ale pro Heronův vzorec budeme potřebovat ještě vektor *c*. Naše 3D matice je tedy pro výpočet Heronova vzorce nedostačující, ale to neznamená, že se nedá vektorizovat.

Jak bylo popsáno výše, nejdříve získáme souřadnice vrcholů *verX*, kde X je číselné označení vrcholu trojúhelníku. Poté na řádku osm vypočítáme vektory *a, b, c* pro všechny trojúhelníky najednou. Na řádcích jedenáct až třináct spočítáme velikosti těchto vektorů. Dále dosadíme jednotlivé pole do Heronova vzorce a spočítáme obsahy všech trojúhelníků zvlášť. Posledním krokem je součet obsahů všech trojúhelníků.

```
1  % example_13_vectorize_heron.m
2  n1=elements(:,1); n2=elements(:,2); n3=elements(:,3);
3  ver1 = coordinates(n1,:);
4  ver2 = coordinates(n2,:);
5  ver3 = coordinates(n3,:);
6
7  % ziskani vektoru a, b, c
8  a = ver2-ver1; b = ver3-ver1; c = ver2-ver3;
9
10 % vypocet velikosti vektoru a, b, c
11 a = sqrt(a(:,1).^2+a(:,2).^2);
12 b = sqrt(b(:,1).^2+b(:,2).^2);
13 c = sqrt(c(:,1).^2+c(:,2).^2);
14
15 s = (a+b+c)/2;
16 % Heronuv vzorec
17 plocha = sum(sqrt(s.*(s-a).*(s-b).*(s-c)));
18 fprintf("Plocha pomoci vektorizace a Heronova v: %f\n", plocha)
```

Takto jednoduše vypočteme obsah trojúhelníkové sítě za použití Heronova vzorce bez smyčky `for`.

8. Metoda konečných prvků a její využití

V této kapitole nahlédneme do problematiky metody konečných prvků a její vektorizace. Definujeme si základní pojmy spjaté s metodou a její implementací v MATLABu.

Základem této kapitoly je článek [20]. Článek se zabývá efektivním, flexibilním a rychlým sestavením matic tuhosti a hmotnosti v MATLABu. Hlavní myšlenkou je odstranění cyklů přes elementy, které zabírají nejvíce času. Cykly jsou poté nahrazeny vektorizací. Na tento článek navážeme a pokusíme se ho zrychlit pomocí GPU.

8.1 Sestavení matice tuhosti

Lokální matice tuhosti je určena souřadnicemi vrcholů matice odpovídajícího trojúhelníku. Sestavení globální matice tuhosti se budeme věnovat v této sérii skriptů. Matici rovněž budeme sestavovat za využití GPU a na konci sekce srovnáme jednotlivé časy sestavení CPU a GPU.

8.1.1 Společná část sestavení matice tuhosti

Přestože budeme sestavovat matici tuhosti pro dvě využití (elasticita a Poissonova rovnice), obě mají společný základ. Prakticky je třeba získat souřadnice trojúhelníků do struktury, se kterou se bude snáze pracovat a sestavit matici gradientů jednotlivých bázových funkcí. Tímto se budeme zabývat v následujících podkapitolách.

Získání souřadnic trojúhelníků

V této části skriptu získáme z již známé struktury souřadnice trojúhelníků a uložíme je do nové struktury *coords*. Struktura *coords* má uloženy dimenzionální složky souřadnic (parciální souřadnice dané dimenze) pro každou basickou funkci.

```
1  % stiffness_matrixP1_3D_gpu.m
2  elements = gpuArray(elements);
3  coordinates = gpuArray(coordinates);
4  NE=size(elements,1);                  %pocet prvku
5  DIM=size(coordinates,2);              %dimenze
6  NLB=4;                                %pocet lokalnich funkci
7  coord=zeros(DIM,NLB,NE, 'gpuArray'); %definice GPU pole
8  for d=1:DIM
9      for i=1:NLB
10         coord(d,i,:)=coordinates(elements(:,i),d); %souradnice
11     end
12 end
13 IP=gpuArray([1/4 1/4 1/4]');
14 [dphi,jac] = phider_gpu(coord,IP,'P1');
```

Funkce phider

Funkce `phider` počítá gradienty všech bázových funkcí definovaných na elementech. Funguje v dimenzích 2 a 3 a my ji zrychlíme pomocí GPU. Tato funkce má jako vstup *coords*, které jsou už z předchozí sekce uloženy na GPU, *IP* neboli (Gaussovy) integrační body a typ elementu (u nás tzv. po částech lineární funkce, neboli P1 bázové funkce). Funkce poté vrátí jako výstup

gradienty bázových funkcí vzhledem k lokálním souřadnicím, Jacobiho matici zobrazení z referenčního do aktuálního elementu a její determinant.

```
% phider_gpu.m
dshape = gpuArray(shapeder(point,etype));
nod = size(coord,1); % dimenze elementu
nop = size(point,2); % pocet integracnich bodu
nos = size(coord,2); % pocet tvarovych  funkci
noe = size(coord,3); % pocet elementu
dphi  = zeros(nod,nos,nop,noe, 'gpuArray');
jac   = zeros(nod,nod,nop,noe, 'gpuArray');
detj = zeros(1,nop,noe, 'gpuArray');
```

V první řadě je třeba získat *dshape* (derivaci referenčního elementu) pomocí funkce `shapeder`, tato funkce byla dodaná v balíčku *library_vectorization_faster*. Vrátí matici kde pro náš případ *P1* bázových funkcí v prvním sloupci budou 1 a v druhém -1, počet řádků je pak počet integračních bodů. Tuto matici si převedeme na GPU, abychom v dalších částí pracovali v jednotném prostředí (GPU). Nadále si předdefinujeme čtyř-dimenzionální pole (matici) pomocí MATLAB funkce `zeros`. Pokud jako poslední argument funkce `zeros` předáme řetězec `'gpuArray'`,pak se pole vytvoří přímo na GPU. Stejnou logiku využijeme i při předdefinování polí *jac* pro Jacobiho matici a *detj* její determinant.

```
% phider_gpu.m
% Linearni tvarove funkce - P1.
dshape = [1 -1];
dshape = reshape(dshape,2,1);
dshape = dshape*ones(1,nop);
dshape = reshape(dshape,1,2,nop);

poi = 1; % pro nas pripad
tjac                = smamt_gpu(dshape(:,:,poi),coord);
[tjacinv,tjacdet] = aminv_gpu(tjac);
dphi(:,:,poi,:)     = amsm_gpu(tjacinv,dshape(:,:,poi));
jac(:,:,poi,:)  = tjac;
detj(1,poi,:)  = abs(tjacdet);
```

Tato část cyklí přes počet integračních bodů a postupně získáváme částečné řešení, které se ukládá do předdefinovaných polí na správnou pozici pole aktuálního běhu cyklu. Po ukončení cyklu dostaneme naplněná pole *dphi*, *jac*, *detj*, která funkce vrátí jako výstup.

Formátování výstupů phider

Jelikož je funkce `phider` univerzální, je třeba formátovat výstup, který bude připraven pro následující využití pro specifické části. Využitím funkce `squeeze` odstraníme přebytečnou dimenzi. Pole tedy bude za běhu potřebovat méně paměti a se bude snáze iterovat.

```
volumes=abs(squeeze(jac))/factorial(DIM); % vypocet objemu
dphi = squeeze(dphi); % odstraneni prebytecne dimenze
```

8.1.2 Část specifická pro Poissonovu rovnici

Funkce `astam` a `amtam` provádějí operace s maticemi. GPU verze `amtam_gpu` využívá předdefinování pole pomocí funkce `zeros` přímo na GPU, jelikož jsou pak všechny pole na GPU,

operace s nimi se nadále dějí na GPU. Do proměnné Z dostaneme všechny lokální 3D matice tuhosti.

```
1 % stiffness_matrixP1_3D_gpu.m
2 Z=astam(volumes',amtam_gpu(dphi,dphi)); %lokalni matice tuhosti
3 Y=reshape(repmat(elements,1,NLB)',NLB,NLB,NE); %sd
4 X=permute(Y,gpuArray([2 1 3])); %sd
5 K=sparse(X(:),Y(:),Z(:)); % vysledna matice tuhosti pro objekt
```

Výstupem jsou pak proměnné matice K a vektor *volumes* z předchozí sekce. Kde K je řídká matice tuhosti pro zadané parametry. Taková matice figuruje v numerických řešeních Poissonovy rovnice.

8.1.3 Část specifická pro elasticitu

Napřed předdefinujeme pole R, které reprezentuje operátor tenzoru malých deformací na grafické kartě a poté do jej na konkrétních indexech naplníme gradienty. Jelikož je sestavení elasticity náročnější na paměť než sestavení matice Poissonovy rovnice, uvolníme proměnné (pole) pomocí příkazu `clear` a jméno proměnné. Uvolnění proměnné sice zabere nějaký čas, ale v tomto případě je to akceptovatelné, neboť budeme moci provádět sestavení pro sítě s více elementy.

```
1 % stiffness_matrixP1_3D_elasticity_gpu.m
2 R=zeros(6,12,NE, 'gpuArray');
3 R([1,4,5],1:3:10,:)=dphi;
4 R([4,2,6],2:3:11,:)=dphi;
5 R([5,6,3],3:3:12,:)=dphi;
6 clear dphi
7 clear coord
8 clear jac
```

Tímto obdržíme najednou bázové funkce pro tenzory malých deformací pro všechny elementy v síti.

Další řádky vygenerují materiálové matice v závislosti na tzv. Lammého parametrech λ, μ v případě homogenního materiálů.

```
1 % stiffness_matrixP1_3D_elasticity_gpu.m
2 C=mu*gpuArray(diag([2 2 2 1 1 1]))+lambda...
3     *gpuArray(kron([1 0; 0 0],ones(3)));
4 Elem=3*elements(:,gpuArray(kron(1:4,[1 1 1])))...
5     -gpuArray(kron(ones(NE,1),gpuArray(...
6     kron([1 1 1 1],[2 1 0]))));
7 clear elements
```

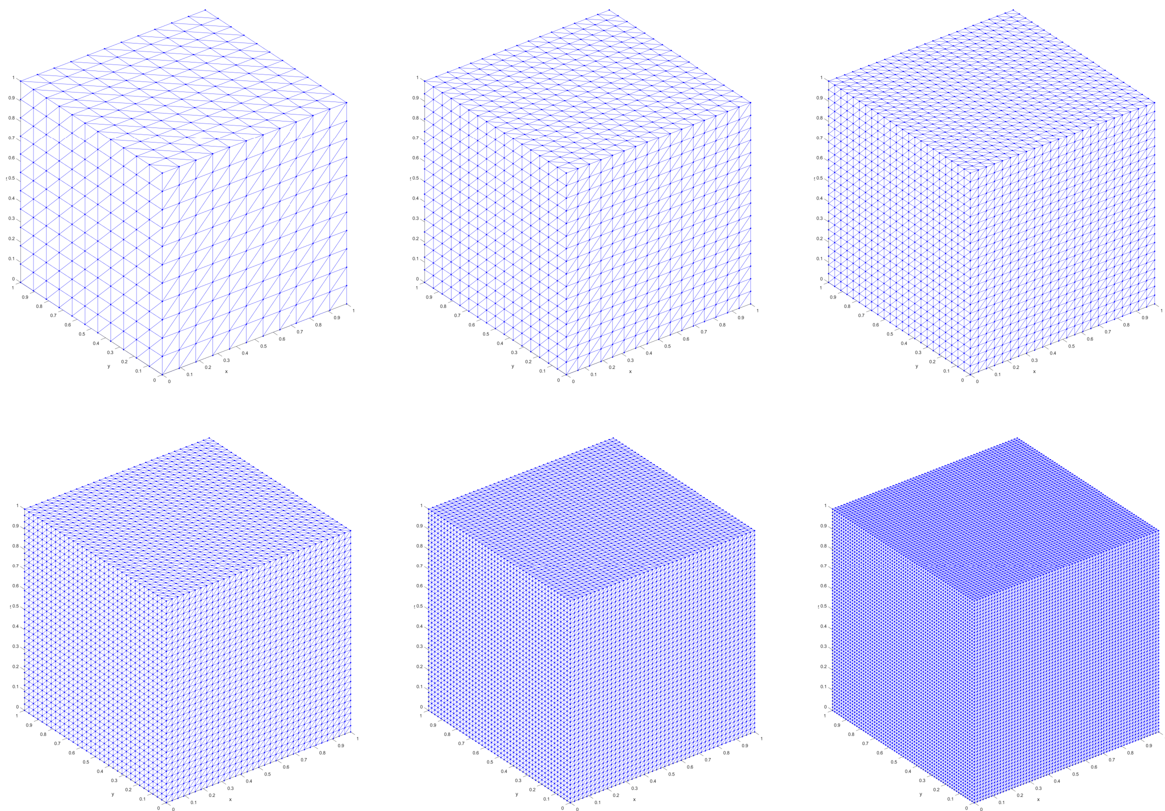
Na závěr vhodnou kombinací výše uvedených objektů sestavím výslednou matici tuhosti pro úlohu elasticity.

```
% stiffness_matrixP1_3D_elasticity_gpu.m
Z=astam(areas',amtam_gpu(R,smamt_gpu(C,...
permute(R,gpuArray([2 1 3])))));
Y=reshape(repmat(Elements,1,12)',12,12,NE);
X=permute(Y,gpuArray([2 1 3]));
K=sparse(X(:),Y(:),Z(:));  % vysledna matice tuhosti pro objekt
```

Výstupem je matice tuhosti K odpovídající rovnici elasticity. Tato matice je v řídkém tvaru a využívá se pro výpočty v lineární elasticitě.

8.1.4 Zrychlení sestavení matice tuhosti

Pro otestování je třeba mít k dispozici síť s velkým počet elementů, aby byl procesor co nejvíce zatížen a projevily se výhody grafických karet. Toto bohužel model hrnku neumožňuje, protože onshape vytvoří model s omezeným počtem elementů (v aktuální verzi programů je možné sestavit pouze síť s 8541 elementy). My využijeme model kostky z Obrázku 8.1, který je snáze škálovatelný na počet elementů a umožňuje sestavit prakticky libovolně jemnou síť (omezení je dáno jen pamětí počítače).



Obrázek 8.1: Modely kostky různé hrubosti: zleva v prvním řádku level 3.5, 4, 4.5 a druhém level 5, 5.5, 6. Nejjemnější síť obsahuje 1572864 tetrahedrů.

Testovat budeme na šesti různě hrubých kostkách z Obrázku 8.1. Parametry modelů kostek jsou popsány v Tabulce 8.1. Na tuto tabulku budeme v následujících sekcí odkazovat pomocí sloupce *Level*, abychom věděli pro jakou kostku s jakými parametry jsme dostali výsledek.

Level	Počet *elements*	Počet *coordinates*	Velikost matice tuhosti
3.5	7986	1728	1728 x 1728
4	24576	4913	4913 x 4913
4.5	73002	13824	13824 x 13824
5	196608	35937	35937 x 13824
5.5	546750	97336	97336 x 97336
6	1572864	274625	274625 x 274625

Tabulka 8.1: Parametry testovacích modelů.

Matice tuhosti pro Poissonovu rovnici

V Tabulce 8.2 jsou srovnány časy na jednotlivých zařízení. Z tabulky je patrné, že se účinnost grafické karty se zvyšuje s počtem počtu elementů. Zrychlení sestavení matice tuhosti pro Poissonovu rovnici až dvaceti násobná v závislosti na počtu prvků a využitém zařízení.

Level	Velikost matice tuhosti	GPU GTX1650 Max-Q	GPU A30	CPU Intel i7-8565U	CPU AMD EPYC 7313
4	4913	0.105	0.012	0.042	0.028
4.5	13824	0.259	0.016	0.182	0.074
5	35937	0.153	0.039	0.466	0.196
5.5	97336	0.428	0.105	1.257	0.558
6	274625	1.463	0.242	4.838	2.086

Tabulka 8.2: Srovnání časů sestavení matice tuhosti pro Poissonovu rovnici v sekundách.

Matice tuhosti pro úlohu elasticity

Z Tabulky 8.3 je patrné, že sestavení matice tuhosti pro elasticitu je časově složitější než pro Poissonovu rovnici. Zároveň vyplývá, že je třeba mnohem více paměti pro sestavení. Na grafické kartě GTX1650 Max-Q program nedoběhl již pro 546750 elementů (Level=5.5) z důvodu nedostatku paměti. Jak bylo srovnáno v Tabulce 2.2, grafická karta GTX1650 Max-Q má k dispozici pouze 4GB paměti, která byla nedostačující pro dokončení výpočtu. Z Tabulky 8.3 také plyne, že čím větší je spotřeba paměti pro danou úlohu tím více se projeví síla GPU.

Level	Velikost matice tuhosti	GPU GTX1650 Max-Q	GPU A30	CPU Intel i7-8565U	CPU AMD EPYC 7313
4	14739	0.323	0.207	0.399	0.201
4.5	41472	0.456	0.067	1.645	0.495
5	107811	1.144	0.236	4.524	1.803
5.5	292008	out of memory	0.412	15.054	5.592
6	823875	out of memory	1.095	95.216	17.530

Tabulka 8.3: Srovnání časů sestavení matice tuhosti pro elasticitu v sekundách.

8.2 Sestavení matice hmotnosti

Pro sestavení matice hmotnosti budeme potřebovat struktury *elements* a *volumes* (získané při sestavování matice hmotnosti), toto budou vstupní parametry naší funkce. Výstupem pak bude matice hmotnosti M. Vstupní parametry si přesuneme na GPU. Následně pomocí funkce `kron` a na polí alokovaných na paměti GPU pomocí funkce `ones`, získáme z *elements* X a Y skaláry.

```
1 % mass_matrixP1_3D_gpu.m
2 elements = gpuArray(elements);
3 volumes = gpuArray(volumes);
4 Xscalar=kron(ones(1,4,'gpuArray'),elements);
5 Yscalar=kron(elements,ones(1,4,'gpuArray'));
6 Z=kron(volumes,gpuArray(reshape((ones(4)+eye(4))/20,1,16)));
7 M=sparse(Xscalar,Yscalar,Z);
```

Následně z *volumes* za použití funkce `kron` získáme prvky matice hmotnosti, které pomocí funkce `sparse` přetvoříme na klasickou dvou-dimenzionální matici hmotnosti.

8.3 Výpočet elasticity

Než začneme s výpočtem je třeba si definovat vstupní parametry, jako jsou tuhost materiálu a síla *force*, která bude na objekt působit. Poté si rozdělíme objekt na pravou a levou stranu. Podle právě vzniklého rozdělení objektu na strany do pole f uložíme kladnou hodnotu síly pro levou stranu a zápornou pro pravou.

```
1 % solve_elasticity_3D_gpu.m
2 material.E = 1e9; % material
3 material.nu = 0.25;
4 nbfn = 3;
5 force = 10e6; % sila
6 % mesh.nn dostaneme pri sestaveni site a je to pocet uzlu
7 f = zeros(nbfn*mesh.nn,1,'gpuArray'); % definice gpu pole
8 nodes_L = find(coordinates(:,1)<0); % elementy na leve strane
9 nodes_R = find(coordinates(:,1)>0); % elementy na prace strane
10 f(3*nodes_L-2) = +force; % sila pusobici zleva
11 f(3*nodes_R-2) = -force; % sila pusobici zprava
```

V tomto kroku spočítáme lineární elasticitu naší sítě. Pro výpočet využijeme funkci `solve_elasticityLinear_gpu`. Nejdříve si připravíme na GPU výstupní pole *u_init* a pole z. Následně sestavíme matice tuhosti a hmotnosti z předchozí kapitoly.

```
1 % solve_elasticityLinearu_simple_gpu.m
2 u_init = zeros(params.nbfn*mesh.nn,1,'gpuArray');
3 z = ones(mesh.nn,1,'gpuArray');
4 K = stiffness_matrixP1_3D_elasticity_gpu      ...
5     (mesh.elems2nodes,mesh.nodes2coord,       ...
6     params.material.lambda,params.material.mu);
7 M = mass_matrixP1_3D_vector_gpu(mesh.elems2nodes,mesh.volumes);
```

Než se dostaneme k naplnění výstupních parametrů *u_init* a *eElastic* získáme vektor b z funkce `evaluate_loading_vector` , který bude již na GPU jelikož má funkce jako vstupní parametry GPU pole. Kvůli následujícímu kroku kde budeme indexovat matici tuhosti K, musíme tuto

matici přesunout zpět na operační paměti počítače pomocí funkce `gather` , jelikož MATLAB v této verzi nepodporuje indexaci řídké matice na GPU jak je vidět na Obrázku 8.2.

```
35  u_init(dofsMinim) = u_init(dofsMinim)+K(dofsMinim,dofsMinim)\bModif(dofsMinim);
Error using indexing
Sparse gpuArrays do not support indexing.

Error in solve_elasticityLinear_gpu (line 35)
u_init(dofsMinim) = u_init(dofsMinim)+K(dofsMinim,dofsMinim)\bModif(dofsMinim);

Error in solve_elasticity_3D_gpu (line 70)
u = solve_elasticityLinear_gpu(mesh,params,f);
```

Obrázek 8.2: Error při indexaci řídké matice na GPU.

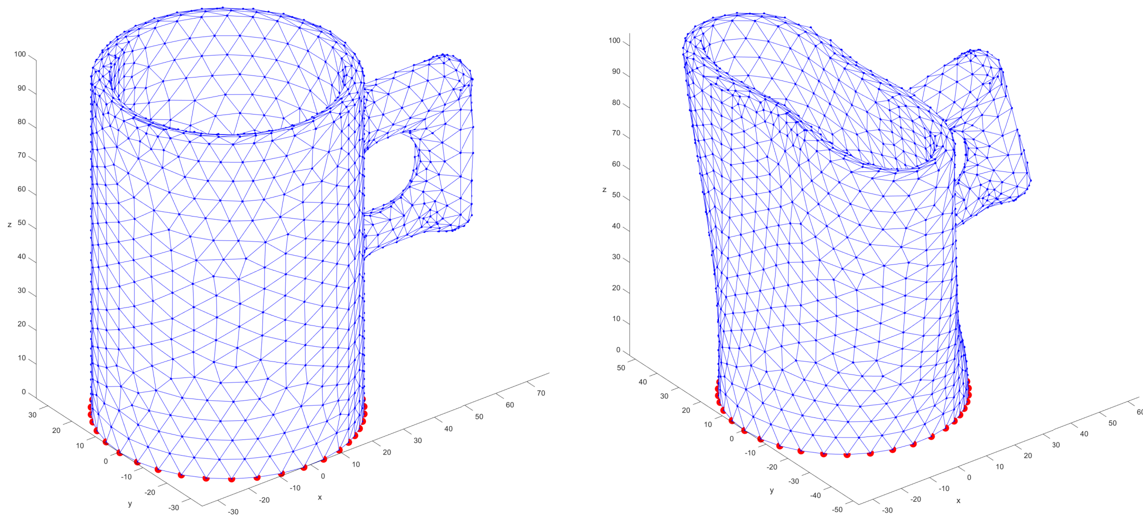
Po přesunutí matice tuhosti můžeme dopočítat vektor *u_init*. Pro náš případ je vektor *u_init* naplněn nulami, takže ho nepotřebujeme přičítat, ale v rámci univerzálnosti ho v kódu ponecháme.

```
1 % solve_elasticityLinear_gpu_simple.m
2 b = evaluate_loading_vector(mesh,f_v,params.nbfn,M,z);
3 bModif = b-K*u_init;
4 dofsMinim = gpuArray(mesh.dofs.Minim);
5 clear mesh M z % uvolneni pameti pro pametove narocne operace
6 K = gather(K);
7 K_dofs = gpuArray(K(dofsMinim,dofsMinim));
8 clear K % uvolneni pameti pro pametove narocne operace
9 u_init(dofsMinim) = ...
      u_init(dofsMinim)+K_dofs\bModif(dofsMinim);lambda,mu);
```

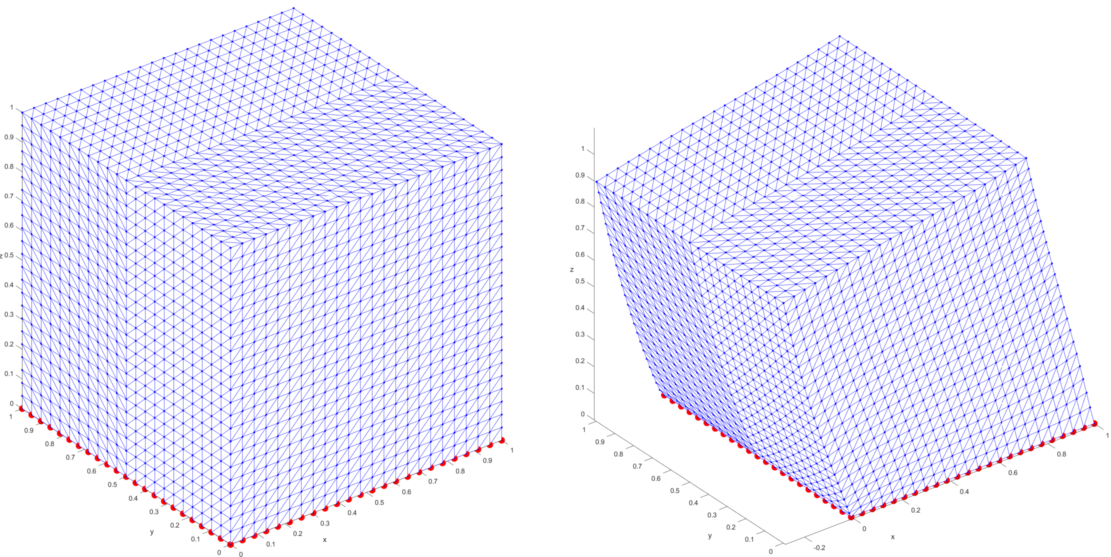
Posledním krokem je vytvoření kopie *coordinates*, ke které přičteme vektor *u*. Pro každou dimenzi zvláště přičteme každý třetí prvek, kde začátek iterace je index dimenze. Následně stačí pouze vykreslit původní a nový elasticitou deformovaný hrnek jak je vidět na Obrázku 8.3.

```
1 % solve_elasticity_3D_gpu.m
2 coordinates_new = coordinates;
3 coordinates_new(:,1) = coordinates_new(:,1) +u(1:3:end);
4 coordinates_new(:,2) = coordinates_new(:,2) +u(2:3:end);
5 coordinates_new(:,3) = coordinates_new(:,3) +u(3:3:end);
6 figure;
7 figure_mesh(coordinates_new,coordinates);
8 highlight_Dirichlet_3D(mesh,params);
```

Výpočet elasticity má velké požadavky na čas a paměť, z toho důvodu se pro kostky z Obrázku 8.1 testoval pouze na serveru a omezeném počtu elementů maximálně do levelu čtyři, poté by nám došla paměť na GPU. Kostku levelu 3.5, která vznikla při výpočtu úlohy elasticity můžeme vidět na Obrázku 8.4. Z Tabulky 8.4 vyčteme, že větší rychlost výpočtu elasticity na GPU projevuje s větším počtem prvků. Stejně tomu bylo i v případě sestavení matice tuhosti.



Obrázek 8.3: Vlevo původní a vpravo elasticitou deformovaný hrnek.



Obrázek 8.4: Vlevo původní a vpravo elasticitou deformovaná kostka (level 3.5).

Level	Velikost matice tuhosti	GPU GTX1650 Max-Q	GPU A30	CPU Intel i7-8565U	CPU AMD EPYC 7313
1.5	1029	0.513	0.316	0.872	0.486
2	5616	0.587	0.292	0.419	0.280
2.5	14739	1.533	0.642	0.742	0.476
3	39744	10.925	2.655	1.174	0.982
3.5	107811	out of memory	14.246	22.210	10.895
4	3234333	out of memory	13.393	52.455	51.565

Tabulka 8.4: Srovnání časů výpočtu elasticity mezi GPU a CPU v sekundách.

8.3.1 Zrychlení výpočtu úlohy elasticity

Výpočet úlohy elasticity se podařilo razantně zrychlit. V této podsekci si povíme kde nejvíce výpočet urychlil. Pomocí nástroje profiler si rozebereme program za běhu a podíváme se na časy běhů jednotlivých řádek. Toto zrychlení bylo testováno pouze na serveru, proto budeme srovnávat serverové GPU a CPU. Tento test se odehrával na kostce level 4.

Nejdříve si pomocí příkazů níže resetujeme profiler, spustíme skript pro výpočet elasticity a vygenerujeme report z profileru. To samé si poté uděláme i pro GPU verzi `example_15_solve_elasticity_3D`.

```
profile off
profile on
example_15_solve_elasticity_3D
profile report
```

Výstup profileru si seřadíme podle sloupce "Self Time (s)", abychom viděli, jaká část skriptu trvá nejdéle. Na Obrázku 8.5 vidíme, že nejvíce času při běhu programu potřebuje funkce `solve_elasticityLinear_simple` kliknutím na název funkce se do ní podíváme a zjistíme co tam tak dlouho trvá.

Function Name	Calls	Total Time (s)	Self Time* (s)	Total Time Plot (dark band = self time)
solve_elasticityLinear_simple	1	50.770	49.416	
stiffness_matrixP1_3D_elasticity	1	1.234	0.765	
amtam	1	0.311	0.311	

Obrázek 8.5: Profiler funkce solve_elasticity_3D_simple.

To samé vyčteme i u GPU verze skriptu z Obrázku 8.6, ale tady nám již nezabírá tolik času sestavení matice tuhosti ani funkce `amtam`, jelikož jsme obě funkce zrychlili pomocí GPU.

Function Name	Calls	Total Time (s)	Self Time* (s)	Total Time Plot (dark band = self time)
solve_elasticityLinear_simple_gpu	1	12.664	12.361	
entryInWhichRows	6	0.256	0.220	
unique>uniqueR2012a	11	0.130	0.130	

Obrázek 8.6: Profiler funkce solve_elasticity_3D_simple_gpu.

Přesuneme se tedy v profileru do funkce `solve_elasticityLinear_simple` respektive `solve_elasticityLinear_simple_gpu`. Výstupy profileru pro tyto skripty vidíme na Obrázcích 8.7 a 8.8. První zajímavý rozdíl najdeme v obou funkcích na řádku číslo 5. Můžeme vidět, že zrychlení sestavení matice tuhosti je šestinásobné. V tomto případě je zrychlení pouze jedna sekunda, v poměru celkového času je to pouze zlomek. Ale kvůli tomuto zrychlení funkce pro sestavení matice tuhosti již není v top 3 časově nejnáročnějších částí programu pro GPU verzi, jak můžeme vidět na Obrázku 8.6.

Time	Calls	Line	
		1	`function [u_init] = solve_elasticityLinear_simple(mesh,params,f_v)`
< 0.001	1	2	`u_init = zeros(params.nbfn*mesh.nn,1);`
< 0.001	1	3	`z = ones(mesh.nn,1);`
		4	
1.235	1	5	`K = stiffness_matrixP1_3D_elasticity(mesh.elems2nodes,mesh.nodes2coord,params.material.lambda,params.material.mu);`
0.119	1	6	`M = mass_matrixP1_3D_vector(mesh.elems2nodes,mesh.volumes);`
		7	
0.004	1	8	`b = evaluate_loading_vector(mesh,f_v,params.nbfn,M,z);`
		9	
0.004	1	10	`bModif = b-K*u_init;`
< 0.001	1	11	`dofsMinim = mesh.dofs.Minim;`
49.408	1	12	`u_init(dofsMinim) = u_init(dofsMinim)+K(dofsMinim,dofsMinim)\bModif(dofsMinim);`
< 0.001	1	13	`end`

Obrázek 8.7: Profiler solve_elasticityLinear_simple.

Největší zrychlení se poté odehrává na řádku 15 pro z Obrázku 8.8. Zrychlení oproti řádku 12 z Obrázku 8.7 je skoro čtyřnásobné. Tohoto zrychlení jsme dosáhli tím, že jsme si připravili pole *u_init* na GPU. Jelikož i pole *bModif* je na GPU bude se tedy soustava rovnic řešit na GPU. Toto se možná zdá poněkud zvláštní, když jsme museli v předchozí sekci převést matici K z paměti GPU do operační paměti počítače kvůli indexaci. Pouze to znamená, že indexace se provede na CPU pak vzniklou matice převede do paměti GPU. Jelikož máme tedy všechny členy operace v paměti GPU vyřešíme i soustavu rovni rychleji.

Time	Calls	Line	
		2	`function u_init = solve_elasticityLinear_simple_gpu(mesh,params,f_v)`
< 0.001	1	3	`u_init = zeros(params.nbfn*mesh.nn,1,'gpuArray');`
< 0.001	1	4	`z = ones(mesh.nn,1,'gpuArray');`
		5	
0.202	1	6	`K = stiffness_matrixP1_3D_elasticity_gpu(mesh.elems2nodes,mesh.nodes2coord,params.material.lambda,params.material.mu);`
0.106	1	7	`M = mass_matrixP1_3D_vector_gpu(mesh.elems2nodes,mesh.volumes);`
		8	
0.002	1	9	`b = evaluate_loading_vector(mesh,f_v,params.nbfn,M,z);`
		10	
< 0.001	1	11	`bModif = b-K*u_init;`
< 0.001	1	12	`dofsMinim = gpuArray(mesh.dofs.Minim);`
< 0.001	1	13	`clear mesh M z % Clear memory for memory heavy operation`
0.040	1	14	`K = gather(K);`
12.758	1	15	`u_init(dofsMinim) = u_init(dofsMinim)+K(dofsMinim,dofsMinim)\bModif(dofsMinim);`
< 0.001	1	16	`end`

Obrázek 8.8: Profiler solve_elasticityLinear_simple_gpu.

9. Práce s vytvořeným frameworkem

V této kapitole si vysvětlíme jak pracovat se vzniklými kódy. Na odkazu

Matlab-optimalization-with-GPU

najdeme všechny vytvořené kódy nebo kódy, které jsou součástí dodaného frameworku [20], aby bylo možné některé funkcionality otestovat.

Práce s examples

Ve složce examples najdeme jednoduché zdrojové kódy prezentované v průběhu práce. Všechny soubory s příklady jsou popsány v seznamu níže.

- **start_examples.m** skript, který podle parametrů automatizuje spouštění všech ostatních příkladů. Skript je rozdělen do devíti sekcí. Každá sekce má svůj spouštěcí parametr ve tvaru *run_<název sekce>*, který je třeba nastavit na 1 pro spuštění a 0 pro vypnutí, podle toho se poté při spuštění rozhodne zda má daná sekce běžet. Pokud sekce potřebuje vstupní parametry jako například *coordinates*, *elements* budou mu skriptem dodány. V části parametry skriptu lze nastavit možnosti běhu funkcí jako je například *stl_file_name* kterému můžeme napsat jméno dodaného stl souboru pro sestavení sítě.
- **Sekce *run_basics_GPU***
  - **example_1_gpuArrayDeclaration.m** je jednoduchý skript na ukázku deklarace pole, jeho převodu do GPU paměti a zpět.
  - **example_2_memoryAlocation.m** je skript na ukázku časové výhody před alokováním pole.
  - **example_3_memoryAlocationGPU.m** je obdobný skript jako **memoryAlocation.m**, ale pro GPU paměť.
  - **example_4_resourcesCosts.m** ukazuje rychlosti GPU vs CPU pro výpočet funkce *sinus* pro mnoho argumentů.
- **Sekce *run_areas***
  - **example_5_gaussMethod.m** je skript pro výpočet obsahu obrazce definovaného z bodů na hranici pomocí Gaussovy metody.
  - **example_6_heron.m** slouží pro výpočet obsahu obrazce pomocí Heronova vzorce. Pro spuštění tohoto skriptu je třeba dodat síť v strukturách *coordinates*, *elements* generované například pomocí funkce **meshGenerator.m**.
  - **example_7_det2x2.m** slouží pro výpočet obsahu obrazce pomocí determinantu matice 2x2. Pro spuštění tohoto skriptu je třeba dodat síť v strukturách *coordinates*, *elements* generované například pomocí funkce **meshGenerator.m**.
  - **example_8_det3x3.m** slouží pro výpočet obsahu obrazce pomocí determinantu matice 3x3. Pro spuštění tohoto skriptu je třeba dodat síť v strukturách *coordinates*, *elements* generované například pomocí funkce **meshGenerator.m**.
- **Sekce *run_dalauney_mesh*** obsahuje pouze skript **example_9_meshDelaunay.m**, který vytvoří trojúhelníkovou síť pomocí Delaunayho triangulace.

- **Sekce *run_vectorize*** obsahuje pouze skript **example_10_vectorization.m**, který slouží jako ukázka implementace vektorizace pro struktury *coordinates*, *elements*.

- **Sekce *run_areas_vectorize*** vektorizace kódu pro výpočet obsahu pomocí:
  - determinantu 2x2 **example_11_vectorize_det2x2.m**.
  - determinantu 3x3 **example_12_vectorize_det3x3.m**.
  - Heronova vzorce **example_13_vectorize_heron.m**.

- **Sekce *run_compare_stiffness*** obsahuje pouze **example_14_compare_stiffness.m** skript na otestování rychlosti sestavení matice tuhosti pro různě husté sítě. Hustota sítě závisí na obsahu pole *lvl* a počet testů závisí na počtu prvků tohoto pole.

- **Sekce *run_compare_elasticity*** obsahuje pouze **example_15_compare_elasticity_3D.m** skript na otestování rychlosti výpočtu úlohy elasticity pro různě husté sítě. Hustota sítě závisí na obsahu pole *lvl* a počet testů závisí na počtu prvků tohoto pole.

Složka examples ostatní soubory

Následující soubory ve složce jsou funkce využité ve skriptu *start_examples*, které nebyly pojmenovány stylem *example*, aby byly snáze přenositelné/použitelné.

- **meshGenerator.m** funkce vytvoří trojúhelníkovou síť pomocí PDE toolboxu ze zadaných souřadnic na hranici a vrátí struktury *coordinates*, *elements* a pole Dirichletových bodů *dirichlets*.

- **meshFromSTL.m** funkce pro nahrání tetrahedrální sítě ze STL souboru, jehož název je jejím vstupním parametrem. Vrátí struktury *coordinates*, *elements*.

- **hrnek.stl** STL soubor se sítí ukázkového hrnku. Potřebný pro funkci **meshFromSTL.m**.

- **getDependenciesAdvanced.m** skript, který v současné verzi nalezne soubory z projektu nutné ke spuštění skriptu v dané složce a překopíruje je do nové složky zadané parametrem *dep_dir_name*, soubory se poté budou nacházet ve složkách pojmenovaných podle jejich kořenového adresáře. Skript je užitečný pro nalezení a sestavení částí projektu nutných k jeho spuštění.

Podsložka Dependencies

Složka vytvořená pomocí **getDependenciesAdvanced.m**, která obsahuje skripty a funkce potřebné pro spuštění skriptu **start_examples.m**. Některé funkce byly vytvořeny v průběhu této práce (soubory s příponou *_gpu*) a jiné dodány ve frameworku [20].

- **phider_gpu.m** zjednodušená implementace funkce pro získaní derivace bázových funkcí.

- **stiffness_matrixP1_3D_elasticity_gpu.m** zjednodušená implementace funkce pro sestavení matice tuhosti pro výpočet úlohy elasticity.

- **mass_matrixP1_3D_gpu.m** zjednodušená implementace sestavení matice hmotnosti.

- **solve_elasticity_3D_simple_gpu.m** výpočet úlohy elasticity využívající GPU.

- **solve_elasticityLinear_simple_gpu.m** lineární řešič pro úlohu elasticity na GPU.

- **minAndMaxDist.m** funkce pro nalezení nejkratší a nejdelší vzdálenosti mezi body. Využita pro generování parametrů funkce **meshGenerator.m**.

10. Závěr

Dnešní technologie jsou na takové úrovni, že výpočty s využitím grafických karet jsou mnohonásobně rychlejší něž klasické počítání na procesoru počítače. Existuje mnoho technologií, které umožňují využít grafické karty. Není to však pouze o technologiích, ale i zkušenostech a znalostech programátora, který program urychluje.

V průběhu této práce jsem popsal parametry testovacího prostředí. Vysvětlil jsem jak provádět výpočty s využitím grafické karty NVIDIA v MATLABu. Popsal jsem strukturu ukládání trojúhelníkové sítě a možnosti výpočtu plochy rovinného obrazce, při kterém se tato struktura využívá. Taktéž jsem zmínil možnosti vytvoření trojúhelníkové sítě rovinného obrazce a vektorizované implementace výpočtu jeho plochy. Na závěr jsem využil a aplikoval tyto znalosti na výpočty pomocí metody konečných prvků. Pro ní se mi povedlo již vektorizované operace na GPU dále zrychlit, např. sestavení matice tuhosti úlohy lineární elasticity a řešení odpovídající soustavy lineárních rovnic.

Při práci jsem se prakticky přesvědčil o vysoké rychlosti výpočtů na grafických kartách. Zrychlení sestavení matice tuhosti v MATLABu bylo až devadesáti násobné (CPU Intel i7-8565U vs. GPU A30). Z daných testů vyplývá, že při využití grafické karty k výpočtu větších úloh je v průměru pětkrát rychlejší něž klasické počítání na procesoru. Vzhledem k velkému počtu jednotlivých kroků a jejich náročnosti jsem se soustředil pouze na technologii MATLAB, ačkoliv mojí částečnou ambicí bylo to samé otestovat i v jazyce Python.

Bibliografie

[1] *Jádra*. 2022. URL: https://www.computerhope.com/jargon/c/core.htm.

[2] *Vlákna*. 2022. URL: https://www.computerhope.com/jargon/t/thread.htm.

[3] *CPU clock speed*. 2022. URL: https://www.intel.com/content/www/us/en/gaming/resources/cpu-clock-speed.html.

[4] *CPU cache*. 2022. URL: https://en.wikipedia.org/wiki/CPU_cache.

[5] *What is GPU*. 2022. URL: https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html.

[6] *CUDA cores*. 2022. URL: https://www.trustedreviews.com/explainer/what-are-cuda-cores-4226433.

[7] *What's the Difference Between a CPU and a GPU?* 2009. URL: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/.

[8] *What is CUDA*. 2012. URL: https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/.

[9] Lukáš Zaorálek. *Úvod do technologie CUDA*. 2006. URL: https://www.root.cz/clanky/uvod-do-technologie-cuda/.

[10] *Matlab toolbox*. 2022. URL: https://www.mathworks.com/help/thingspeak/matlab-toolbox-access.html.

[11] *MATLAB Profiler-app*. 2022. URL: https://www.mathworks.com/help/matlab/ref/profiler-app.html.

[12] Alexej Moskovka, Jan Valdman a Marta Vohnoutová. „On minimizations of nonlinear energies from physics using fem in matlab, PPAM 2022 (přijato)“. In: (2022).

[13] Stanislav Daniš. *Základy programování v prostředí Octave a Matlab*. 1. vyd. MATFYZPRESS, 2009. ISBN: 978-80-7378-082-1.

[14] *Run matlab functions on a gpu*. 2022. URL: https://www.mathworks.com/help/parallel-computing/run-matlab-functions-on-a-gpu.html.

[15] *GpuDevice Matlab*. 2022. URL: https://www.mathworks.com/help/parallel-computing/parallel.gpu.gpudevice.html.

[16] Jung W. Suh a Youngmin Kim. *Accelerating Matlab with GPU computing: A Primer with examples*. Elsevier/Morgan Kaufmann, 2014. ISBN: 978-0-12-408080-5. DOI: https://doi.org/10.1016/C2012-0-06517-9. URL: https://www.sciencedirect.com/book/9780124080805/accelerating-matlab-with-gpu-computing?via=ihub=.

[17] *Matlab gather*. 2022. URL: https://www.mathworks.com/help/matlab/ref/tall.gather.html.

[18] *Matlab sparse*. 2022. URL: https://www.mathworks.com/help/matlab/ref/sparse.html.

[19] J Alberty, C Carstensen a SA Funken. „Remarks around 50 lines of Matlab: short finite element implementation“. English. In: *NUMERICAL ALGORITHMS* 20.2-3 (1999), s. 117–137. ISSN: 1017-1398. DOI: 10.1023/A:1019155918070.

[20] Talal Rahman a Jan Valdman. „Fast MATLAB assembly of FEM matrices in 2D and 3D: Nodal elements“. In: *Applied Mathematics and Computation* 219.13 (2013). ESCO 2010 Conference in Pilsen, June 21- 25, 2010, s. 7151–7158. ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2011.08.043. URL: https://www.sciencedirect.com/science/article/pii/S0096300311010836.

[21] *Formát STL*. 2022. URL: https://www.adobe.com/cz/creativecloud/file-types/image/vector/stl-file.html.

Seznam obrázků

2.1 Technologie CUDA. Převzato z [9]. 4
2.2 Příklad výstupu MATLAB profileru. Převzato z [12]. 6

3.1 Výstup funkce *gpuDevice*. 7

4.1 Jednoduchá trojúhelníková síť se třemi trojúhelníky. 11
4.2 Vizualizace 3D pole *coord*. 12
4.3 Příklad zjemnění sítě. 13

6.1 Delaunayho triangulace (vlevo) a PDE triangulace (vpravo). 16
6.2 Model hrnku zobrazen ve dvou prostředích. 18

7.1 Vektorizace 2D trojúhelníků 20
7.2 Vizualizace vzniklé 3D matice. 21

8.1 Modely kostky různé hrubosti. 26
8.2 Error při indexaci řídké matice na GPU. 29
8.3 Vlevo původní a vpravo elasticitou deformovaný hrnek. 30
8.4 Vlevo původní a vpravo elasticitou deformovaná kostka (level 3.5). 30
8.5 Profiler funkce solve_elasticity_3D_simple. 31
8.6 Profiler funkce solve_elasticity_3D_simple_gpu. 31
8.7 Profiler solve_elasticityLinear_simple. 32
8.8 Profiler solve_elasticityLinear_simple_gpu. 32

Seznam tabulek

2.1 Procesory užité k testování. 2
2.2 Grafické karty užité k testování. 3
2.3 CPU vs GPU [7]. 4

4.1 Vizualizace pole *coordinates* pro síť na Obrázku 4.1. 11
4.2 Vizualizace pole *elements* pro síť na Obrázku 4.1. 12
4.3 Vizualizace 2D pole, které se součástí *coord* pro síť na Obrázku 4.1. 12

8.1 Parametry testovacích modelů. 27
8.2 Srovnání časů sestavení matice tuhosti pro Poissonovu rovnici v sekundách. . 27
8.3 Srovnání časů sestavení matice tuhosti pro elasticitu v sekundách. 27
8.4 Srovnání časů výpočtu elasticity mezi GPU a CPU v sekundách. 30