České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical Engineering

Masivně paralelní algoritmy pro numerické řešení parciálních diferenciálních rovnic

Massively parallel algorithms for numerical solution of partial differential equations

Ing. Tomáš Oberhuber, Ph.D.

Souhrn

Tato habilitační přednáška se zabývá využitím grafických akcelerátorů (GPU, graphics processing units) pro urychlení výpočetně náročných numerických algoritmů zejména při numerickém řešení parciálních diferenciálních rovnic. Tyto akcelerátory byly od počátku navrženy pro paralelní zpracování kódu a pro práci s velkými objemy dat reprezentující textury. Díky tomu dnes nabízí mnohem vyšší výkon v porovnání s běžnými procesory. GPU obsahují velký počet jednoduchých výpočetních jednotek a návrh GPU navíc umožňuje relativně snadno tento počet navyšovat na rozdíl od procesorů. To naznačuje, že výkon GPU by i v budoucnu mohl růst stále rychleji a to i v porovnání s růstem výkonu procesorů. Nevýhodou návrhu GPU je ovšem to, že je poměrně náročné tyto architektury programovat. Zájemce o programování GPU musí mít velice dobrou znalost této architektury. Samotný návrh algoritmů pak vyžaduje zcela jiný přístup, než na co jsme zvyklí u procesorů. Řada numerických algoritmů je navíc výrazně sekvenčních a tím pádem zcela nevhodných pro běh na GPU. Paralelizace těchto algoritmů může být velmi netriviální a může vyžadovat intenzivní několikaletý výzkum. Ukazuje se, že je téměř nemožné modifikovat numerické knihovny vyvinuté původně pro procesory tak, aby mohly efektivně využívat vysoký výkon GPU. To vše způsobuje, že i po téměř dvaceti letech, co jsou GPU úspěšně využívána pro numerické výpočty, je jejich nasazení při numerickém řešení parciálních diferenciálních rovnic stále poměrně omezené.

Cílem této přednášky je ukázat způsoby, jak lze GPU využít právě pro řešení parciálních diferenciálních rovnic. Prezentujeme paralelní řešiče pro různé úlohy a numerické metody, jmenovitě metody konečných diferencí, konečných objemů a konečných prvků. U každého algoritmu uvádíme urychlení dosažené při porovnání doby výpočtu na srovnatelném procesoru a GPU. Autor tohoto textu je zároveň vedoucím projektu TNL (Template Numerical Library, www.tnl-project.org), což je numerická knihovna vyvíjena na katedře matematiky Fakulty jaderné a fyzikálně inženýrské. Tato knihovna nejen implementuje některé paralelní algoritmy a datové struktury, ale jednou z priorit tohoto projektu je usnadnění vývoje numerických řešičů pro paralelní architektury obecně. Knihovna proto nabízí unifikované rozhraní pro vývoj paralelních algoritmů, které pak mohou být automaticky provozovány jak na vícejádrových procesorech tak i na GPU. Některé numerické řešiče pro parciální diferenciální rovnice byly implementovány právě s pomocí knihovny TNL.

Summary

This habilitation thesis deals with a use of graphics processing units (GPU) for accelerating solution of computationally expensive numerical algorithms, especially solvers of partial differential equations. These accelerators were from the beginning designed for parallel processing and for work with large data sets representing textures. Because of this, they offer significantly higher computational performance compared to common CPUs. GPUs contain many computational cores and the design of GPUs allows increasing this number relatively easily, in contrast to CPUs. It indicates that the performance of GPUs could be increasing faster compared with the increase of performance of CPUs. A disadvantage of the GPU design is its complexity, which makes this architecture hard for programming. Developers who want to write code for GPU must have deep knowledge of this architecture, which is not true for common CPUs. The design of parallel algorithms for GPUs requires a completely different approach compared to what we are used to from programming of CPUs. A number of numerical algorithms are strongly sequential and thus completely inadequate to run on GPUs. Parallelization of such algorithms can be a difficult task, demanding years of research. It turns out that it is almost impossible to modify older numerical libraries which were designed for CPUs for efficient run on GPUs. This all is a reason why even after almost twenty years of using GPUs for numerical simulations, their setting in numerical solution of partial differential equations is still rather limited.

The aim of this presentation is to show the ways how one can use GPUs for numerical solution of partial differential equations. We present parallel solvers for different problems and numerical methods, namely methods of finite difference, finite volume and finite element. For each algorithm we show speedup obtained by comparison of computational times on CPU and GPU. The author of this text is also a leader of TNL (Template Numerical Library, www.tnl-project.org) project, which is a numerical library being developed at the Department of Mathematics at Faculty of Nuclear Sciences and Physical Engineering. This library offers a number of algorithms and data structures. One of the main priorities of this project is to simplify the development of parallel algorithms to run on multicore CPUs and GPUs. Some numerical solvers of partial differential equations presented in this text were implemented with the use of TNL library.

Klíčová slova

masivně paralelní algoritmy
grafické akcelerátory, GPU
parciální diferenciální rovnice
zpracování obrazových dat
výpočetní dynamika tekutin
indefitní úlohy

Keywords

massively parallel algorithms
graphical processing units, GPU
partial differential equations
image processing
computational fluid dynamics
indefinite problems

Obsah

1 Úvod

Řešení parciálních diferenciálních rovnic patří mezi jedny z výpočetně nejnáročnějších úloh. Počínaje snahou o co nejpřesnější řešení turbulentního proudění, které dokáže vytížit i dnešní nejvýkonnější superpočítače, přes různé diferenciální modely aplikované v materiálových vědách, energetice, biologii až po zpracování medicínských dat, které bychom rádi prováděli v řádu sekund maximálně minut. Poptávka po neustále větším výpočetním výkonu neutichá od počátků využití počítačů pro numerické metody v polovině minulého století. Neustále se nám odkrývají nové možnosti, jak odvozovat stále komplexnější a výpočetně náročnější metody. A stále existuje řada fyzikálních problémů na jejichž řešení jsou i dnešní nejvýkonnější superpočítače nedostačující. Jmenovat můžeme například již zmíněné turbulentní proudění nebo výpočetní jadernou fyziku. I proto jsme dnes svědky závodu o postavení prvního superpočítače s výkonem větším než jeden exaflop. Je velice pravděpodobné, že takový počítač bude intenzivně využívat grafické akcelerátory (GPU, graphic processing unit).

Od počátku konstrukce počítačů založených na von Neumannově architektuře až téměř do konce dvacátého století rostl výkon procesorů (CPU, central processing unit) exponenciálně zejména díky rostoucímu taktu. Ten začínal na v řádech kHz a dospěl až k několika GHz. Souběžně s tím docházelo ke zmenšování výrobního procesu, což umožňovalo na stejnou plochu vměstnat více logických prvků a vytvářet tak více sofistikované architektury. Samotný růst frekvence ovšem výrazně navyšuje energetickou spotřebu mikročipů a jakmile se tato frekvence dostala na řád gigahertzů, začaly se projevovat problémy nejen s vyzářeným teplem ale také se spotřebou elektrické energie. Výrobci procesorů se tak vydali cestou navyšování počtu procesorových jader. Tím jsme byli schopni dosáhnout vyšší energetické efektivity ovšem za cenu nutnosti vyvíjet paralelní aplikace. Takto konstruované procesory patří mezi architektury se sdílenou pamětí, tzn. že všechna procesorová jádra jsou připojena ke sdíleným paměťovým modulům. A právě propojení procesorových jader mezi sebou a se sdílenými paměťovými moduly se stává obtížné s rostoucím počtem těchto jader. Historie ukazuje, že největší systémy se sdílenou pamětí dosahují maximálního počtu nižších stovek jader. V nedávné minulosti tento fakt potvrdil například akcelerátor Intel Xeon Phi, který byl navržen jako procesor s velkým počtem jednodušších jader. Vývoj této architektury byl ovšem ukončen, velice pravděpodobně právě z důvodu obtížného navyšování počtu jader.

Již před příchodem vícejádrových procesorů se koncem devadesátých let dvacátého století objevily výkonné grafické akcelerátory. Velice brzy se ukázalo, že tyto karty je možné použít i pro jiné výpočty než jen vykreslování grafických scén. GPU se v průběhu let vyvíjely v programovatelná zařízení. Nejprve proto, aby vývojáři zejména počítačových her mohli implementovat své vlastní grafické efekty. Za tím účelem používané tzv. *pixel shadery* a *vertex shadery* byly později přetvořeny v obecné výpočetní jednotky. Ty jsou na rozdíl od procesorových jader mnohem jednodušší a na GPU jich je vetší počet. Jsou rozděleny do několika nezávislých multiprocesorů, což znamená, že přímo spolu mohou komunikovat pouze jednotky v rámci jednoho multiprocesoru. Tím GPU chytře obchází

již zmíněnou past architektur se sdílenou pamětí. Díky tomu dnes mohou mít GPU i několik tisíc výpočetních jednotek. Nevýhodou tohoto přístupu jsou výrazné komplikace pro vývojáře.

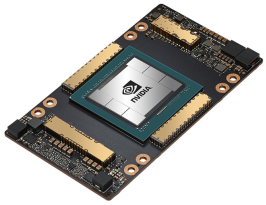
Pokud se výrobci mikroprocesorů potýkali s problémem rostoucí energetické spotřeby, o to víc to platí pro výrobce paměťových modulů, jejichž výkon díky tomu rostl výrazněji pomaleji než u procesorů. Tím se stalo, že rozdíl mezi výkonem procesorů a paměťových susbsystémů se zvětšoval po dobu více než dvou dekád až do stavu, kdy procesor může čekat i více než 200 taktů na přenos dat z paměťových modulů. Paměti jsou tedy až o dva řády pomalejší než současné procesory. V mnoha aplikacích je tím nejdůležitějším činitelem ovlivňujícím výkon právě efektivní přístup do paměti a intenzivní využívání vyrovnávacích pamětí tzv. *cache memory*. Paradoxně tak platí, že např. operace z lineární algebry, na kterých stojí výrazná většina numerických metod a řešičů, je limitována v prvé řadě pomalým přístupem do paměti a až potom výkonem procesoru. V takové situaci pak ani navyšování počtu procesorových jader nepřinese žádné urychlení. Další výhodou GPU tak je fakt, že paměti dodávané s těmito akcelerátory jsou výrazně rychlejší než paměti připojené k procesorům. I to s sebou ovšem nese některé komplikace. Díky tomu, že GPU využívá vlastní optimalizované paměťové moduly, není tato paměť přímo sdílená s procesorem a vývojáři tak musí explicitně provádět kopírování dat mezi operační pamětí procesoru a pamětí spojenou s GPU. Manipulace se dvěma adresovými prostory pak výrazně komplikuje vývoj algoritmů pro GPU.

Obrázek 1 ukazuje porovnání architektur CPU a GPU. Ukazuje, že energetická efektivita GPU je výrazně vyšší než je tomu u procesorů. Jelikož právě energetické efektivita je klíčovým parametrem při návrhu nejvýkonnějších superpočítačů, lze nalézt GPU v mnoha systémech uvedených na seznamu Top 500 [1].

O výhodách architektury GPU pro vědecké výpočty již dnes nelze ani v nejmenším pochybovat. Přesto by se dalo říci, že GPU ve vědeckých výpočtech nenašla tak významné uplatnění, jak by bylo možné. Důvodem je již zmíněná výrazná odlišnost od obecných procesorů. To způsobuje, že je prakticky nemožné portovat již existující knihovny a softwarové balíky původně navržené na CPU pro běh na GPU. GPU často vyžaduje jinou organizaci dat v paměti, což by vyžadovalo kompletní přepsání mnoha datových struktur. U velkých balíků jako OpenFOAM, PETSc, Ansys apod. je toto prakticky nemožné. Zřejmě i to je důvodem, proč GPU jsou dnes využívána mnohem více v oblasti strojového učení, neboť významné softwarové balíky jako TensorFlow, Theano, Caffe2 byly napsány až po příchodu nástroje CUDA (Compute Unified Device Architecture), který významně zjednodušil vývoj kódu pro GPU od společnosti Nvidia.

Pokud jde o softwarové nástroje s podporou GPU využitelné pro řešení parciálních diferenciálních rovnic, omezuje se jejich výčet spíše na menší jednoúčelové nástroje. Kompletní ucelený balík, alespoň pokud je autorovi známo, neexistuje. Alespoň částečně se tuto mezeru snaží zaplnit projekt TNL (Template Numerical Library, www.tnl-project.org) jenž je pod vedením autora vyvíjen na katedře matematiky na Fakultě jaderné a fyzikálně inženýrské od roku 2004. V tomto textu předvedeme základní algo-

[1] www.top500.org



	Nvidia A100	AMD Instinct MI100
Počet jader	6912 @ 1.41GHz	7680 @ 1.2GHz
Max. výkon	19.5/9.7 TFlops	23/11 TFlops
Max. výkon s tenzory	156 /– TFlops	–/–
Max. RAM	80 GB	32 GB
Datová propustnost	2039 GB/s	1200 GB/s
Energetická náročnost	400 W	300 W

	Intel Xeon W-3375	AMD EPYC 2 Rome 7H12
Počet jader	38 @ 2.5 GHz	64 @ 2.4GHz
Max. výkon	≈ 1.14 / 0.57 TFlops	≈ 2 / 1 TFlops
Max. výkon s tenzory	–/–	–/–
Max. RAM	4 TB	2 TB
Datová propustnost	204 GB/s	204 GB/s
Energetická náročnost	270W	280 W

Obrázek 1: Porovnání architektur grafických akcelerátorů a procesorů.

ritmy a datové struktury nezbytné pro vývoj paralelních numerických řešičů parciálních diferenciálních rovnic. Některé z nich jsou již i součástí knihovny TNL jejíž prioritou je i obalení těchto algoritmů a datových struktur do rozhraní využívající moderních vlastnosti jazyka C++ za účelem vytvoření jednoduchého a flexibilního rozhraní pro uživatele neznalé detailů programování pro GPU.

2 Lineární algebra

Vektorové operace patří mezi základní operace lineární algebry. Jak již bylo řečeno, jsou tyto operace nejvíce omezovány datovou propustností použité hardwarové architektury. Je proto potřeba tyto operace implementovat s využitím optimálních přístupů do paměti. Přesto, že se jedná o velice jednoduché operace, jejich optimalizovaná implementace může být poměrně komplikovaná. Toto platí zejména pro operace s maticemi, kde lze efektivním využitím vyrovnávacích pamětí urychlit výpočty až o jeden řád. Jelikož optimální implementace může být výrazně závislá na použité hardwarové architektuře, různí dodavatelé hardwaru často poskytují vlastní implementace. Ty většinou používají standardizované rozhraní Blas (Basic Linear Algebra Subprorgrams), které následně umožňuje snadné portování numerického kódu pro různé architektury. Blas se dělí na tři úrovně podle složitosti implementovaných algoritmů. Blas level 1 implementuje vektorové operace se složitostí $O(n)$, pokud n značí velikost vektoru. Blas level 2 implementuje operace s maticemi se složitostí $O(n^2)$, kde n značí rozměry matice. Nakonec Blas level 3 implementuje maticové algoritmy se složitostí $O(n^3)$. S příchodem GPU logicky vznikly i varianty Blasu optimalizované pro běh na těchto kartách. Například pro karty od společnosti Nvidia jde o knihovnu Cublas [1].

Nevýhodu standardu Blas lze z dnešního pohledu spatřovat v jeho zastaralosti. Byl navržen koncem sedmdesátých let a jeho základ vychází z knihovny psané v jazyce Fortran. Jde tedy o čistě funkcionální přístup, který nevyužívá moderní programovací techniky jako například šablonové programování nebo lambda funkce. V této části ukážeme, jak lze těchto nástrojů využít k snazšímu vývoji paralelních algoritmů, které navíc mohou být efektivnější.

2.1 Vektorové operace

Jako první si předvedeme využití techniky zvané výrazové šablony (expression templates) pro zpracování vektorových výrazů. Jako příklad budeme uvažovat vyčíslení následujícího výrazu.

$$\vec{x} = \vec{a} + 2\vec{b} + 3\vec{c} \tag{1}$$

To lze například s pomocí knihovny Cublas implementovat takto:

```
cublasHandle_t c_h;
cublasSaxpy(c_h,N,1.0,a,1,x,1);
cublasSaxpy(c_h,N,2.0,b,1,x,1);
cublasSaxpy(c_h,N,3.0,c,1,x,1);
```

Jak můžeme vidět, samotný kód není velmi přehledný a je potřeba často hledat v referenční příručce knihovny Cublas pro jeho správné pochopení. Takováto implementace navíc provádí sčítání jednoho vektoru po druhém. Nejprve se načte $\vec{a}$ do vektoru $\vec{x}$. Následně se přičte $2\vec{b}$ k vektoru $\vec{x}$ a nakonec s přičte $3\vec{c}$. Tímto přístupem se do vektoru $\vec{x}$ zapisuje celkem třikrát, což výrazně zatěžuje paměťový subsystém. A jak jsme již zmínili několikrát, právě ten je úzkým hrdlem při zpracování operací z lineární algebry. Základním problémem je tu samotný návrh Cublasu potažmo Blasu, který může poskytnout jen předkompilované funkce. Naproti tomu moderní překladače jazyka C++ jsou plně programovatelné [2] a lze je využít ke generování kódu podle našich potřeb.

S pomocí výrazové šablony můžeme napsat přímo daný algebraický výraz v následujícím tvaru:

```
x = a + 2 * b + 3 * c;
```

Překladač ho nejprve naparsuje a pak vygeneruje specializovaný kód pro vyčíslení právě tohoto výrazu. S pomocí nástroje CUDA je pak stejným způsobem možné generovat specializované kernely pro GPU. Výhodou je, že provádíme vždy jen jedno čtení z vektorů $\vec{a}, \vec{b}$ a $\vec{c}$ a jen jeden zápis do vektoru $\vec{x}$. Tím dostáváme efektivnější výpočet. Navíc použití výrazových šablon je mnohem jednodušší pro uživatele. V knihovně TNL je u každého vektoru uvedeno, na jakém zařízení je alokován. Překladač tak sám pak pozná, zda má generovat kód pro CPU nebo kernel pro GPU.

V další částí si ukážeme, jak lze s výhodou využít lambda funkcí z jazyka C++ na příkladu efektivního programování s operací redukce. Redukce se využívá všude tam, kde máme za vstup jeden nebo více vektorů a výsledkem je jeden nebo několik skalárů. Typickým příkladem může být výpočet vektorové normy:

```
float norm( 0.0 )
for( int i = 0; i < size; i++ )
    norm += abs( a[ i ] );
```

Toto je snadná implementace pro jedno procesorové jádro. Pokud bychom chtěli stejnou operaci implementovat v CUDA na GPU, museli bychom napsat více než 150 řádků kódu. Tzv. paralelní redukce je typickým příkladem algoritmu, který je velmi triviální k implementaci na CPU ale relativně komplikovaný na GPU. Jednak je potřeba implementaci redukce na GPU dobře porozumět, což vyžaduje dobrou znalost architektury GPU a to nelze očekávat od běžného vývojáře numerických řešičů. Za druhé, i pokud dobře rozumíme tomu, jak paralelní redukci na GPU implementovat, je to práce velice

zdlouhavá. I zkušený vývojář se ji raději vyhne, pokud je to alespoň trochu možné. V této situaci nám mohou výrazně pomoci lambda funkce. S jejich pomocí lze výpočet normy zapsat takto:

```
1  auto fetch=[=](int i)->float{
2      return abs(a[i]);};
3  auto reduce=[](float x,float y)->float{
4      return x+y;};
5
6  float result( 0.0 );
7  for( int i = 0; i < size; i++ )
8      reduce( result, fetch( i ) );
```

Na první pohled složitější zápis je ale mnohem výhodnější. Celý kód jsme rozdělili na dvě části. První z nich je definice lambda funkcí na řádcích 1-4, které definují, co chceme s pomocí redukce vypočítat. Funkce `fetch` načítá data ze vstupního vektoru, a jelikož počítáme l_1 normu vektoru, vypočítá i absolutní hodnotu jednotlivých složek. Jednotlivé mezivýsledky se pak předávají funkci `reduce`, která provádí jejich redukci pomocí vhodné asociativní operace. V našem případě jde o sčítání. Druhá část tohoto kódu je for cyklus na řádcích 7-8. Ten je nyní zcela závislý jen na použitých lambda funkcích. Pokud chce uživatel vypočítat jinou redukci, změní pouze definici lambda funkcí, ale zmíněný for cyklus zůstane zachován. Důležité je, že definice lambda funkcí nezávisí na použité hardwarové architektuře, na té závisí pouze zmíněný for cyklus. Pokud ho nahradíme příslušným kernelem pro GPU, který bude využívat předdefinované lambda funkce stejným způsobem, můžeme snadno celou redukci počítat na GPU. Přesně takto funguje tzv. *flexibilní redukce* v knihovně TNL.

Jelikož se operace redukce vyskytuje v mnoha operacích jako je např. skalární součin, výpočet různých vektorových norem, porovnávání dvou vektorů apod. je dobré mít možnost pracovat s redukcí snadno a efektivně. Podobně jako u výrazových šablon i zde využíváme překladač k tomu, aby nám generovala specializované kernely nejen pro GPU. Jako příklad ještě uvedeme možnost snadného sloučení dvou kernelů do jednoho. Následující kód s pomocí knihovny TNL vypočítá výraz (1) včetně jeho normy:

```
1  auto fetch=[=] __cuda_callable__ (int i)->float mnutable {
2      auto aux = x[i] = a[i] + 2*b[i] + 3*c[i];
3      return abs(aux);};
4  auto norm = TNL::Algorithms::reduce< TNL::Devices::Cuda >
5      (0,size,fetch,TNL::Plus{});
```

Lambda funkce `reduce` je zde nahrazena předefinovanou funkcí `Plus`. Lambda funkce `fetch` v sobě obsahuje jak výpočet výrazu (1), tak i okamžité vrácení absolutní hodnoty jednotlivých složek. Tím se opět ušetří jeden zápis do vektoru $\vec{x}$ a optimalizují se tak přístupy do paměti.

Výše uvedeným postupem se tak daří výrazně usnadnit programování GPU karet i pro programátory, kteří nemají detailnější znalost GPU. Zároveň se často daří vytvářet efektivnější kód. V následující části si ukážeme jak podobný princip aplikovat i pro řídké matice.

2.2 Řídké matice a segmenty

Maticové operace patří mezi ty nejčastější v oblasti numerické matematiky. Pokud jde o řešení parciálních diferenciálních rovnic, výsledkem aplikace některé z nejoblíbenějších metod jako jsou konečné diference, konečné objemy nebo konečné prvky bývá soustava lineárních rovnic s řídkou maticí. Tato soustava je nejčastěji řešena pomocí některé iterativní metody Krylovových podprostorů. Ty jsou založené zejména na operaci násobení matice a vektoru. V případě řídkých matic se tato operace označuje jako SpMV (sparse-matrix vector multiplication).

U řídkých matic je převážná většina prvků nulových, a proto se k jejich ukládání do paměti počítače používají specializované formáty, které ukládají jen nenulové maticové prvky. Ty jsou organizovány tak, aby se během provádění operace SpMV při čtení maticových prvků přistupovalo do paměti sekvenčně. V takovém režimu dokáže procesor přenášet data z paměťových modulů nejefektivněji. Mezi nejoblíbenější formáty patří formát CSR (Compressed Sparse Rows) [3]. Pokud jde o GPU, i zde je nutný sekvenční přístup do paměti. Jde ale o trochu jiný princip, který se označuje jako sloučené přístupy do paměti (coallesced memory accesses). K provedení těchto sloučených přístupů GPU vyžaduje, aby 32 po sobě jdoucích vláken, tzv. *warp* četl souvislý blok 128 bajtů zarovnaný v paměti na násobek 128 bajtů. Pokud u CSR formátu mapujeme jedno CUDA vlákno na jeden maticový řádek, nebudou tyto podmínky splněny a přenos dat bude velice pomalý. Tento přístup se označuje jako Scalar CSR [4]. Druhou možností je mapovat jeden warp vláken na jeden maticový řádek, který je často označován jako Vector CSR [4]. V jedné z prvních publikací zabývající se ukládáním řídkých matic na GPU Bell a Garland [4] reportují, že Vector CSR podává lepší výkon než Scalar CSR, ale oba přístupy se jevily jako neefektivní v porovnání s jinými.

Dalo by se říci, že počátky výzkumu ukládání řídkých matic na GPU se nesly v duchu vývoje různých modifikací formátu Ellpack [3]. Základní Ellpack formát se pro GPU modifikuje tak, že se provede transpozice nenulových prvků v paměti. Tím pak při mapování jednoho CUDA vlákna na jeden maticový řádek dojde ke splnění podmínek pro sloučené přístupy do paměti. Nevýhoda základního Ellpack formátu je v tom, že využívá tzv. zarovnávací nulové prvky a pro každý řádek matice alokuje tolik prvků, kolik vyžaduje nejvíce zaplněný řádek. Stačí pak, aby v matici byl jeden výrazně zaplněný řádek nebo dokonce plný řádek a Ellpack formát pak vyžaduje alokaci více paměti, než kdybychom matici ukládali jako plnou tj. i se všemi nulovými prvky. Různé modifikace formátu Ellpack se snaží řešit nejen tento problém, ale také zlepšit mapování CUDA vláken na maticové řádky v počtu odpovídajícím zaplnění daného maticového řádku. Právě z důvodů snahy najít optimální organizaci dat v paměti spolu s vybalancováním zátěže

jednotlivých multiprocesorů GPU patří výzkum formátů pro ukládání řídkých matic na GPU mezi jedny z nejzajímavějších a např. studenti zajímající se o programování GPU se na této úloze mohou naučit hodně o fungování GPU.

Vraťme se ale k modifikacím formátu Ellpack. Jmenovat můžeme například *hybridní formát* [5]. Zde autoři využívají k uložení extrémně zaplněných řádků formátu COO (ten ukládá explicitně pro každý nenulový maticový prvek kromě jeho hodnoty i index sloupce a index řádku). Po určitou dobu byl tento formát uznáván jako referenční, dnes již je ovšem překonaný. V článku [6] autor tohoto textu publikoval formát nazvaný *Row-grouped CSR* formát. Jde o modifikaci formátu Ellpack velice podobnou formátu Sliced Ellpack [7]. Oba formáty dělí maticové řádky do řezů o velikosti 32 řádků. Každý řez se pak ukládá jako samostatná matice ve formátu Ellpack. Pokud matice obsahuje výrazně zaplněný řádek, bude jím ovlivněn jen příslušný řez. Jde o výraznou optimalizaci vůči základnímu Ellpacku. Tyto dva formáty jsou velice vhodné pro matice, které mají všechny řádky přibližně stejně zaplněné. V tom případě oba formáty profitují ze své jednoduchosti a podávají velice dobrý výkon. Pro mnoho vzorů zaplnění řídkých matic ale nejsou příliš vhodné. Další vylepšení od autora tohoto textu bylo publikováno v [8]. Tento formát, nazývaný Chunked Ellpack, se zaměřuje právě na matice, které mají výrazně proměnlivé zaplnění maticových řádků. Formát umožňuje mapovat více CUDA vláken na jeden řádek. Podobnými modifikacemi jsou i Bisection Ellpack [9], SELL-C-σ, [10], AdEll [11], CoAdEll [12], AdEll+ [13], nebo SURAA [14].

Pozitivní ohlasy vzbudil formát CSR5 [15], který je založený na segmentované paralelní redukci. V posledních letech se pozornost zaměřila spíše na vývoj efektivních kernelů pracujících s formátem CSR. Ukazuje se, že modernější GPU, která jsou vybavena většími vyrovnávacími pamětmi, si zřejmě dokáží lépe poradit s ne úplně optimálními přístupy do paměti. Proto se výkon dosažitelný s formátem CSR zlepšuje a navíc jde o naprosto nejoblíbenější formát pro práci s řídkými maticemi, který využívá většina numerických knihoven. Fakt, že takto uložené matice stačí pouze překopírovat na GPU a není nutné provádět žádnou transformaci, mluví jasně pro formát CSR. Navíc jej využívá i knihovna Cusparse [16] od společnosti Nvidia. Tato knihovna je v současnosti považována za referenční standard. S formátem CSR pracují např. kernely VCSR [17], CSR s automatickým laděním parametrů [18], Adaptive CSR [19], LSRB-CSR [20] nebo LightSpMV [21], což jasně dokazuje, že v této oblasti stále probíhá aktivní výzkum. Široký přehled formátů, které se objevily do roku 2017, lze najít v [22]. Nejnověji se i objevují snahy o využití metod strojového učení za účelem optimálního mapování CUDA vláken na maticové řádky [23].

2.3 Segmenty

Podobně jako jsme u vektorových operací využili lambda funkcí v jazyce C++ pro zobecnění paralelní redukce, je možné lambda funkce využít i pro zobecnění operací s řídkými maticemi nebo jim podobných operací. Jak již bylo řečeno, operace SpMV je typická tím, že vyžaduje co nejlepší organizaci dat v paměti a mapování vláken na

maticové řádky. Řídké matice se mohou výrazně lišit svým vzorem zaplnění a ukazuje se, že zejména na GPU může docházet k velkým rozdílům ve výkonu právě v závislosti na zaplnění matice. Zdá se tedy, že by bylo výhodné mít možnost snadno testovat různé formáty podle toho, jaký vzor zaplnění generuje daná úloha.

Dále je možné si všimnout, že v oblasti vysoce výkonných výpočtů (high-performance computing, HPC) existuje více úloh, které se nápadně podobají zejména operaci SpMV. Jde například o:

1. **Řídké grafy a grafové algoritmy** tvoří důležitou třídu úloh. Grafy lze vyjadřovat pomocí adjacenčních matic [24], proto nepřekvapí, že řada grafových algoritmů bude svou podstatou podobná maticovým operacím.
2. **Nestrukturované numerické sítě** jsou důležité datové struktury zejména pro metody konečných prvků a konečných objemů. Tyto sítě lze vyjádřit pomocí adjacenčních a incidenčních matic [25], které jsou řídké.
3. **Metoda Particle-in-cell** je metoda kombinující lagrangeovský a eulerovský přístup pro simulaci proudění. Eulerovský přístup je použit pro popis proudění hustší látky, např. vody nebo vzduchu. Lagrangeovský přístup pak popisuje řidší látku jako např. prach nebo písek, u které je možné simulovat jednotlivé částice. Ty jsou promítnuty na použitou numerickou síť, na které je počítána eulerovská metoda. Pro vyšší efektivitu těchto metod ukládáme seznam částic v dané buňce numerické sítě. Právě tento počet se může výrazně lišit buňku od buňky podobně jako je tomu se zaplněním různých řádků u řídkých matic.
4. **K-means shluková analýza** je algoritmus známý z datové analýzy nebo strojového učení. Používá se pro klasifikaci vektorů. Cílem je rozdělit zadané vektory do k klastrů podle zvolené metriky definující blízkost vektorů. Ačkoliv jde o NP-úplnou úlohu, existuje iterativní aproximační algoritmus. V průběhu jednotlivých iterací je potřeba udržovat seznam vektorů v daných klastrech. Tento počet se opět může výrazně lišit klastr od klastru.
5. **Hešování** je velice efektivní technika pro práci s řídkými daty. Hešovací algoritmy pro GPU se objevují teprve v posledních letech. HashGraph [26] je jedním a nich a využívá právě CSR formát k řešení hešovacích kolizí tím, že pro každou kolizi v daném řádku hešovací tabulky alokuje více slotů. Počet těchto slotů se také může výrazně lišit pro různé řádky tabulky.

V [27] autor tohoto textu navrhuje abstrakci nad formáty pro řídké matice zvanou *segmenty*. Ta řeší mapování prvků příslušejícím k různě velkým skupinám do lineárního pole. K jednotlivým prvkům pak můžeme přistupovat pomocí indexu skupiny (tj. např. číslo řádku matice nebo číslo klastru) a lokálního indexu v rámci skupiny (tj. např. pořadí nenulového prvku v maticovém řádku nebo pořadí vektoru v rámci klastru). Segmenty na základě těchto indexů dokáží každému elementu přiřadit globální index v rámci lineárního pole, kde jsou jednotlivé prvky uloženy. Segmenty pak umožňují rychlé iterování přes všechny prvky nebo výpočet redukce v rámci jednotlivých skupin. Jednotlivé segmenty reprezentují různé formáty pro ukládání řídkých matic. Díky tomu

lze tyto formáty s výhodou využít i pro efektivní řešení jiných úloh. Název této abstrakce je odvozen od segmentované paralelní redukce, která připomíná segmenty založené na CSR formátu. Podstatu této abstrakce podrobněji znázorňuje obrázek 2.

Obrázek ukazuje uložení řídké matice M ve formátu CSR. Tento formát využívá dvou velkých polí v nichž se ukládají hodnoty nenulových maticových prvků a jejich sloupcové indexy. Maticové prvky jsou zde uloženy po řádcích a hranice jednotlivých řádků jsou uloženy v poli ukazatelů na jednotlivé řádky ve formě indexů odkazujících se do polí hodnot a sloupcových indexů. Abstrakce segmentů tak v sobě obsahuje právě pole ukazatelů na jednotlivé řádky a na jejich základě pak implementuje jednotlivé operace. Abstrakce segmenty může být založena i na jiných formátech pro ukládání řídkých matic. Jelikož mají segmenty jednotné rozhraní, je pak možné snadno zkoušet různé formáty pro danou úlohu.

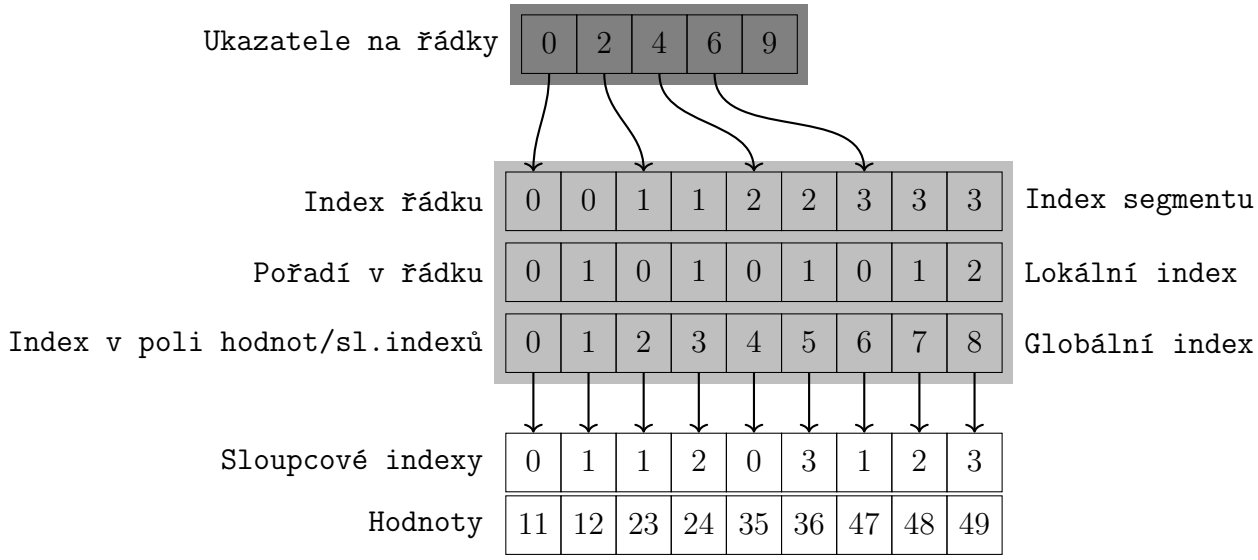
Abstrakce segmentů je implementována v knihovně TNL. Díky ní nabízí tato knihovna jednotné rozhraní pro práci s řídkými maticemi a umožňuje snadno měnit použitý maticový formát, což se zdá být jako unikátní vlastnost této knihovny. Například součin řídké matice a vektoru lze s pomocí segmentů implementovat takto:

```
1  vectorProduct( const InVector& inVector, OutVector& outVector )
2  {
3      auto fetch = [=] __cuda_callable__ ( Index globalIdx ) -> Real {
4          const Index column = columnIndexes[ globalIdx ];
5          return values[ globalIdx ] * inVector[ column ];
6      };
7      auto reduce = [=] __cuda_callable__ ( const Real& a, const Real& b ) -> Real {
8          return a + b;
9      };
10     auto keep = [=] __cuda_callable__ ( Index row, const Real& value ) mutable {
11        outVector[ row ] = value;
12     };
13     this->segments.reduceAllSegments( fetch, reduce, keep, 0.0 );
14 }
```

Na řádku 13 se volá metoda `reduceAllSegments`, která provádí redukci v jednotlivých segmentech. Opět využívá lambda funkci `fetch`, která zodpovídá za načítání dat. Dostává globální index ukazující do polí `values` s hodnotami maticových prvků a pole `columnIndexes`, které obsahuje sloupcové indexy nenulových prvků. Za pomocí hodnot z těchto polí provede násobení maticového prvku s příslušným prvkem vstupního vektoru. Obdržený mezivýsledek je předán funkci `reduce`, která provede sečtení jednotlivých součinů. Pro každý maticový řádek tak dostaneme výsledek pronásobením se vstupním vektorem, který je potřeba uložit do výstupního vektoru. To obstarává lambda funkce `keep`.

Tento přístup umožňuje knihovně TNL spravovat jen jednu třídu pro implementaci řídkých matic. Použitý formát je vyjádřen pomocí šablonového parametru udávající typ použitých segmentů potažmo formátu pro ukládání řídkých matic. Díky tomu jsou v této knihovně implementovány i specializace pro binární a symetrické matice.

$$M = \begin{array}{|c|c|c|c|}\hline 11 & 12 & & \\ \hline & 23 & 24 & \\ \hline 35 & & & 36 \\ \hline & 47 & 48 & 49 \\ \hline \end{array}$$



Obrázek 2: Tento příklad demonstruje vztah mezi *segmenty* a maticí M reprezentovanou pomocí CSR formátu. *Segmenty* (zobrazené šedivou barvou) poskytují přístup do polí pro hodnoty maticových prvků a jejich sloupcové indexy na základě znalosti indexu řádku a pořadí nenulového prvku v rámci maticového řádku. V řeči segmentů jde o přepočet indexu segmentu a lokálního segmentu na globální index ukazující právě do dvou zmíněných polí.

Jak bylo již zmíněno, segmenty lze využít také pro implementaci nestrukturovaných numerických sítí. Toto téma je náplní následující části.

3 Nestrukturované numerické sítě

Nestrukturované numerické sítě jsou základem zejména pro metody konečných prvků a konečných objemů. Jsou důležité pro popis komplexních geometrii bez nichž by nebylo možné řešit reálné úlohy například z aplikací v průmyslu. Různé numerické metody vyžadují různý přístup k nestrukturovaným numerickým sítím a ukládání různých informací. Přitom ukládání zbytečných údajů může výrazně navýšit paměťové nároky pro uložení sítě. Paměť ja ale zejména na GPU stále relativně omezená. V knihovně TNL využíváme možností šablonových specializací za účelem vytvoření staticky konfigurovatelné sítě, kterou lze nastavit přesně podle potřeb dané numerické metody. TNL dokonce nabízí ukládání nestrukturovaných numerických sítí libovolné dimenze.

Celá síť $\mathcal{M}$ se skládá z tzv. entit. Například pro trojrozměrnou síť jde o buňky, stěny, hrany a vrcholy. Dále budeme používat tzv. subentity a superentity.

Definition 3.1 (subentita)**.** Entita $E_1 \in \mathcal{M}$ je subentitou entity $E_2 \in \mathcal{M}$, pokud platí, že $E_1 \subset E_2$, tj. dimenze entity E_1 je menší než dimenze entity E_2.

Definition 3.2 (superentita)**.** Entita $E_1 \in \mathcal{M}$ je superentitou entity $E_2 \in \mathcal{M}$, pokud platí, že $E_2 \subset E_1$, tj. dimenze entity E_1 je větší než dimenze entity E_2.

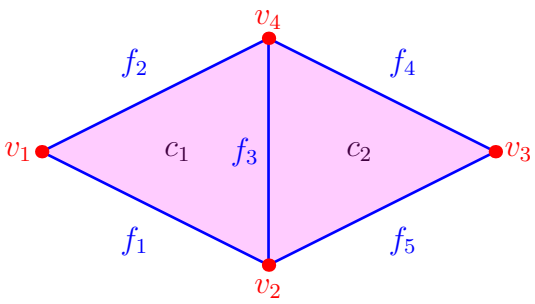
Omezujeme se na tzv. nestrukturované konformní silně homogenní sítě. Přesněji tyto sítě popisují následující definice.

Definition 3.3 (konformní síť)**.** Nechť $\mathcal{M}$ je síť. Pokud pro každé dvě entity $E_1, E_2 \in \mathcal{M}$ platí, že průnik jejich uzávěrů $\overline{E}_1 \cap \overline{E}_2$ je buď prázdná množina nebo síťová entita, pak $\mathcal{M}$ nazýváme *konformní* sítí.

Definition 3.4 (nestrukturovaná síť)**.** Síť se nazývá *nestrukturovaná*, pokud každý vrchol může být vrcholem nekonstantního počtu buněk.

Definition 3.5 (homogenní síť)**.** Pokud všechny buňky sítě, tj. entity nejvyšší dimenze, mají stejný tvar, pak říkáme, že síť je *slabě homogenní*. Pokud mají všechny entity stejné dimenze stejný tvar, říkáme, že síť je *silně homogenní*.

Definice nestrukturované sítě vlastně říká, že počet superentit libovolné entity této sítě není obecně konstantní neboť závisí na počtu sousedních entit. Na druhou stranu, počet subentit závisí pouze na tvaru dané entity. V případě silně homogenních sítí je počet subentit vždy konstantní.



Obrázek 3: Příklad dvojrozměrné sítě skládající se ze dvou buněk (trojúhelníků) c_1, c_2, stěn (nebo hran) f_1, f_2, f_3, f_4, f_5 a vrcholů v_1, v_2, v_3, v_4.

Obrázek 3 ukazuje příklad jednoduché dvojrozměrné sítě skládající se ze dvou trojúhelníků, pěti stěn a čtyř vrcholů.

Síť je jednoznačně určena souřadnicemi svých vrcholů a definicemi svých buněk. Každá buňka je definována výčtem svých vrcholů. Známe-li tvar buněk sítě, lze ostatní informace, tj. vztahy mezi subentitami a superentitami, již dopočítat. Vztahy mezi entitami různých dimenzí popisují incidenční matice na obrázku 4

Vidíme, že k vyjádření vztahů mezi entitami různých dimenzí, je potřeba uložit poměrné velký počet matic. S rostoucí dimenzí sítě tento počet narůstá ještě výrazněji. Datová struktura pro numerické sítě v TNL umožňuje nakonfigurovat, které incidenční matice se budou ukládat. Jednotlivé incidenční matice jsou vyjádřeny buď pomocí segmentů nebo jako binární matice. Díky této specializaci implementované v knihovně TNL lze redukovat paměťové požadavky pro uložení sítě v paměti GPU. Návrh této datové struktury je schématicky zobrazen na obrázcích 5 a 6.

V závislosti na dimenzi buněk sítě a také podle použité konfigurace `Config`, se vytvoří několik instancí třídy `StorageLayer`. Ty ukládají jednak entity dané dimenze, ale v závislosti na konfiguraci také subentity a superentity k daným entitám.

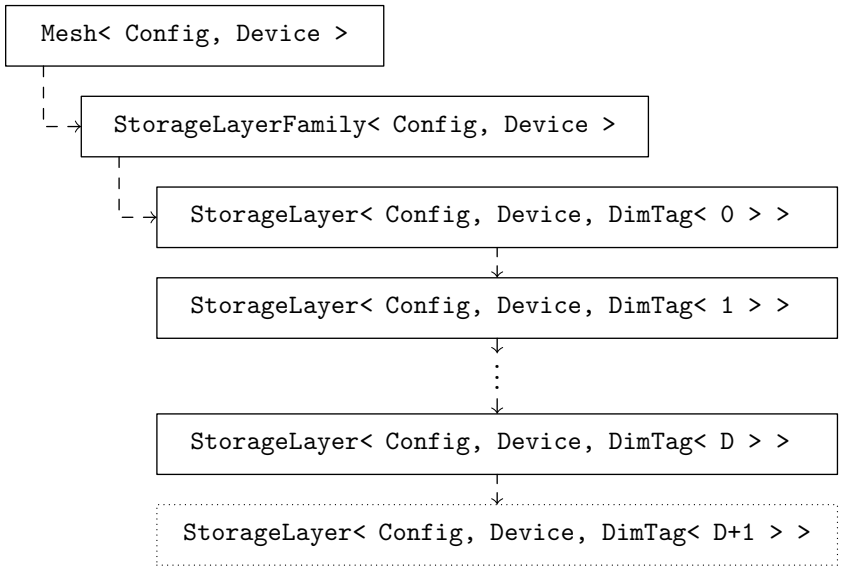
Implementace nestrukturované numerické sítě v knihovně TNL také podporuje distribuované výpočty na klastrech a klastrech vybavených GPU kartami za pomocí rozhraní MPI.

$$I_{0,1} = \left(\begin{array}{c|ccccc} & f_1 & f_2 & f_3 & f_4 & f_5 \\ \hline v_1 & 1 & 1 & & & \\ v_2 & 1 & & 1 & & 1 \\ v_3 & & & & 1 & 1 \\ v_4 & & 1 & 1 & 1 & \end{array}\right) \qquad I_{1,0} = \left(\begin{array}{c|cccc} & v_1 & v_2 & v_3 & v_4 \\ \hline f_1 & 1 & 1 & & \\ f_2 & 1 & & & 1 \\ f_3 & & 1 & & 1 \\ f_4 & & & 1 & 1 \\ f_5 & & 1 & 1 & \end{array}\right)$$

$$I_{0,2} = \left(\begin{array}{c|cc} & c_1 & c_2 \\ \hline v_1 & 1 & \\ v_2 & 1 & 1 \\ v_3 & & 1 \\ v_4 & 1 & 1 \end{array}\right) \qquad I_{2,0} = \left(\begin{array}{c|cccc} & v_1 & v_2 & v_3 & v_4 \\ \hline c_1 & 1 & 1 & & 1 \\ c_2 & & 1 & 1 & 1 \end{array}\right)$$

$$I_{1,2} = \left(\begin{array}{c|cc} & c_1 & c_2 \\ \hline f_1 & 1 & \\ f_2 & 1 & \\ f_3 & 1 & 1 \\ f_4 & & 1 \\ f_5 & & 1 \end{array}\right) \qquad I_{2,1} = \left(\begin{array}{c|ccccc} & f_1 & f_2 & f_3 & f_4 & f_5 \\ \hline c_1 & 1 & 1 & 1 & & \\ c_2 & & & 1 & 1 & 1 \end{array}\right)$$

Obrázek 4: Incidenční matice reprezentující vztahy mezi entitami sítě na obrázku 3.

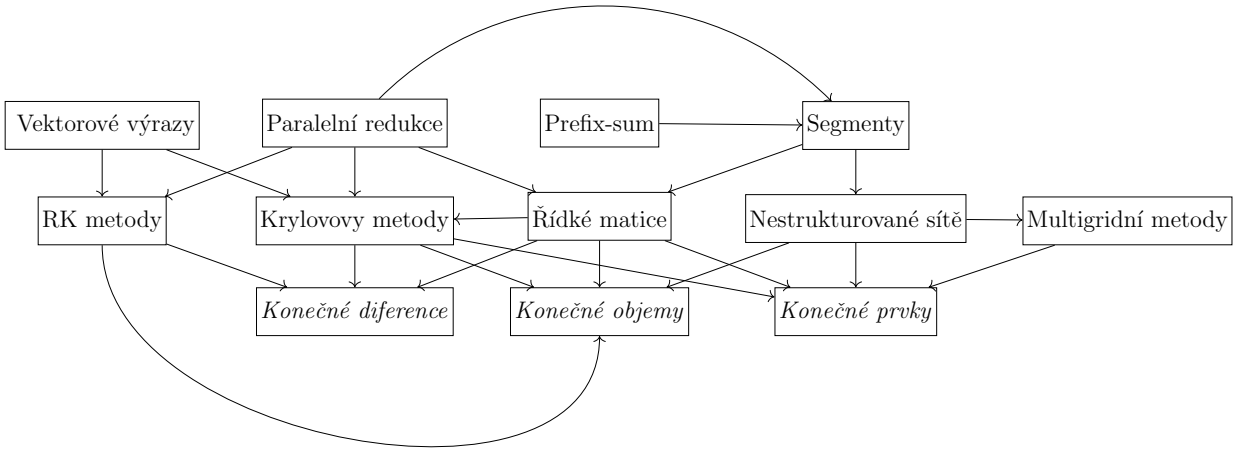


Obrázek 5: Základ návrhu datové struktury pro ukládání nestrukturovaných numerických sítí. Podle dimenze síťových buněk je vygenerováno několik instancí třídy `StorageLayer`, které ukládají entity daných dimenzí.

Obrázek 6: Detail třídy `StorageLayer`. Tato třída ukládá nejen entity dané dimenze (`StorageLayer`), ale v závislosti na konfiguraci sítě (`Config`) také zvolené subentity (`SubentityStorageLayer`) a superentity (`SuperentityStorageLayer`).

4 Řešení parciálních diferenciálních rovnic na GPU

V této části se budeme zabývat vývojem paralelních algoritmů pro řešení různých typů parciálních diferenciálních rovnic. Obrázek 7 ukazuje schématicky využití jednotlivých algoritmů a datových struktur, které jsme představili v předchozím textu. Základem



Obrázek 7: Tento obrázek ukazuje vztahy mezi jednotlivými algoritmy a datovými strukturami používanými při vývoji paralelních řešičů parciálních diferenciálních rovnic.

jsou operace jako vektorové výrazy a paralelní redukce, které se využívají hlavně k implementaci Rungových-Kuttových metod pro řešení systémů obyčejných diferenciálních rovnic, které vznikají při explicitním řešení časově závislých parciálních diferenciálních rovnic. Stejně tak i metody Krylovových podprostorů využívají vektorové výrazy a paralelní redukci pro výpočet skalárních součinů. Operace prefix-sum je obdobou redukce. Napočítává např. částečné sumy pro zadanou vstupní posloupnost. Tato operace je užitečná pro efektivní inicializaci formátů pro řídké matice potažmo struktury zvané *segmenty* v knihovně TNL. Na základě segmentů jsou pak postaveny řídké matice, které jsou základem pro semi-implicitní metody řešení časově závislých nebo lineárních eliptických parciálních diferenciálních rovnic. Segmenty jsou také základem, na kterém stojí datová struktura pro nestrukturované numerické sítě, jež jsou nezbytné pro metodu konečných prvků a konečných objemů. S nestrukturovanými sítěmi souvisí i multigridní metody, které využívají sadu numerických sítí s různě jemným rozlišením pro urychlení konvergence při řešení lineárních soustav.

V následující části předvedeme paralelní řešiče na řešení některých typů parciálních diferenciálních rovnic na GPU.

4.1 Vývoj ploch a křivek podle křivosti

Mnoho úloh z fyziky a aplikované matematiky se zabývá modelováním tvarů. Jmenovat můžeme například fázové přechody jako tání nebo tuhnutí [28], růst krystalů [29], šíření lesních požárů [30], vícefázové proudění [31], segmentace obrazových dat [32] nebo simulace 3D tisku [33]. Z matematického pohledu se zabýváme evolucí jedné nebo více variet v $\mathbb{R}^n$, které ovšem nemusí být všude hladké nebo diferencovatelné. Navíc může docházet k topologickým změnám jako je spojení více variet do jedné nebo naopak rozdělení jedné variety na několik separovaných variet. Právě tyto topologické změny tvoří největší překážky při numerickém řešení těchto úloh. Z tohoto důvodu vzniklo několik různých přístupů a formulací. Lze je rozdělit do tří základních skupin:

1. *Grafová formulace* je nejjednodušší a ve skutečnosti tento přístup neumožňuje žádné topologické změny. V řadě úloh je ale tato formulace více než dostačující. V tomto případě je modelovaná křivka nebo plocha popsána jako

$$\Gamma \equiv \{(\mathbf{x}, u(\mathbf{x})) \in \mathbb{R}^n \mid \mathbf{x} \in \Omega\}.$$

2. *Lagrangeovská (parametrická) formulace* je založena na parametrizaci modelované křivky nebo plochy. Jde o velice efektivní přístup, který umožňuje provádět topologické změny ovšem za cenu výrazně komplikovaných algoritmů zejména pokud jde o topologické změny u ploch. Pro svou efektivitu je však s oblibou používán zejména například v úlohách, kde k topologickým změnám nedochází. V tomto případě je křivka ($m = 1$) nebo plocha ($m = 2$) popsána jako

$$\Gamma \equiv \{u(\mathbf{x}) \in \mathbb{R}^n \mid \mathbf{x} \in <0, 1>^m\}.$$

3. *Implicitní formulace (vrstevnicová metoda a metoda fázového pole)* je navržena zejména za účelem snadného modelování topologických změn. Využívá se implicitní popis variety, který vyžaduje navýšení dimenze úlohy a tím a zvýšení výpočetní náročnosti. V implicitním popisu jsou však topologické změny ošetřeny automaticky a tím se tyto metody stávají robustnějšími. V tomto případě je modelovaná křivka nebo plocha popsána jako

$$\Gamma \equiv \{\mathbf{x} \in \Omega \mid u(\mathbf{x}) = 0\}$$

Mezi základní úlohy při modelování evoluce křivek a ploch patří vývoj podle střední křivosti. Tuto úlohu je možné odvodit pomocí minimalizace délky křivky nebo obsahu plochy, obecně funkcionálu

$$\min_{\Gamma} \int_{\Gamma} 1 \mathrm{d}S.$$

Eulerovy-Lagrangeovy rovnice spolu s metodou gradientního sestupu vedou na rovnici

$$\partial_t \varphi = -Q\nabla \cdot \left(\frac{\nabla \varphi}{Q}\right) + F(t) \text{ na } \Omega \times (0,T], \tag{2}$$

$$\varphi\,|_{t=0} = \varphi_{ini} \text{ na } \Omega, \tag{3}$$

$$\varphi = g \text{ na } \partial\Omega, \tag{4}$$

kde

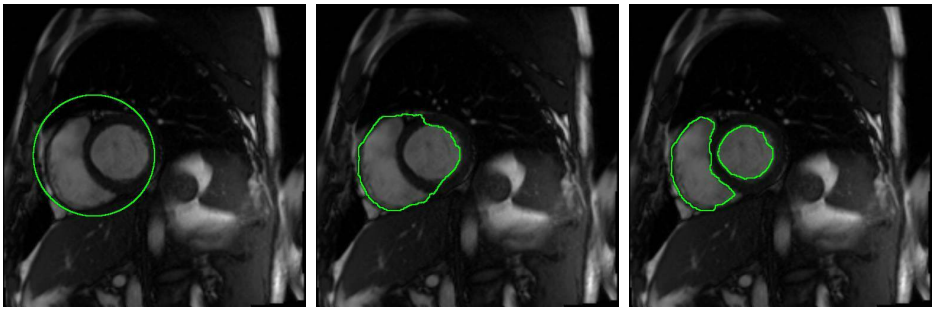
- $Q = \sqrt{1 + |\nabla \varphi|^2}$ pro grafovou formulaci,
- $Q = \sqrt{\epsilon + |\nabla \varphi|^2}$ pro vrstevnicovou formulaci.

Rovnice 3 minimalizuje délku křivky nebo obsah modelované plochy a lze pro ní dokázat následující energetickou rovnost [34]

$$\int_\Omega \frac{(\partial_t u)^2}{Q} \mathrm{d}x + \underbrace{\frac{1}{2}\frac{\mathrm{d}}{\mathrm{d}t}\int_\Omega Q \mathrm{d}x}_{\leq 0} = 0, \tag{5}$$

která potvrzuje, že časová derivace plochy grafu funkce je nekladná.

Využívá se například při modelování fázových přechodů. Má silný vyhlazovací účinek, což je užitečné při segmentování obrazových dat obsahujících šum. Obrázek 8 ukazuje aplikaci vrstevnicové metody pro segmentaci dat z magnetické rezonance.



Obrázek 8: Aplikace vrstevnicové metody pro vývoj křivek podle křivosti na segmentaci dat z magnetické rezonance.

Tuto úlohu lze numericky aproximovat pomocí metody konečných diferencí kombinovanou buď s některou Rungovou-Kuttovou metodou v případě explicitní časové diskretizace nebo vhodnou metodou Krylovových podprostorů v případě semi-implicitní časové diskretizace. Při porovnání rychlosti výpočtu na grafické kartě na GeForce GTX 480 a procesoru AMD Opteron 6172 s 12 jádry bylo dosaženo urychlení až **10x** [35].

4.2 Anizotropní Willmorův tok

Jako další příklad zmíníme řešení úlohy anizotropního Willmorova toku. Za tím účelem nejprve zavedeme pojem anizotropie.

Definition 4.1. Říkáme, že funkce γ je pozitivně homogenní s řádem jedna, právě když platí

$$\gamma\left(\lambda\mathbf{P}\right)=\lambda\gamma\left(\mathbf{P}\right)\ \text{pro}\ \mathbf{P}\in\mathbb{R}^{n}\setminus\left\{0\right\},\lambda>0.$$

Definition 4.2. Přípustná anizotropní funkce (admissible anisotropy function) je funkce $\gamma:\mathbb{R}^{n}\setminus\left\{0\right\}\rightarrow\mathbb{R}^{+}$, $\gamma\in C^{3}\left(\mathbb{R}^{n+1}\setminus\left\{0\right\}\right)$, která je pozitivně homogenní s řádem jedna a je konvexní v tom smyslu, že existuje konstanta $c_{0}>0$ taková, že

$$\mathbf{q}^{T}D^{2}\left(\gamma\left(\mathbf{p}\right)\right)\mathbf{q}\geq c_{0}\left\|\mathbf{q}\right\|^{2}\ \text{pro všechna}\ \mathbf{p},\mathbf{q}\in\mathbb{R}^{n}\ \text{s}\ \mathbf{p}\cdot\mathbf{q}=0,\left\|\mathbf{p}\right\|=1. \tag{6}$$

Anizotropní funkce $\gamma=\gamma(\mathbf{p})$ umožňuje definovat anizotropní křivost H_{γ} jako

$$H_{\gamma}=\nabla\cdot\left(\nabla_{\mathbf{p}}\gamma(\nabla\varphi,-1)\right), \tag{7}$$

Na základě takto definované anizotropní křivosti lze zavést Willmorův funkcionál $\mathcal{W}_{\gamma}$

$$\mathcal{W}_{\gamma}(\Gamma(t))=\frac{1}{2}\int_{\Gamma(t)}H_{\gamma}^{2}dS. \tag{8}$$

Willmorův tok (Willmore flow) je gradientní tok (gradient flow) pro minimalizaci funkcionálu (8). Jde o úlohu čtvrtého řádu tvaru

$$\begin{aligned}
\partial_{t}u &= -Q\nabla\cdot\left(\mathbb{E}_{\gamma}\nabla w_{\gamma}-\frac{1}{2}\frac{w_{\gamma}^{2}}{Q^{3}}\nabla u\right)\ \text{na}\ (0,T)\times\Omega, && (9)\\
w_{\gamma} &= QH_{\gamma}\ \text{na}\ (0,T)\times\Omega, && (10)\\
u\mid_{t=0} &= u_{0}\ \text{na}\ \Omega, && (11)\\
u &= g_{1},\ w_{\gamma}=g_{2}\quad \text{na}\ \partial\Omega, && (12)
\end{aligned}$$

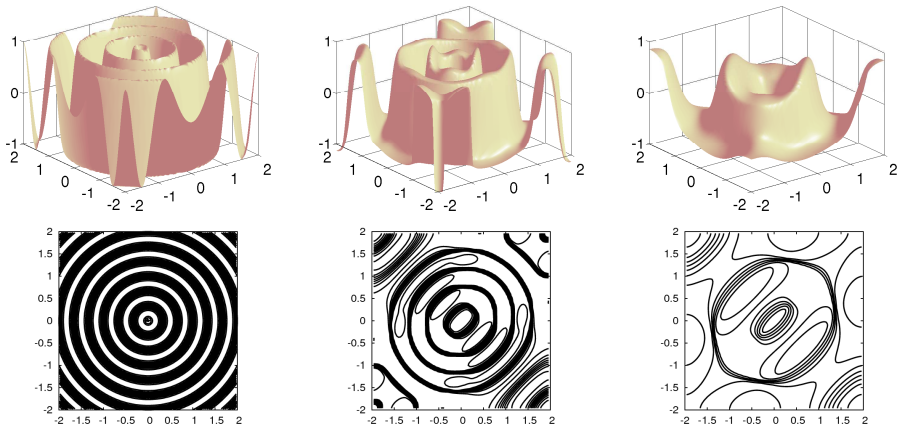
kde $\mathbb{E}_{\gamma}\left(u\right)_{ij}:=\partial p_{i}\partial p_{j}\gamma\left(\nabla u,-1\right)$ je tenzor anizotropní projekce do tangenciálního prostoru. Pro tuto úlohu autor tohoto textu ukázal energetickou rovnost [36] podobnou rovnosti (5)

$$\int_{\Omega}\frac{\left(\partial_{t}u\right)^{2}}{Q}\mathrm{d}x+\underbrace{\frac{1}{2}\frac{\mathrm{d}}{\mathrm{d}t}\int_{\Omega}H_{\gamma}^{2}Q\mathrm{d}x}_{\leq0}=0,$$

která opět potrvzuje, že časová derivace Willmorovy energie (8) je nekladná.

Právě fakt, že jde o silně nelineární úlohu čtvrtého řádu, klade velké nároky na

výpočetní výkon, neboť pro numerickou aproximaci je nutné volit velmi malé časové kroky. Právě z důvodů výrazné nelinearity rovnice (10) se ukazuje jako výhodnější diskretizace metodou komplementárních objemů v porovnání s čistou metodou konečných diferencí [37, 36]. Obrázek 9 demonstruje vývoj plochy pravě podle anizotropního Willmorova toku.



Obrázek 9: Ukázka vývoje plochy pomocí grafové formulace Willmorova toku.

Při výpočtech na grafické kartě GeForce GTX 280 a procesoru Intel Core 2 Quad se 4 jádry bylo dosaženo urychlení až **6x** [36].

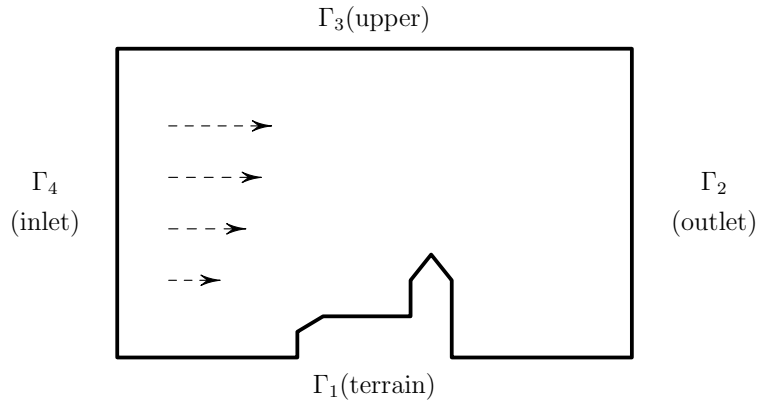
4.3 Navierovy-Stokesovy rovnice pro nestlačitelné proudění

Simulace nestlačitelného proudění pomocí Navierových-Stokesových rovnic patří mezi jedny z nejdůležitějších úloh s mnoha reálnými aplikacemi nejen v průmyslu. Jde o typický příklad výpočetně velmi náročné úlohy. Bez pochyby lze říci, že výpočetní dynamika tekutin (computational fluid dynamics, CFD) je úzce svázaná s doménou vysoce výkonných výpočtů (high-performance computing, HPC). V této části ukážeme paralelizaci vysoce efektivního multigridního řešiče pro GPU.

Uvažujeme zjednodušenou úlohu ve dvou dimenzích následujícího tvaru

$$\begin{aligned}
\mathbf{u}_t + \mathbf{u}\cdot\nabla\mathbf{u} - \nu\Delta\mathbf{u} + \nabla p &= \mathbf{0} \text{ na } \Omega, && (13)\\
\nabla\mathbf{u} &= 0 \text{ na } \Omega, && (14)\\
\mathbf{u}(0,\cdot) &= \mathbf{u}_0 \text{ na } \Omega, && (15)\\
\mathbf{u} &= \mathbf{0} \text{ na } \Gamma_{\text{terrain}}, && (16)\\
u_x = u_{in}, u_y &= 0 \text{ na } \Gamma_{\text{inlet}}, && (17)\\
-p\mathbf{n} + \nu(\nabla\mathbf{u})\cdot\mathbf{n} &= \mathbf{0} \text{ na } \Gamma_{\text{outlet}}, && (18)\\
(\nu(\nabla\mathbf{u})\cdot\mathbf{n})_x = 0, u_y &= 0 \text{ na } \Gamma_{\text{upper}}. && (19)
\end{aligned}$$

kde $\mathbf{u}$ je rychlostní pole, p je tlak a ν vazkost. Význam okrajových podmínek (17)-(19) objasňuje obrázek 10. Jde o zjednodušenou úlohu výpočtu proudění vzduchu nad městskou zástavbou.



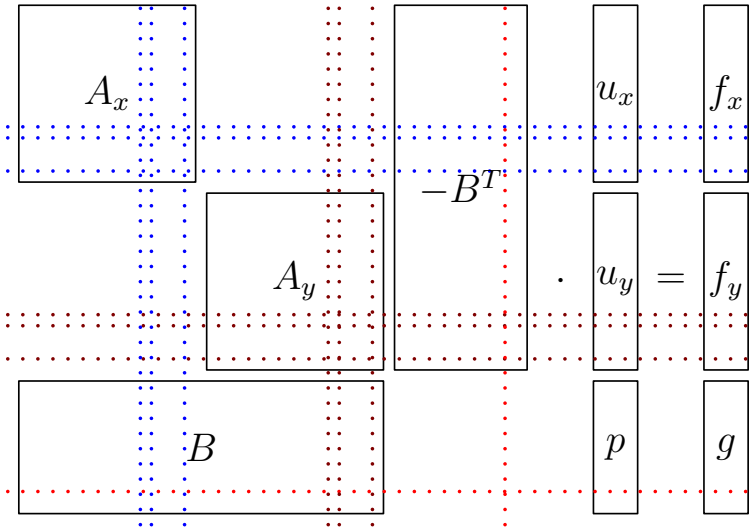
Obrázek 10: Schéma úlohy řešení nestlačitelného proudění pomocí Navierových-Stokesových rovnic.

Diskretizace pomocí Oseenova schématu a Crouzeixových-Raviartových konečných prvků [38] vede na indefinitní systém lineárních rovnic tvaru

$$\begin{pmatrix} \tilde{\mathbf{A}}_x(\tilde{\mathbf{u}}^{k-1}) & \mathbf{0} & -\tilde{\mathbf{B}}_x \\ \mathbf{0} & \tilde{\mathbf{A}}_y(\tilde{\mathbf{u}}^{k-1}) & -\tilde{\mathbf{B}}_y \\ \tilde{\mathbf{B}}_x^T & \tilde{\mathbf{B}}_y^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \tilde{\mathbf{u}}_x^k \\ \tilde{\mathbf{u}}_y^k \\ \tilde{\mathbf{p}}^k \end{pmatrix} = \begin{pmatrix} \tilde{\mathbf{f}}_x(\tilde{\mathbf{u}}^k) \\ \tilde{\mathbf{f}}_y(\tilde{\mathbf{u}}^k) \\ \tilde{\mathbf{g}} \end{pmatrix}, \qquad (20)$$

kde $\tilde{\mathbf{A}}_x$, $\tilde{\mathbf{A}}_y$ obsahuje advekční a viskozní členy, $\tilde{\mathbf{B}}_x$, $\tilde{\mathbf{B}}_y$ obsahuje divergenční členy, $\tilde{\mathbf{u}}^k = (\tilde{\mathbf{u}}_x^k, \tilde{\mathbf{u}}_y^k)$ je vektor reprezentující aproximaci rychlostního pole $\mathbf{u}^k$ a $\tilde{\mathbf{f}}_x(\tilde{\mathbf{u}}^k)$, $\tilde{\mathbf{f}}_y(\tilde{\mathbf{u}}^k)$ s $\tilde{\mathbf{g}}$ zastupují pravou stranu v Oseenově schématu. Z důvodu indefinitnosti je tento systém těžko řešitelný za pomocí samotných metod Krylovových podprostorů. Systém proto řešíme multigridním řešičem Vankova typu. Ten vychází z Jacobiho iterační metody, jako

většina multigridních metod. V tomto případě jde však o blokovou variantu Jacobiho metody. Iteruje se přes jednotlivé elementy sítě a řeší se malé lokální lineární systémy svázané jen s proměnnými pro daný element. Za tím účelem se lineární systém (20) nejprve transformuje do podoby, kde vystupují lokální systémy pro jednotlivé elementy. Tuto extrakci demonstruje obrázek 11.

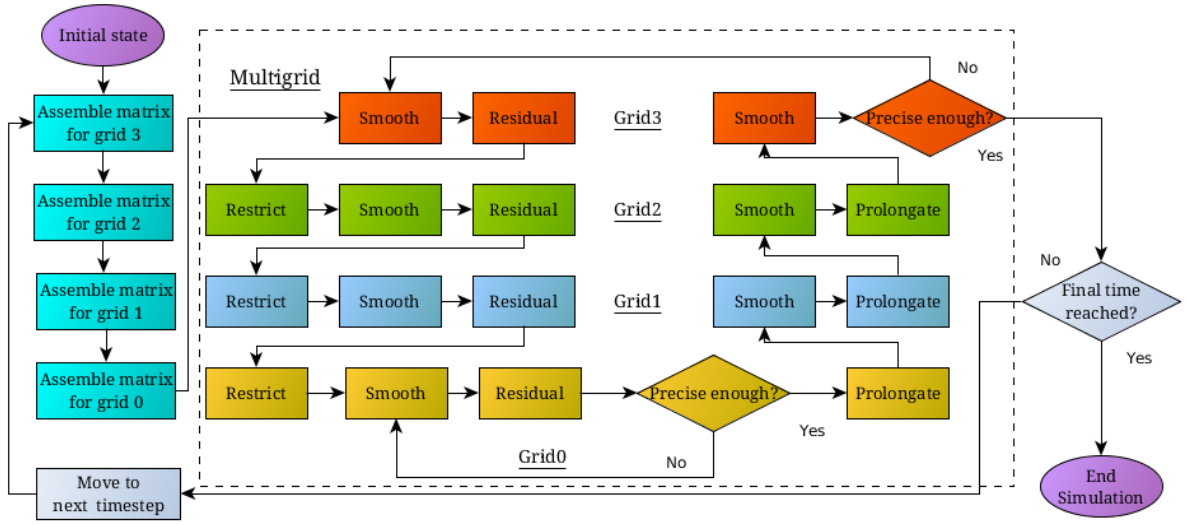


Obrázek 11: Extrakce lokálních lineárních systémů pro Vankův řešič.

Výsledné lokální systémy mají stejnou strukturu jako globální systém (20), skládají se ale ze dvou diagonálních bloků o rozměrech 3×3 pro x-ovou a y-ovou složku a dále ze dvou bloků o rozměrech 3×1, které reprezentují divergenční členy. Celý blok vypadá takto

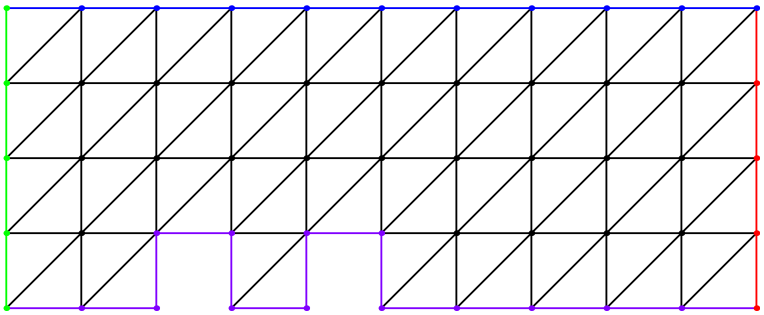
$$\begin{pmatrix} \mathbf{A}_x^K & \mathbf{0} & -\mathbf{b}_x^K \\ \mathbf{0} & \mathbf{A}_y^K & -\mathbf{b}_y^K \\ (\mathbf{b}_x^K)^T & (\mathbf{b}_y^K)^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{u}_x^K \\ \mathbf{u}_y^K \\ p^K \end{pmatrix} = \begin{pmatrix} \mathbf{f}_x^K \\ \mathbf{f}_y^K \\ g^K \end{pmatrix}. \tag{21}$$

Pro každý element numerické sítě je potřeba v rámci blokové Jacobiho metody vyřešit příslušný lokální systém. To se provádí pomocí Schurova doplňku a adjungovaných matic. Výhoda tohoto přístupu je v tom, že tyto lokální systémy lze efektivně řešit na jednotlivých multipocesorech GPU s využitím rychlých sdílených pamětí. Eliminuje se tak úzké hrdlo v podobě paměťového subsytému a lépe se tak využívá výpočetní potenciál GPU. Tento multigridní řešič pak lze počítat kompletně na GPU [39], jak ukazuje obrázek 12



Obrázek 12: Schéma multigridního řešiče pro řešení nestlačitelných Navierových-Stokesových rovnic.

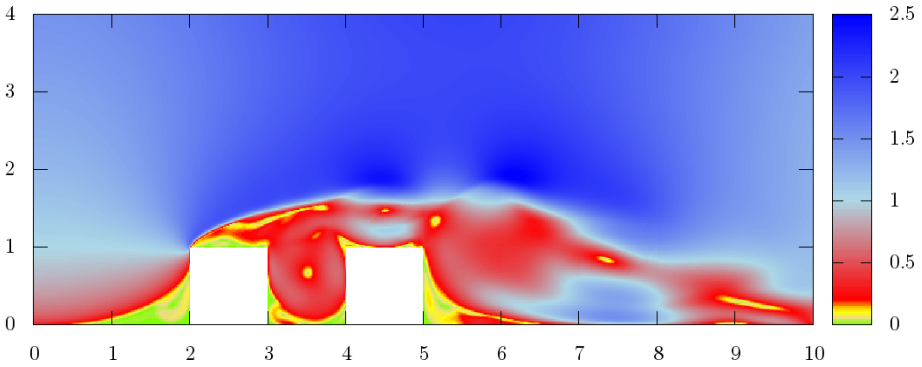
Při simulacích na grafické kartě Nvidia Tesla K40 a procesoru Intel i7-3770 se 4 jádry bylo dosaženo urychlení až **11x**. Výsledné proudové pole spolu se základní numerickou sítí je zobrazeno na obrázku 14.



Obrázek 13: Numerické síť použitá pro simulaci nestlačitelného proudění ve 2D.

4.4 Vícefázové proudění v porézním prostředí

Na závěr této části se budeme věnovat řešení vícefázového proudění v porézním prostředí. Tuto úlohu lze matematicky formulovat jako systém parciálních diferenciálních rovnic tvaru



Obrázek 14: Výsledné proudové pole ze simulace nestlačitelného proudění ve 2D.

$$\sum_{j=1}^{n} N_{i,j}\partial_t Z_j + \sum_{j=1}^{n} \mathbf{u}_{i,j}\cdot\nabla Z_j + \sum_{j=1}^{n} r_{i,j}Z_j +$$

$$\nabla\cdot\left(m_i\underbrace{\left(-\sum_{j=1}^{n} D_{i,j}\nabla Z_j + \mathbf{w}_i\right)}_{\mathbf{v}_i} + \sum_{j=1}^{n} Z_j\mathbf{a}_{i,j}\right) = f_i \text{ na } \Omega\times(0,T) \tag{22}$$

$$\mathbf{Z}\,|_{t=0} = \mathbf{Z}_{ini},\ \text{na } \Omega, \tag{23}$$

$$\begin{aligned} Z_j &= \mathbf{Z}_j^D,\ \text{na } \Gamma_j^D \ pro\ j\in\hat{n},\\ \mathbf{v}_i\cdot\mathbf{n}_{\partial\Omega} &= \mathbf{v}_i^N,\ \text{na } \Gamma_j^N \text{ pro } i\in\hat{n}, \end{aligned} \tag{24}$$

pro $i = 1,\ldots,n$, kde $\mathbf{Z} = (Z_1,\ldots,Z_n)^T$ je vektor neznámých funkcí $Z_j = Z_j(t,\mathbf{x})$, $\mathbf{v}_i$ je rychlostní člen daný jako

$$\mathbf{v}_i = -\sum_{j=1}^{n}\mathbf{D}_{i,j}\nabla Z_j + \mathbf{w}_j, \tag{25}$$

a okraje $\partial\Omega$ oblasti Ω splňují

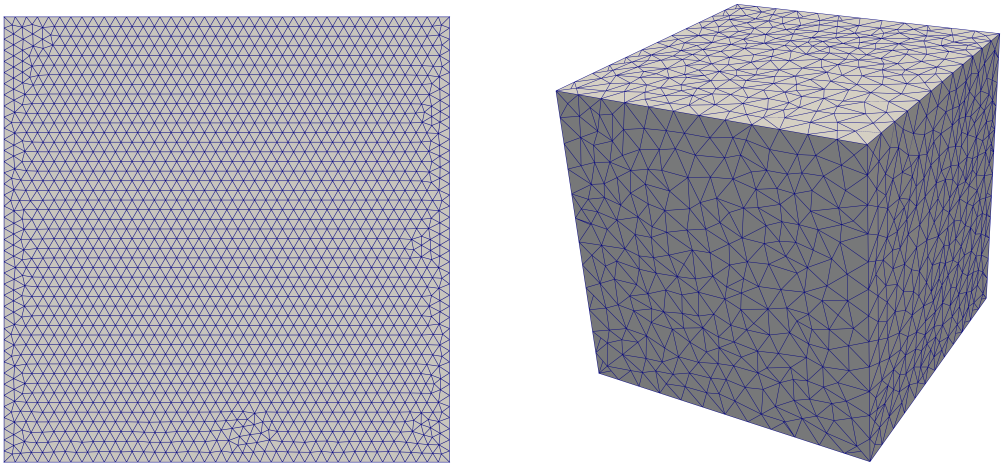
$$\begin{aligned} \Gamma_j^D\cup\Gamma_j^N &\equiv \partial\Omega\ pro\ j = 1,\ldots,n,\\ \Gamma_j^D\cap\Gamma_j^N &\equiv \emptyset\ pro\ j = 1,\ldots,n. \end{aligned}$$

Všechny ostatní koeficienty $N_{i,j}$, $u_{i,j}$, m_i, $D_{i,j}$, w_i, $a_{i,j}$, $r_{i,j}$ a f_i jsou dané pro všechna $i,j = 1,\ldots,n$.

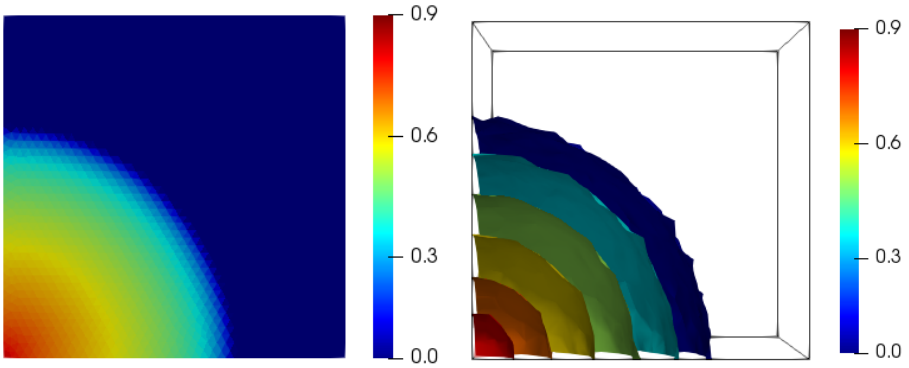
Systém (22)-(24) lze diskretizovat za pomoci metody smíšených konečných prvků [40]. V publikaci [41] jsou použity Raviartovy-Thomasovy-Nédélecovy konečné prvky. Člen $\mathbf{v}_i$ ve výrazu (22) způsobuje, že výsledný lineární systém je opět indefinitní. Tentokrát je k jeho řešení použita metoda hybridizace [40]. Její hlavní myšlenka spočívá ve využití

metody dekompozice oblasti (domain decompostion method) k transformaci proměnných ukládaných na konečných prvcích na jinou veličinu uloženou na hranici konečného prvku. Tím se podobně jako u Vankova řešiče dojde k menším lokálním systémům, které se řeší na hranicích jednotlivých elementů. Následně je použita vhodná metoda Krylovových podprostorů, konkrétně GMRES metoda. Metodu hybridizace tak lze v tomto kontextu chápat i jako jistý druh předpodmínění. Odvozený řešič byl implementovaný s pomocí knihovny TNL a může běžet buď kompletně na vícejádrovém procesoru nebo na GPU.

Výsledky obdržené při řešení McWhorterovy-Sunadovy úlohy ve 2D a ve 3D [41] jsou zobrazeny na obrázku 15 (použité nestrukturované numerické sítě) a 16 (výsledek simulace).



Obrázek 15: Nestrukturované numerické sítě použité pro řešení McWhorterovy-Sunadovy úlohy ve 2D a ve 3D.



Obrázek 16: Výsledné koncentrace injektované látky při simulaci McWhorterovy-Sunadovy úlohy ve 2D a ve 3D.

Při výpočtech na grafické kartě Nvidia Tesla K40 a procesoru Intel i7-5820K se 6 jádry bylo dosaženo urychlení až **7.5x**.

5 Shrnutí a závěr

V této přednášce jsme se snažili zachytit širší aspekty vývoje paralelních řešičů parciálních diferenciálních rovnic pro běh na GPU. Ukázali jsme výhody architektury GPU, která ze své podstaty nabízí lepší škálovatelnost v porovnání s běžnými procesory, a tak lze v budoucnu očekávat stále zvětšující se rozdíl mezi výkonem procesorů a GPU. Vývoj algoritmů pro GPU ale není v žádném případě jednoduchý a vyžaduje podrobnou znalost této architektury. To je může činit těžko přístupným např. pro komunitu numerických matematiků. Ukázali jsme, že funkcionální knihovny jako je Blas nebo Cublas nemusí být nejlepším řešením a nastínili jsme možnosti využití moderních vlastností jazyka C++.

Dále jsme představili knihovnu TNL, která se snaží profitovat právě z vlastností nových standardů jazyka C++. Využívá techniky jako šablonové specializace, výrazové šablony nebo lambda funkce k tomu, aby nabídla jednotné rozhraní pro vývoj paralelních algoritmů. V některých případech je pak možné napsat jeden kód, který může běžet jak na vícejádrových procesorech tak i na GPU. Představili jsme abstrakci formátů pro ukládání řídkých matic zvanou *segmenty*, na základě kterých TNL implementuje robustní datovou strukturu pro práci s řídkými maticemi ale také datovou strukturu pro ukládání a operace s nestrukturovanými numerickými sítěmi.

Za pomoci zmíněných algoritmů a datových struktur pak lze vyvíjet paralelní řešiče pro numerickou aproximaci parciálních diferenciálních rovnic. Ukázali jsme řešiče pro různé typy těchto rovnic. Jednak šlo o modelování evoluce křivek a ploch a to konkrétněji o vývoj podle střední křivosti s aplikací na segmentaci medicínských dat. Dále jsme předvedli paralelní řešič pro aproximaci anizotropního Willmorova toku. Nakonec jsme se zabývali efektivním řešením indefinitních lineárních systému, které vznikají například pro řešení nestlačitelných Navierových-Stokesových rovnic nebo vícefázového proudění v porézním prostředí. Na těchto úlohách jsme demonstrovali paralelizaci Vankova multigridního řešiče a hybridní metodu smíšených konečných prvků. U zmíněných úloh se paralelizací na GPU podařilo dosáhnout urychlení 6 krát až 11 krát v porovnání s výpočty na procesorech.

Literatura

1. NVIDIA. *Cublas*. [N.d.]. Dostupné také z: `https://developer.nvidia.com/cublas`.
2. VANDEVOORDE, D.; JOSUTTIS, N.; GREGOR, D. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 2017.

3. SAAD, Y. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.
4. BELL, N.; GARLAND, M. *Efficient Sparse Matrix-Vector Multiplication on CUDA*. 2008. Tech. zpr., Technical Report NVR-2008-004. NVIDIA Corporation.
5. BELL, N.; GARLAND, M. *Implementing sparse matrix-vector multiplication on throughput oriented processors*. [N.d.]. In Supercomputing '09, Nov. 2009.
6. OBERHUBER, T.; SUZUKI, A.; VACATA, J. New Row-grouped CSR format for storing the sparse matrices on GPU with implementation in CUDA. *Acta Technica*. 2011, roč. 56, s. 447–466.
7. MONAKOV, A.; LOKHMOTOV, A.; AVETISYAN, A. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In: *HiPEAC 2010*. Springer-Verlag Berlin Heidelberg, 2010, s. 111–125.
8. HELLER, M.; OBERHUBER, T. Improved Row-grouped CSR Format for Storing of Sparse Matrices on GPU. In: Z. MINARECHOVÁ, A. H. nad; ŠEVČOVIČ, D. (ed.). *Proceedings of Algoritmy 2012*. 2012, s. 282–290.
9. ZHENG, C.; GU, S.; GU, T.-X.; YANG, B.; LIU, X.-P. BiELL: A bisection ELLPACK-based storage format for optimizing SpMV on GPUs. *Journal of Parallel and Distributed Computing*. 2014, roč. 74, č. 7, s. 2639–2647.
10. KREUTZER, M.; HAGER, G.; WELLEIN, G.; FEHSKE, H.; BISHOP, A. R. A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. *SIAM Journal on Scientific Computing*. 2014, roč. 36, č. 5, s. C401–C423.
11. MAGGIONI, M.; BERGER-WOLF, T. AdELL: An Adaptive Warp-Balancing ELL Format for Efficient Sparse Matrix-Vector Multiplication on GPUs. In: *42nd International Conference on Parallel Processing*. 2013, s. 11–20.
12. MAGGIONI, M.; BERGER-WOLF, T. CoAdELL: Adaptivity and Compression for Improving Sparse Matrix-Vector Multiplication on GPUs. In: *IEEE International Parallel & Distributed Processing Symposium Workshops*. 2014, s. 933–940.
13. MAGGIONI, M.; BERGER-WOLF, T. Optimization techniques for sparse matrix–vector multiplication on GPUs. *Journal of Parallel and Distributed Computing*. 2016, roč. 93-94, s. 66–86.
14. MUHAMMED, T.; MEHMOOD, R.; ALBESHRI, A.; KATIB, I. SURAA: A Novel Method and Tool for Loadbalanced and Coalesced SpMV Computations on GPUs. *Applied Sciences*. 2019, roč. 9, č. 947.
15. LIU, W.; VINTER, B. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In: *ICS '15: Proceedings of the 29th ACM on International Conference on Supercomputing*. 2015, s. 339–350.
16. NVIDIA. *Cusparse*. [N.d.]. Dostupné také z: `https://developer.nvidia.com/cusparse`.

17. REGULY, I.; GILES, M. Efficient sparse matrix-vector multiplication on cache-based GPUs. In: *Innovative Parallel Computing (InPar)*. 2012, s. 1–12.

18. GUO, D.; GROPP, W. Adaptive thread distributions for SpMV on a GPU. In: *BW-XSEDE '12: Proceedings of the Extreme Scaling Workshop*. 2012, s. 1–15. Č. 2.

19. DAGA, M.; GREATHOUSE, J. L. Structural Agnostic SpMV: Adapting CSR-Adaptive for Irregular Matrices. In: *IEEE 22nd International Conference on High Performance Computing (HiPC)*. 2015, s. 64–74.

20. LIU, L.; LIU, M.; WANG, C.; WANG, J. LSRB-CSR: A Low Overhead Storage Format for SpMV on the GPU Systems. In: *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. 2015, s. 733–741.

21. SCHMIDT, B.; LIU, Y. LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs. In: *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2015, s. 82–89.

22. FILLIPONE, S.; CARDELLINI, V.; BARBIERI, D.; FANFARILLO, A. Sparse Matrix-Vector Multiplication on GPGPUs. *ACM Transactions on Mathematical Software*. 2017, roč. 43, č. 4, 30:1–30:49.

23. NISA, I.; SIEGEL, C.; RAJAM, A. S.; VISHNU, A.; SADAYAPPAN, P. Effective Machine Learning Based Format Selection and Performance Modeling for SpMV on GPUs. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018, s. 1052–1065.

24. CABRERA, W.; ORDONEZ, C. Scalable parallel graph algorithms with matrix–vector multiplication evaluated with queries. *Distrib Parallel Databases*. 2017, roč. 35, 335–362. Dostupné z DOI: 10.1007/s10619-017-7200-6.

25. KLINKOVSKÝ, J.; OBERHUBER, T.; FUČÍK, R.; ŽABKA, V. Configurable open-source data structure for distributed conforming unstructured homogeneous meshes with GPU support. *submitted to ACM Trans. Math. Softw.* 2021.

26. GREEN, O. HashGraph—Scalable Hash Tables Using a Sparse Graph Data Structure. *ACM Transactions on Parallel Computing*. 2021, roč. 8, č. 2, Article No.11, 1–17. Dostupné z DOI: 10.1145/3460872.

27. OBERHUBER, T.; KLINKOVSKÝ, J.; ČEJKA, L.; FENCL, M.; KOLESNIK, I. *Segments: Parallel algorithmic pattern for computations on sparse data*. 2021.

28. LEVITAS, I. L.; K., S. Coherent solid/liquid interface with stress relaxation in a phase-field approach to the melting/solidification transition. *Physical Review B*. 2011, roč. 84, č. 140103.

29. CHEN, S.; MERRIMAN, B.; OSHER, S.; SMEREKA, P. A Simple Level Set Method for Solving Stefan Problems. *Journal of Computational Physics*. 1997, roč. 135, s. 8–29.

30. BALAŽOVJECH, M.; MIKULA, K.; PETRÁŠOVÁ, M.; J., U. Lagrangean method with topological changes for numerical modelling of forest fire propagation. In: *Proceedings of Algoritmy 2012*. 2012, s. 45–52.

31. TORNBERG, A.-K.; B., E. A finite element based level-set method for multiphase flow aplications. *Computing and Visualization in Science*. 2000, roč. 3, s. 93–101.

32. VESE, L. A.; CHAN, T. F. A Multiphase Level Set Framework for Image Segmentation Using Mumford and Shah model. *International Journal of Computer Vision*. 2002, roč. 50, s. 271–293.

33. CACACE, S.; CRISTIANI, E.; ROCCHI, L. A level set based method for fixing overhangs in 3D printing. *Applied Mathematical Modelling*. 2017, roč. 44, s. 446–455.

34. DECKELNICK, K.; DZIUK, G. Error estimates for a semi implicit fully discrete finite element scheme for the mean curvature flow of graphs. *Interfaces and Free Boundaries*. 2000, roč. 2, s. 341–359.

35. OBERHUBER, T.; SUZUKI, A.; VACATA, J.; ŽABKA, V. Image segmentation using CUDA implementations of the Runge-Kutta-Merson and GMRES methods. *Journal of Math-for-Industry*. 2011, roč. 3, s. 73–79.

36. OBERHUBER, T. Numerical solution for the anisotropic Willmore flow of graphs. *Applied Numerical Mathematics*. 2015, roč. 88, s. 1–17.

37. OBERHUBER, T. Finite difference scheme for the Willmore flow of graphs. *Kybernetika*. 2007, roč. 43, s. 855–867.

38. CROUZEIX, M.; RAVIART, P.-A. Conforming and nonconforming finite element methods for solving the stationary Stokes equations I. *Revue francaise d'automatique informatique recherche opérationelle*. 1973, roč. 7, č. R3, s. 33–75.

39. BAUER, P.; KLEMENT, V.; OBERHUBER, T.; ŽABKA, V. Implementation of the Vanka-type multigrid solver for the finite element approximation of the Navier-Stokes equations on GPU. *Computer Physics Communication*. 2016, roč. 200, s. 50–56.

40. BREZZI, F.; FORTIN, M. *Mixed and hybrid finite elements method*. Springer-Verlag, 1991.

41. FUČÍK, R.; KLINKOVSKÝ, J.; SOLOVSKÝ, J.; OBERHUBER, T.; MIKYŠKA, J. Multidimensional Mixed-Hybrid Finite Element Method for Compositional Two-Phase Flow in Heterogeneous Porous Media and its Parallel Implementation on GPU. *Computer Physics Communications*. 2019, roč. 238, s. 165–180.

Ing. Tomáš Oberhuber, Ph.D.

Vzdělání

- 2002 – 2009: doktorské studium na ČVUT v Praze, FJFI, obor Matematické inženýrství. Dizertační práce *Numerical Solution of Willmore Flow*
- 1996 – 2002: magisterské studium na ČVUT v Praze, FJFI, obor Matematické inženýrství. Diplomová práce *Využití modelovací techniky Blobs pro optimalizaci objemu grafických dat.*

Zaměstnání

- od roku 2010: odborný asistent, ČVUT v Praze, Katedra matematiky FJFI
- 2004 – 2010: pomocný asistent, ČVUT v Praze, Katedra matematiky FJFI

Zahraniční spolupráce a pobyty v zahraničí

- 2003-2004: University of Sussex, Brighton, Velká Británie, Marie-Curie fellowship, 9 měsíců.
- 2005: Marist College, Poughkeepsie, USA, školení v mainframe, 1 týden.
- 2006: CINECA, Bologna, Italy., projekt HPC Europe, 1 měsíc.
- 2007: CINECA, Bologna, Italy., projekt HPC Europe, 1 měsíc.
- 2009: EPCC, University of Edinburgh, Edinburgh, Velká Británie, projekt HPC Europe, 1 měsíc.
- 2013: Lousiana State University, Baton Rouge, Lousiana, USA, výzkumný pobyt na Katedře fyziky a astronomie, 1 měsíc.
- 2014: Lousiana State University, Baton Rouge, Lousiana, USA, výzkumný pobyt na Katedře fyziky a astronomie, 2 týdny.
- Princeton University, Princeton, Princeton, New Jersey, USA, výzkumný pobyt na Katedře fyziky plazmatu, 2 týdny.

Spoluřešitel projektu

- *Quantitative Mapping of Myocard and of Flow Dynamics by Means of MR Imaging for Patients with Nonischemic Cardiomyopathy - Development of Methodology*, AZV, projekt číslo. 15-27178A, Ministerstvo zdravotnictví České republiky, hlavní řešitel J. Tintěra, IKEM Praha, 2015-2018.
- *Analysis of flow character and prediction of evolution in endovascular treated arteries by magnetic resonance imaging coupled with mathematical modeling*, AZV, projekt číslo. NV19-08-00071, Ministerstvo zdravotnictví České republiky, hlavní řešitel J. Tintěra, IKEM Praha, 2019-2022.

Účast na projektech

- *Large structures in boundary layers over complex surfaces under high Reynolds numbers*, projekt GAČR č. 18-09539S, 2018-2020, řešitelka RNDr. K. Jurčáková, Ph.D., ÚT AV ČR, spoluřešitel R. Fučík.
- *Development of symmetry-guided methods for first principle modeling of medium-mass atomic nuclei*, Projekt GAČR, GA16-16772S, řešitel Ing. Tomáš Dytrych, Ph.D., Ústav jaderné fyziky AV ČR.
- *Center for Advanced Applied Sciences, project of excellent research*, projekt MŠMT ČR, OPVVV č. CZ.02.1.01/0.0/0.0/16_019/0000778, 2018–2022, řešitel prof. Ing. I. Jex, DrSc.
- *Research centre for Informatics*, projekt MŠMT ČR, OPVVV č. CZ.02.1.01/0.0/0.0/16_019/0000765, 2018–2022.
- *Advanced Control and Optimization of Biofuel Co-Firing in Energy Production*, projekt č. TA01020871 TAČR, 2011–2013, řešitel prof. Ing. V. Havlena, CSc., Honeywell Prague Laboratory.
- *Jindřich Nečas Center for Mathematical Modelling*, Centrum základního výzkumu, hlavní řešitel prof. RNDr. J. Málek, CSc., MFF UK, 2006–2012.

Publikace

- Autor **17 publikací** indexovaných na Web of Science [2].
- Web of Science [1] zaznamenal **59 citací** (bez autocitací), h-index 4.
- Rezence pro časopisy: International Journal of Computer Mathematics, Computers and Fluids, IAENG International Journal of Computer Science, Journal of Parallel and Distributed Computing, Kybernetika, Journal of Computational and Applied Mathematics, Open Physics.

Výuka a vedení studentů

- Garant a přednášející předmětů a cvičení: Numerická matematika 1, Pokročilá algoritmizace, Aplikace optimalizačních metod, Paralelní algoritmy a architektury, Úvod do mainframe, Programování pro mainframe, Cvičení z numerické matematiky 2
- Školitel závěrečných prací: obhájena 1 dizertační práce, 23 diplomových a 39 bakalářských prací, v roce 2021: 4 doktorandi, 2 magisterští a 3 bakalářští studenti

Další aktivity

- Autor vede vývoj softwarového projektu TNL, Template Numerical Library - www.tnl-project.org

[2] stav k 10.11.2021