



**ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE**

---

**FAKULTA BIOMEDICÍNSKÉHO INŽENÝRSTVÍ**  
Katedra biomedicínské techniky

# **Využití GPGPU a paralelního zpracování signálů pro mapování biologických signálů**

## **GPGPU and Parallel Signal Processing in Biosignal Mapping**

Bakalářská práce

Studijní program: Biomedicínská a klinická technika

Studijní obor: Biomedicínská a klinická technika

Vedoucí práce: Ing. Jan Mužík

**Martin Petřík**

---

**Kladno 2010**

# Využití GPGPU a paralelního zpracování signálů pro mapování biologických signálů

Tato práce se zabývá možnostmi využití paralelního zpracování signálů. Nejprve jsou popsány základní informace o obecných výpočtech na grafických procesorech. Dále je uveden přehled dostupných programovacích prostředků. Poté jsou stručně popsány možnosti paralelního zpracování na CPU s použitím knihovny Task Parallel Library. Nakonec jsou popsány implementace Sheppardovi a Hermitovi interpolace na CPU i GPU, jejich výhody a nevýhody a výsledky měření.

Klíčová slova: GPGPU, GPU, CPU, Task Parallel Library, OpenCL, Interpolace, Programovací jazyk C#

## GPGPU and Parallel Signal Processing in Biosignal Mapping

This work deals with possibilities of using parallel signal processing. At first, the basic information of general-purpose computing on graphics processors are described. Then the review of usable programming resources is listed. Then, the possibilities of parallel processing on CPU by using of Task Parallel Library are shortly described. At last, the implementations of Sheppard and Hermite interpolation on CPU and GPU are described, with their advantages, disadvantages and measurement results.

Key words: GPGPU, GPU, CPU, Task Parallel Library, OpenCL, Interpolation, Programming language C#

## **Poděkování**

Zde bych chtěl poděkovat především panu Ing. Janu Mužíkovi za jeho vstřícné jednání. Dále bych chtěl poděkovat zaměstnancům Společného pracoviště biomedicínského inženýrství FBMI a 1.LF na Albertově, za poskytnuté prostředky a přátelský přístup.

## **Prohlášení**

Prohlašuji, že jsem týmový projekt s názvem

*Využití GPGPU a paralelního zpracování signálů pro mapování biologických signálů*  
vypracoval(a) samostatně a použil(a) k tomu úplný výčet citací použitých pramenů,  
které uvádím v seznamu přiloženém k závěrečné zprávě.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona  
č.121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o  
změně některých zákonů (autorský zákon).

V ..... dne .....

.....

podpis

# Obsah

Úvod.....	1
1. GPGPU .....	2
2. Programování GPGPU.....	4
2.1 DirectCompute.....	5
2.2 CUDA (Compute Unified Driver Architecture) .....	5
2.3 OpenCL .....	5
3. .NET Framework .....	6
3.1 Paralelní programování na CPU v .NET .....	7
3.2 Task Parallel Library (TPL) .....	7
3.3 PLINQ - Parallel LINQ .....	10
4. Sheppardova Interpolace.....	11
4.1 Implementace s použitím TPL .....	11
4.2 Implementace s použitím OpenCL .....	15
5. Hermitova interpolace .....	19
5.1 Implementace s použitím TPL .....	20
5.2 Implementace s použitím OpenCL .....	22
6. Testování implementací.....	24
6.1 Výsledky měření Sheppardovi interpolace .....	26
6.2 Výsledky měření Hermitovi interpolace .....	30
7. Shrnutí výsledků .....	31
Závěr.....	32
Seznam použité literatury .....	33

# Úvod

Paralelní zpracování signálů v rámci bakalářské práce jsem si vybral proto, že v paralelizaci vidím velký potenciál do budoucna. Z hlediska zpracování signálů v reálném čase je paralelizace potřebných výpočtů často nutností.

K paralelizaci výpočtů se v minulých letech začínají využívat grafické procesory (GPU – Graphics Processing Unit), protože disponují vysokým výpočetním výkonem při počítání s čísly v plovoucí desetinné čárce. Tento výpočetní výkon je u dnešních grafických procesorů oproti standardním procesorům (CPU – Central Processing Unit) obrovský a s velmi rychle se rozvíjejícím herním průmyslem a tím spojenou nutností počítat realistickou fyziku v reálném čase stále roste.

V posledních letech také dochází k vývoji výkonných vícejádrových CPU. Moderní vícejádrové procesory navíc podporují SMT (Simultaneous multithreading), což umožňuje každému jádru zpracovávat několik instrukcí najednou. Například osmijádrový procesor Intel® Xeon® 7500 umožňuje každému jádru zpracovat až 16 vláken najednou, dohromady tedy až 128 instrukcí. To otevírá vývojářům další cestu k paralelizaci výpočetních algoritmů.

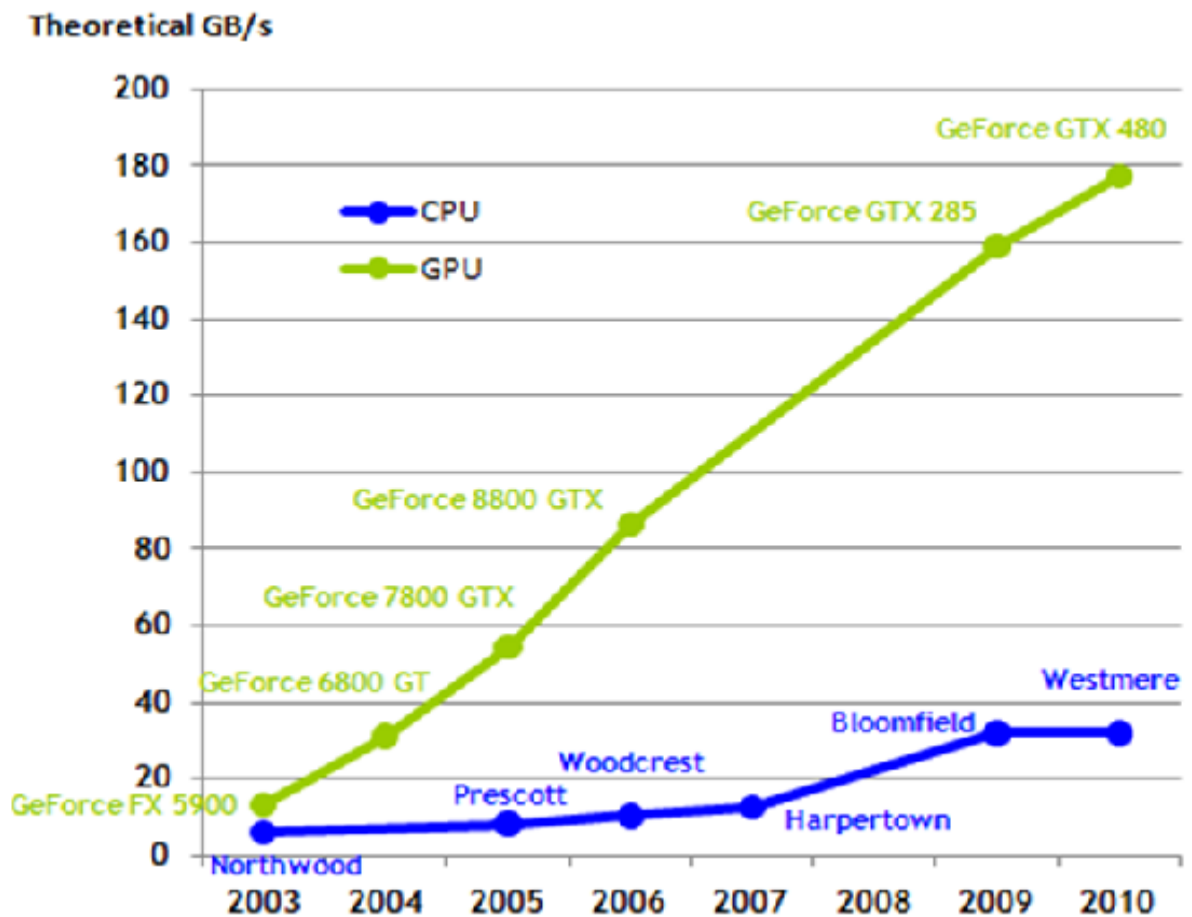
Ještě před rokem byla paralelizace algoritmů na vícejádrových procesorech poměrně náročná. S Visual Studiem 2010 však vyšel .NET Framework 4, obsahující knihovnu TPL (Task parallel library). Díky této knihovně lze jednoduše implementovat algoritmy zpracovávající výpočty paralelně.

Cílem této bakalářské práce je prozkoumat možnosti paralelního zpracování dat, vytvořit algoritmy Sheppardovi a Hermitovi interpolace s využitím paralelizace na CPU i GPU a oba způsoby porovnat, zejména rychlost. Algoritmy dále zaimplementovat do programu LiveMap sloužícího ke zpracování biologických signálů v reálném čase.

# 1. GPGPU

Používání GPU pro jiné než grafické výpočty se označuje zkratkou GPGPU - General-purpose computing on graphics processing units, což překládáme jako obecné výpočty pomocí grafických procesorů.

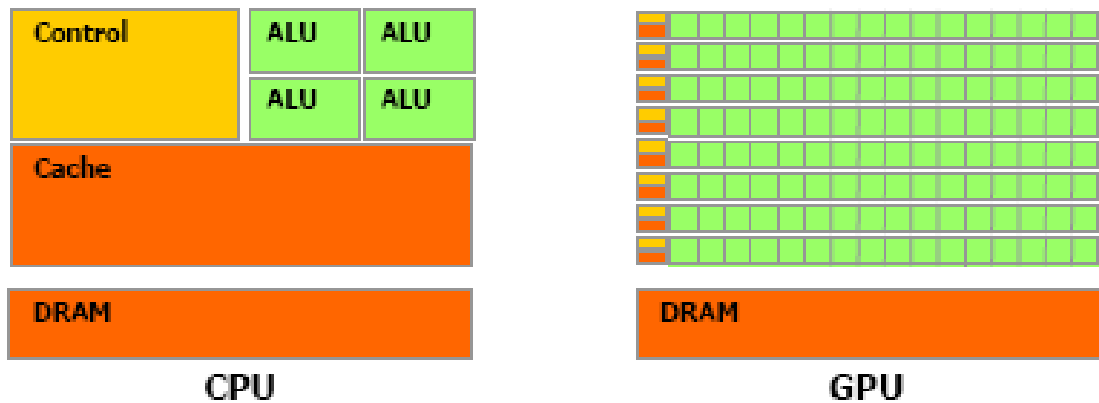
Současné grafické karty disponují asi pětadvaceti násobně větším výkonem než obyčejné procesory. V souvislosti s tím je snaha přenášet složité výpočty z procesoru na grafickou kartu.



Obr.1.1: Vývoj CPU a GPU s teoretickým výkonem, [1]

GPU má na starost především jednodušší aritmetické operace v plovoucí desetinné čárce nad vektory a maticemi. Má méně vnitřní logiky než CPU. Největší rozdíl je v počtu výpočetních jednotek (obrázek č. 1.2). Moderní CPU mají 4 až 8 jader. Na GPU lze nalézt až kolem 800 jader, ale nejedná se o plnohodnotná jádra jako u CPU, nýbrž o tzv. proudové procesory.

Hlavní výhodou GPU je to, že umožňuje datový paralelismus, což znamená, že určitá operace je prováděná zároveň nad všemi daty (proudem dat, proto proudové procesory) paralelně.



Obr.1.2: Rozdíl v architekturách CPU a GPU, [1]

Z toho pramení snaha algoritmy, které jsou na CPU velmi náročné a pomalé, paralelizovat zpracováním na grafickém procesoru a tím dosáhnout až několikanásobného zvýšení výkonu. Grafická karta tedy může být schopna, za zlomek pořizovací ceny, zastat práci výkonného superpočítače. Proto je možnost paralelizace náročných výpočtů na GPU velmi lákavá pro vědecký i komerční sektor.

Nevýhodou grafických procesorů je to, že nemají ochranu paměti. Proto nelze kontrolovat, jakému procesu patří daná část paměti. Další problém spočívá ve zpoždění na sběrnici mezi GPU a CPU.

Pro paralelní zpracování na GPU jsou nejvhodnější cykly, ve kterých probíhají operace nad polem navzájem nezávislých prvků. GPGPU našlo využití hlavně v oblastech zpracování obrazu, zpracování signálů (FFT), modelování a simulace a matematických operací s velkými maticemi.

## 2. Programování GPGPU

K programování na GPU se používá tzv. programovatelný renderovací řetězec (Programmable rendering pipeline), který se skládá ze dvou základních programovatelných částí, Vertex shaderu a Pixel shaderu. Vertex shader má na starosti geometrické zpracování a Pixel shader zpracování pixelů. Od vydání DirectX 10 Microsoftem je k dispozici další shader, Geometry shader. Ten umožňuje tvořit a odstraňovat jednotlivé Vertex shadery, čímž nabízí nové možnosti pro práci s GPU.

Programování GPGPU lze rozdělit do dvou skupin. Do první skupiny patří programovací jazyky, které využívají přímo grafické rozhraní. Pro práci s nimi je nutné rozumět procesům na grafické kartě. Jsou to:

- HLSL (High-Level Shading Language) pro grafické rozhraní DirectX
- GLSL (OpenGL Shading Language) pro grafické rozhraní OpenGL
- Cg (C for graphics) použitelné pro DirectX i OpenGL

Do druhé skupiny patří speciální architektury grafických rozhraní, většinou přímo od výrobců grafických karet, jejichž programování se díky knihovnám obsahujícím potřebné příkazy podobá běžnému programování. Například:

- SlimDX – open-source rozhraní pro programování DirectX v .NET
- ATI Stream – architektura od společnosti AMD/ATI
- DirectCompute – rozhraní od firmy Microsoft
- CUDA – architektura od firmy nVidia
- OpenCL – průmyslový standard od konsorcia Khronos Group

K realizaci algoritmů pro interpolace využívajících GPGPU v programovacím jazyku C# jsem volil mezi DirectCompute, CUDA a OpenCL.

## 2.1 DirectCompute

DirectCompute je rozhraní pro programování aplikací (API - application programming interface), které podporuje GPGPU v operačních systémech Microsoft Windows Vista a Windows 7. Je součástí sbírky aplikačních rozhraní Microsoft DirectX sloužících k umožnění přímého ovládní moderního hardwaru. DirectCompute byl vydán s DirectX 11. Nepříjemné omezení spočívá v nutnosti používat operační systém Windows 7 nebo Vista a rozhraní DirectX.

## 2.2 CUDA (Compute Unified Driver Architecture)

CUDA je architektura od výrobce grafických karet, firmy nVidia. Používá vlastní programovací jazyk C for CUDA, který je velmi podobný klasickému C, ale existují i knihovny umožňující programovat v jiných programovacích jazycích. Např. CUDA.NET pro C#. Největší výhoda spočívá v tom, že CUDA umí využívat rychlou paměť (shared memory) přímo na grafickém procesoru. Také je nejstarší zveřejněnou technologií pro obecné výpočty na GPU, proto je nejlépe rozvinutá, zdokumentovaná a mezi vývojáři velmi rozšířená. Avšak má jednu podstatnou nevýhodu, funguje jen na grafických adaptérech nVidia.

## 2.3 OpenCL

OpenCL je na poli GPGPU naprosto revoluční. Konsorcium Khronos Group, do kterého spadají například firmy Intel, Apple, AMD, nVidia, IBM, DELL a spousta dalších, vytvořilo v roce 2008 pracovní skupinu Khronos Compute Working Group, jejímž úkolem bylo vytvořit otevřený standard pro GPGPU. Tato skupina neotálela a již v prosinci 2008 došlo k uveřejnění první verze.



Obr. 2.1: Členové Khronos Group, [3]

OpenCL je opravdovým standardem. Dá se použít na široké škále hardwarových platform. Na klasických CPU, grafických procesorech (nVidia GeForce řada 8 a vyšší, ATI Radeon HD řada 4 a vyšší), procesorech digitálního signálu, procesorech Cell a dalších. Nezáleží ani na softwarové platformě, OpenCL nepotřebuje konkrétní operační systém. Podporuje Windows, Linux, MacOS a například i prostředí iPhone. OpenCL používá programovací jazyk OpenCL C. Pro implementaci v C# existuje knihovna OpenCL.NET.

Kvůli univerzálnímu využití OpenCL jsem se rozhodl implementovat interpolační metody využívající GPGPU právě s pomocí knihovny OpenCL.NET.

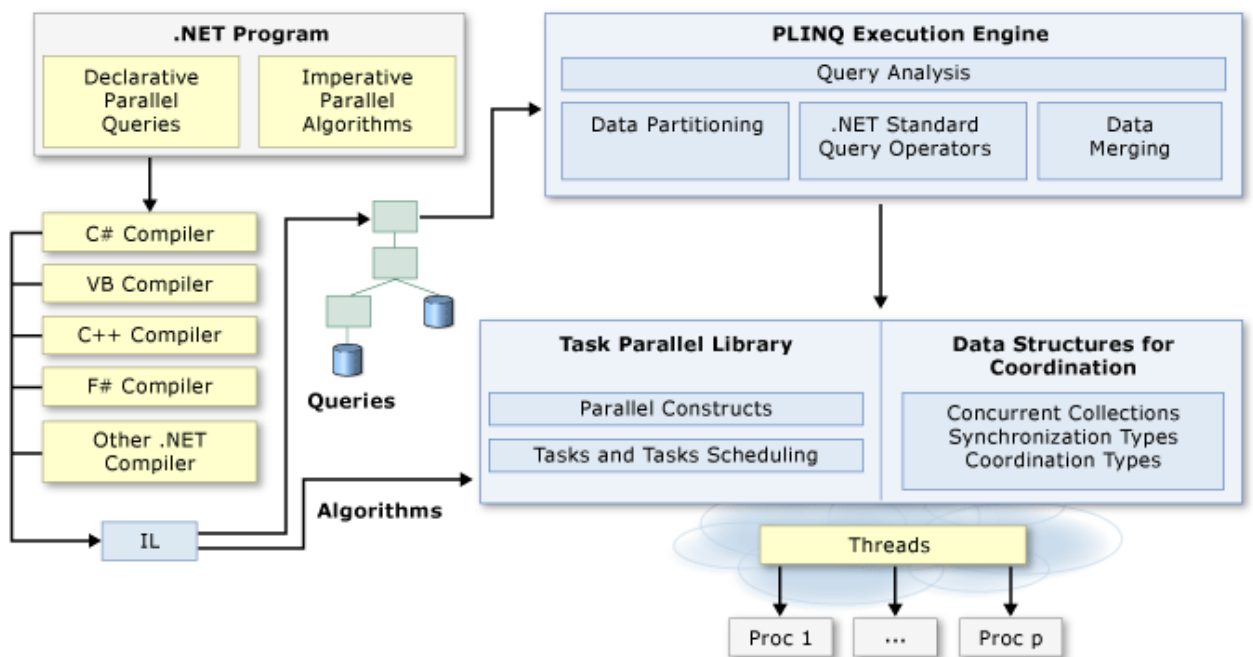
### 3. .NET Framework

„Microsoft .NET Framework je komplexní, jednotný programovací model, který slouží k sestavování aplikací s vizuálně poutavým uživatelským rozhraním a plynulou a zabezpečenou komunikací. Funguje současně se staršími verzemi tohoto rozhraní. Aplikace založené na starších verzích rozhraní budou i nadále pracovat ve verzi, pro kterou jsou ve výchozím nastavení určeny.”[3]

### 3.1 Paralelní programování na CPU v .NET

Dnešní doba nabízí množství různých počítačů a notebooků s vícejádrovými CPU. Jednojádrové CPU se používají prakticky už jen v úsporných noteboocích. Počet jader vícejádrových procesorů rychle roste. Tento fakt dává vývojářům do rukou možnost vyvíjet rychlejší a výkonnější aplikace.

Dříve musel programátor při paralelizaci zacházet s jednotlivými vlákny přímo, což bylo poměrně náročné. Visual Studio 2010 s .NET Framework 4 poskytují podporu pro paralelní programování formou nového běhového prostředí, nových knihoven tříd a novými nástroji pro diagnostiku. Tím velmi usnadňují paralelizaci vhodných aplikací a umožňují tak psaní účinných kódů bez potřeby pracovat přímo s vlákny.



Obr.3.1. Přehled architektury paralelního programování v .NET Framework 4, [4]

### 3.2 Task Parallel Library (TPL)

Knihovna TPL je základem nových Parallel Extensions for .NET obsažených v .NET frameworku 4 a sloužících programátorům ke snadnější paralelizaci úloh. Je založena na pojmu úloha (task), který reprezentuje asynchronní operaci. TPL umožňuje funkční paralelismus, kdy je více úloh spuštěno souběžně.

Knihovna TPL je umístěna v namespace System.Threading a System.Threading.Tasks. Obsahuje několik nových tříd, z nichž jsou pro nás nejzajímavější tyto:

### ***Třída Parallel***

Třída parallel je statická a obsahuje metody For, ForEach a Invoke. Metody For a ForEach jsou paralelní analogií klasických cyklů for a foreach. Obě metody mají za parametr tělo cyklu, které se vykonává paralelně. Jejich funkce spočívá v rozdělení všech volání těla cyklu do určitého počtu skupin a následném rozdělení těchto skupin mezi jednotlivá vlákna. Počet vláken je normálně nastaven na dvojnásobek počtu jader. Můžeme ho však nastavit podle potřeby předáním parametru metodě pomocí instance ParallelOptions. Paralelní cykly For a ForEach neskončí, dokud neproběhnou všechna volání těla cyklu, nebo nenastane výjimka. Parallel.For má mnoho variant. Nejjednodušší vypadá takto:

```
Parallel.For(0, 8, delegate(i)
{
    Hodnota_Funkce[i] = cislo;
});
```

Zde metoda přijímá parametry od 0 - včetně, do 8 – bez, a funkci tvořenou tělem cyklu. Máme-li tedy k dispozici osm vláken, proběhne cyklus pro  $i \in \{0,1,2,3,4,5,6,7\}$  prakticky v jednom okamžiku. U Parallel.For může být tělo cyklu tvořeno funkcemi Int, Int32, Int64 a ParallelLoopState, která umožňuje break.

Parallel.ForEach je velice podobný. Rozdíl mezi metodami For a ForEach spočívá v tom, že ForEach neprojíždí cyklus od hodnoty A do hodnoty B, ale projíždí přímo nějakou sadu dat. Pokud bychom chtěli například k číslům v telefonním seznamu přidat předvolbu, mohli bychom projíždět seznam po několika číslech najednou a přidávat předvolbu.

Poslední je metoda `Invoke`, která nezpracovává jednu funkci, ale pracuje s polem funkcí typu `Action`. Jednotlivé funkce rozdělí do vláken a spustí.

„Metoda `Parallel.Invoke` umožňuje pohodlný způsob, jak souběžně spustit libovolný počet různých příkazů. Stačí pouze předat delegáta `Action` pro každou položku práce. Nejsnadnější způsob, jak vytvořit tyto delegáty je použití lambda výrazů. Lambda výraz může buďto volat pojmenovanou metodu, nebo poskytnout vložený kód.“[5]

Například:

```
Parallel.Invoke(() => UdelejToto(), () => UdelejTamto());
```

Pokud u některé z metod `For`, `ForEach`, nebo `Invoke` nastane výjimka, je zachycena frameworkem, který se následně snaží zastavit všechna spuštěná vlákna běžícího cyklu.

„Pokud je úloha nadřazena připojené úloze, nebo pokud čekáte na více úkolů, může být vyvoláno více výjimek. Proti šíření všech výjimek zpět do hlavního vlákna je nástroj ošetřování výjimek uloží do `AggregateException` instance. `AggregateException` je uložena v `InnerExceptions`. Tyto výjimky je pak možné zpracovat samostatně“[6]

### ***Třída Task***

Třída `Task` reprezentuje asynchronní úlohu, tedy nějakou akci nebo funkci, a je základem TPL. Podobá se normálnímu vláknu, ale obsahuje několik podstatných vylepšení:

- existuje ve 2 variantách – bez a s návratovou hodnotou. Návratová hodnota je uložena v hodnotě `Result`. Pokud chceme v dalším kroku hodnotu použít, ale úloha ještě nebyla dokončena, volající vlákno se pozastaví, dokud se `Result` nevypočítá.

- Podporuje Cancellation, který umožňuje rušit úlohy nenásilně.
- Má stromové uspořádání.
- Obsahuje různé podpůrné metody

Příklad použití:

```
int SpocitejVysledek()
{
    Vysledek = cislo * cislo;
    return vysledek;
}
```

```
Task<int> uloha = new Task<int>(SpocitejVysledek);
uloha.Start();
```

### 3.3 PLINQ - Parallel LINQ

PLINQ je další součástí Parallel Extensions for .NET. Je paralelní obdobou LINQu, dotazovacího jazyku integrovaném přímo v C#. LINQ umožňuje dotazovat se v kolekcích, polích i databázích. PLINQ používá funkci AsParallel. Například, pokud máme řadu čísel a chceme z nich vyjmout jen ty sudé:

```
int[] Cisla = new [] { 1, 2, 3, 4, 5, 6, 7, 8 };
int[] SudeCisla = Cisla.AsParallel().Where(n => n % 2 == 0).ToArray();
```

## 4. Sheppardova Interpolace

Interpolace slouží k nalezení přibližných hodnot funkce v nějakém intervalu, pokud jsou známy jen některé hodnoty v tomto intervalu. Od aproximace se liší tím, že hledaná křivka prochází přímo všemi známými body.

Sheppardova interpolace patří k metodám IDW – Inverse Distance Weighting (inverzní vzdálenostní váhování). Principem Sheppardovi interpolace je přiřazení hodnot neznámým bodům použitím hodnot ze setu známých bodů. Sheppardova interpolace se provádí podle následujícího vzorce:

$$u(x) = \sum_{k=0}^N \frac{w_k(x)}{\sum_{k=0}^N w_k(x)} u_k$$

kde

$$w_k(x) = \frac{1}{d(x, x_k)^p}$$

$W_k$  je váhovací funkce.  $X$  reprezentuje body s neznámou hodnotou, které chceme interpolovat.  $X_k$  jsou body, jejichž hodnoty známe a  $d$  reprezentuje rozdíl vzdáleností mezi  $X$  a  $X_k$ .  $P$  je tzv. power parameter, kladné reálné číslo.  $N$  je celkový počet známých bodů.  $U$  reprezentuje hodnotu ve známém bodě, například napětí na elektrodě.  $U(x)$  je pak hodnota v bodě, který interpolujeme.

### 4.1 Implementace s použitím TPL

Jelikož chceme mapovat signály, budeme se zabývat interpolací dvojrozměrnou. Při Sheppardově interpolaci je k interpolování mapy použito hodnot celého setu známých bodů, v našem případě elektrod. Pro interpolaci musíme znát umístění elektrod a jejich napětí. Výhoda Sheppardovi interpolace spočívá v tom, že pro každé umístění elektrod stačí spočítat váhovací funkci jen jednou. Váhovací funkce totiž není závislá přímo na napětích elektrod, ale jen na jejich rozložení.

Z tohoto důvodu lze Sheppardovu interpolaci rozdělit na dvě části, přípravu spočívající ve výpočtu váhovací funkce a samotnou interpolaci.

První implementace je realizována v programovacím jazyku s použitím knihovny TPL a zakomponována do programu LiveMap. Váhovací funkcí je zde trojrozměrná váhovací matice *WeightMatrix*. Tři rozměry proto, že je třeba ji počítat pro každou elektrodu. Váhovací matice se spočítá po zavolání funkce *Prepare*. Výpočet je rozdělen na tři části:

```
Parallel.For(0, channels.Count, delegate(int a)
{
    WeightMatrix[a] = new double[wmWidth, wmHeight];

    for (int b = 0; b < wmWidth; b++)
        for (int c = 0; c < wmHeight; c++)
            {

                double d = (canvasPosition.X - channels[a].X);
                double e = (canvasPosition.Y - channels[a].Y);
                double distance = Math.Sqrt(d * d + e * e);

                WeightMatrix[a][b, c] = 1 / Math.Pow(distance, PowerParameter);

            }
});
```

V první části je nejprve spuštěn For cyklus, který paralelně projíždí všechny elektrody. Pro každou elektrodu je vytvořena váhovací matice *WeightMatrix*. V následujících dvou For cyklech se po jednotlivých pixelech projíždí oblast, kterou chceme interpolovat. Jelikož pozice elektrod nejsou v pixelech, ale ve fyzikálních jednotkách, musíme pozici pixelu přepočítat. V programu LiveMap k tomu slouží funkce *PositionInCanvas*, která nám každé *b* a *c* převede na *canvasPosition.X* a *canvasPosition.Y*. Poté je Pythagorovou větou vypočítána vzdálenost mezi daným pixelem a elektrodou. Převrácená hodnota vzdálenosti je umocněna *PowerParameterem* (v mém případě je roven 2) a uložena do váhovací matice.

```

for (int a = 0; a < wmWidth; a++)
  for (int b = 0; b < wmHeight; b++)
  {

    SumWM = 0;

    for (int c = 0; c < channels.Count; c++)
      SumWM = SumWM + WeightMatrix[c][a, b];

    for (int d = 0; d < channels.Count; d++)
      WeightMatrix[d][a, b] = WeightMatrix[d][a, b] / SumWM;

  }

```

V druhé části výpočtu je každá váhová matice vydělena sumou všech ostatních. Váhová matice se projíždí po pixelech a pro každý pixel jsou sečteny hodnoty všech matic. Nakonec je hodnota jednotlivých váhových matic pro daný pixel vydělena sumou. Suma se musí pro další pixel vynulovat.

```

for (int a = 0; a < channels.Count; a++)
  for (int b = 0; b < channels.Count; b++)
  {
    if ((bitmapPosition.X - ZeroPosition.X) < wmWidth &&
        (bitmapPosition.Y - ZeroPosition.Y) < wmHeight &&
        (bitmapPosition.X - ZeroPosition.X) > 0 &&
        (bitmapPosition.Y - ZeroPosition.Y) > 0)
    {
      WeightMatrix[a][bitmapPosition.X - ZeroPosition.X,
        bitmapPosition.Y - ZeroPosition.Y] = Convert.ToDouble(a == b);
    }
  }
}

```

V poslední části výpočtu dochází k nahrazení NaN (not a number) jedničkou. Tyto NaN vznikají v první části výpočtu v místě umístění elektrody, protože počítáme převrácenou hodnotu z nulové vzdálenosti. Pomocí funkce *PositionInBitmap* jsou převedeny souřadnice elektrody *b* z fyzikálních jednotek na souřadnice pixelu (*bitmapPosition.X* a *bitmapPosition.Y*). V programu LiveMap lze zvolit velikost celkové oblasti *canvasArea* i velikost oblasti, kterou chceme interpolovat – *mapArea*. Z toho důvodu je nutné spočítat posun oblasti *mapArea* od absolutního počátku. Proto je opět pomocí funkce *PositionInBitmap* spočítána *ZeroPosition*.

Jelikož neinterpolujeme od pozice [0,0], ale [*ZeroPosition.X,ZeroPosition.Y*], musíme tyto souřadnice od aktuálního pixelu odečíst. Podmínka *if* hlídá, aby souřadnice nebyly záporné nebo větší než interpolovaná oblast. Pokud je *a* rovno *b*, je hodnota NaN na pozici elektrody nahrazena.

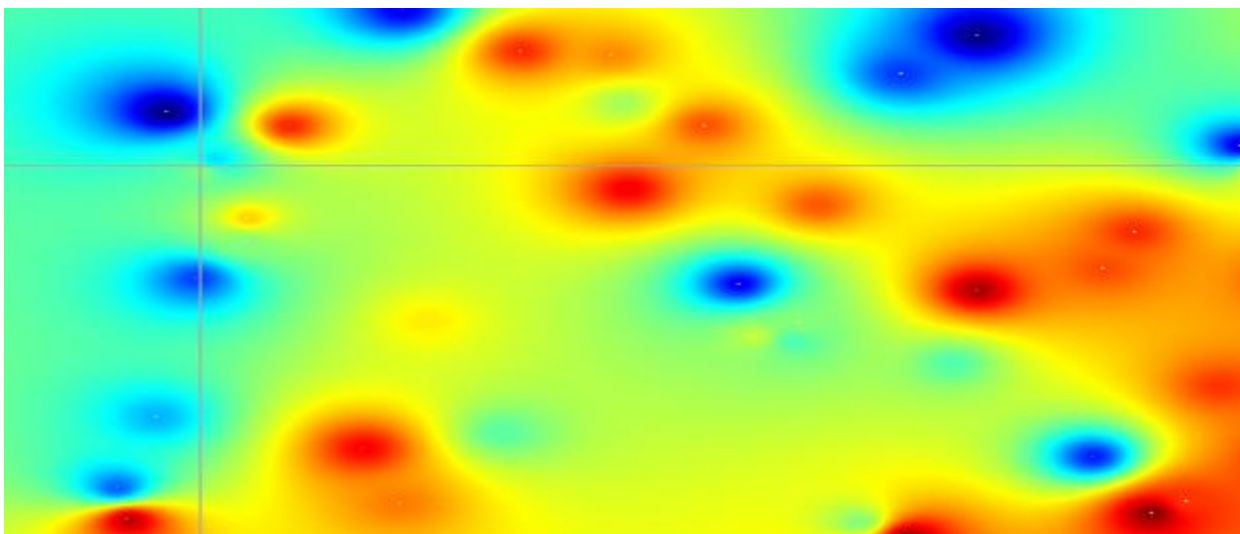
Samotná interpolace probíhá po zavolání funkce *InterpolateToBitmap*:

```
Parallel.For(0, mapFiller.Width, delegate(int a)
{
    for (int b = 0; b < mapFiller.Height; b++)
    {
        double val;

        if (mapArea.Contains(canvasPosition))
        {
            val = 0;

            for (int c = 0; c < channels.Count; c++)
            {
                val += WeightMatrix[c][a - ZeroPosition.X,
                b - ZeroPosition.Y]*values[c];
            }
            mapFiller[a, b] = ColorMap.ToColorSafe(val);
        }
        else
        {
            mapFiller[a, b] = BackColor;
        }
    }
}
```

Paralelně je po pixelech projížděna celá canvasArea. Pokud není splněna podmínka *if*, pixel neleží v interpolované oblasti a je mu přiřazena šedá barva. Pokud pixel leží v interpolované oblasti, je zahájen výpočet. Výsledná hodnota pixelu je sumou součinů hodnot váhovacích matic všech elektrod daného pixelu s hodnotami příslušných elektrod. Poté je výsledná hodnota uložena do barevné bitmapy v příslušné barvě.



Obr.4.1 Příklad Sheppardovi interpolace v programu LiveMap

## 4.2 Implementace s použitím OpenCL

Pro programování s OpenCL v jazyce C# je k dispozici knihovna OpenCL.NET. Pro implementaci Sheppardovi interpolace jsem využil knihovnu OpenCLTemplate, jelikož obsahuje již předdefinované funkce. Například samotná inicializace OpenCL, která je při tomto programování velmi složitá, se zavolá jediným příkazem *InitCL()*. To nám sice neumožňuje nastavit parametry zařízení podle potřeb, nicméně pro naše účely tato implementace stačí.

Váhování je opět provedeno ve funkci *Prepare*:

```
OpenCLTemplate.CLCalc.InitCL();  
OpenCLTemplate.CLCalc.Program.Compile(new string[] { Sheppard });  
OpenCLTemplate.CLCalc.Program.Kernel kernelSheppard = new  
OpenCLTemplate.CLCalc.Program.Kernel("prepareWeightMatrix1");
```

Nejprve dojde k inicializaci OpenCL. Poté je zkompileován řetězec *Sheppard*, z něhož je do funkce *kernelSheppard* načten kernel *prepareWeightMatrix1*. Kernel je funkce sloužící ke zpracování na GPU.

```

for (int b = 0; b < channels.Count; b++)
{
    int[] channel = new int[1] { b };
    weightMatrix[b] = new float[wmWidth * wmHeight];

    OpenCLTemplate.CLCalc.Program.Variable[] args = new
    OpenCLTemplate.CLCalc.Program.Variable[] { varChannelsX, varChannelsY,
    varcanvasx, varcanvasY, varChannel, varWeightMatrix };

    kernelSheppard.Execute(args, workers);

    varWeightMatrix.ReadFromDeviceTo(weightMatrix[b]);
}

```

Výpočet váhovací matice se provádí po jednotlivých elektrodách. Pro každou elektrodu je spuštěn kernel, který vrátí příslušnou váhovací matici. Jelikož grafický procesor provádí výpočet se všemi prvky pole najednou, musí být *weightMatrix* jednorozměrné pole. Tím je dosaženo toho, že má každý prvek pole přidělenou pozici formou jediného čísla.

V proměnné *varChannelsX* jsou uloženy x-ové souřadnice elektrod, ve *varChannelY* y-ové, ve *varcanvasx* a *varcanvasy* jsou uloženy souřadnice všech pixelů přepočítané do fyzikálních jednotek. Proměnná *varChannel* nese informaci o tom, pro jakou elektrodu váhujeme. Do proměnné *varWeightMatrix* se ukládá výstupní matice. Všechny tyto proměnné jsou uloženy do jediné proměnné *args*. Proměnná *workers* obsahuje informaci o počtu řádků a sloupců *wmWidth* a *wmHeight*. Poté je spuštěn kernel, proběhne výpočet na grafickém procesoru a pomocí funkce *ReadFromDevice* je výstupní matice dané elektrody uložena do proměnné *weightMatrix[b]*. Samotný kernel vypadá takto:

```

__kernel void prepareWeightMatrix1(
__global float *inChannelsX,
__global float *inChannelsY,
__global float *incanvasx,
__global float *incanvasy,
__global int *channel,
__global float *outWeightMatrix )
{
    int c = channel[0];
    int i = get_global_id(0);
    int j = get_global_id(1);

```

```

float distX = incanvasx[i] - inChannelsX[c];
float distY = incanvasy[j] - inChannelsY[c];
float distance = native_sqrt( distX*distX + distY*distY );

if (distance==0)
    distance = 0.0001;

outWeightMatrix[i+j*get_global_size(0)] = 1/(distance*distance);
}

```

V řádcích začínajících `__global` jsou načteny předešlé proměnné. Do proměnné `c` je načtena informace o tom, pro jakou elektrodu počítáme vzdálenosti. Do proměnných `i` a `j` jsou načteny souřadnice jednotlivých pixelů. Poté je pythagorovou větou spočítána vzdálenost aktuálního pixelu od příslušné elektrody. Pokud je tato vzdálenost nulová, nahradíme nulu malým číslem. Nakonec je převrácená hodnota druhé mocniny vzdálenosti uložena do matice na příslušnou pozici. Celý výpočet proběhne pro všechny pixely matice najednou a kernel vrátí výslednou matici.

Poté se pomocí tří jednoduchých for cyklů převedou jednorozměrné pole `weightMatrix[ ]` na dvojrozměrné matice `WeightMatrix[ , ]`:

```

for (int n = 0; n < channels.Count; n++)
    for (int i = 0; i < wmWidth; i++)
        for (int j = 0; j < wmHeight; j++)
            WeightMatrix[n][i, j] = weightMatrix[n][i + wmWidth * j];

```

Váhování sumou je pak provedeno stejným způsobem jako u předchozí implementace. Při implementaci tohoto kroku v OpenCL vznikali problémy s alokací paměti a vzhledem k jednoduchosti výpočtu bylo výhodnější nechat tento krok proběhnout na CPU. Po váhování je váhovací matice zpětně převedena do jednorozměrného pole, aby byla připravena na samotnou interpolaci.

```

for (int a = 0; a < channels.Count; a++)
    for (int i = 0; i < wmWidth ; i++)
        for (int j = 0; j < wmHeight; j++)
            weightMatrix[a][i + wmWidth * j] = (float)WeightMatrix[a][i, j];

```

Interpolace je opět spuštěna po zavolání funkce *InterpolateToBitmap*.

```
for (int a = 0; a < channels.Count; a++)
{
    int[] channel = new int[1] { a };

    OpenCLTemplate.CLCalc.Program.Variable[] arg = new
    OpenCLTemplate.CLCalc.Program.Variable[] { varChannel, varValues,
    varWeightMatrix, varValue };

    kernelInterpolate.Execute(arg, workers);
}
varValue.ReadFromDeviceTo(value);
```

Výpočet se opět provádí po jednotlivých elektrodách. V proměnné *varChannel* je uloženo pořadí elektrody, v proměnné *varValues* jsou uloženy hodnoty elektrod. V proměnné *varWeightMatrix* jsou uloženy jednotlivé váhovací matice a do proměnné *varValue* se ukládají vypočtené hodnoty. Pro každou elektrodu je spuštěn kernel a po proběhnutí celého cyklu jsou hodnoty uloženy do pole *value*. Kernel vypadá takto:

```
__kernel void getvalues(
__global int *channel,
__global float *invalues,
__global float *inWeightMatrix,
__global float *outvalue )
{
    int c = channel[0];
    int i = get_global_id(0);
    int j = get_global_id(1);

    outvalue[i+j*get_global_size(0)] =
    outvalue[i+j*get_global_size(0)]+(inWeightMatrix[i+j*get_global_size(0)]
    * invalues[c]);
}
```

Proměnné *c*, *i* a *j* mají stejný význam jako v předešlém kernelu. Samotný výpočet spočívá ve vynásobení hodnoty váhovací matice daného pixelu s hodnotou příslušné elektrody, a přičtením hodnoty z předchozího kroku. Po projetí celého cyklu je tedy výstupní hodnota rovna sumě těchto hodnot. Poté se provede samotné vykreslení.

```

for (int i = 0; i < mapFiller.Width; i++)
    for (int j = 0; j < mapFiller.Height; j++)
    {
        if (mapArea.Contains(canvasPosition))
        {
            {
                mapFiller[i, j] = ColorMap.ToColorSafe(value[(i - ZeroPosition.X)
                    +wmWidth*(j-ZeroPosition.Y)]);
            }
        }
        else
        {
            mapFiller[i, j] = BackColor;
        }
    }
}

```

Opět je projížděna celá *canvasArea* a pokud se aktuální pixel nachází v interpolované oblasti, je příslušná hodnota uložena do bitmapy. Nesmíme zapomenout odečíst *ZeroPosition*. Pokud pixel neleží v interpolované oblasti, je mu přiřazena šedá barva.

## 5. Hermitova interpolace

Hermitova interpolace je rozšířením základních polynomických interpolací. Nevyužívá pouze hodnot bodů, ale také jejich derivací. V našem případě se jedná o bikubickou interpolaci, která je uskutečněna podle následujícího vzorce.

$$u(x, y) = \sum_{i=0}^3 \sum_{j=0}^3 a_{ij} x^i y^j ,$$

kde  $x$  a  $y$  jsou souřadnice interpolovaného bodu v mřížce mezi čtyřmi elektrodami, přepočítané na rozsah 0 až 1 a  $a_{ij}$  jsou interpolační koeficienty, kterých je celkem 16. To v praxi znamená, že pokud interpolujeme oblast mezi čtyřmi elektrodami, používáme hodnoty i z okolních dvanácti elektrod. Z toho plyne zásadní nevýhoda této interpolace. Elektrody musí být rozloženy v pravidelné mřížce.

## 5.1 Implementace s použitím TPL

U Hermitovi interpolace musíme provádět kompletní výpočet při každém interpolování. Proto se po zavolání funkce *Prepare* pouze uloží souřadnice jednotlivých řádků a sloupců elektrod do proměnných *x* a *y*, aby bylo možné projíždět interpolovanou oblast po jednotlivých částích ohraničených čtyřmi elektrodami.

```
for (int a = 0; a < x.Length; a++)
    x[a] = channels[a].X;

for (int a = 0; a < y.Length; a++)
    y[a] = channels[a * x.Length].Y;
```

Pak už je možno spustit funkci *InterpolateToBitmap*. Pomocí paralelního For cyklu se projíždí jednotlivé části interpolované oblasti. V každé části se musí nejprve přepočítat pozice elektrod vzhledem k interpolované oblasti, aby bylo možné dosadit jejich hodnoty do interpolačních koeficientů:

```
Parallel.For(0, y.Length-1, delegate(int a)
{
    for (int b = 0; b < x.Length - 1; b++)
    {
        for (int k = 0; k < 4; k++)
            for (int l = 0; l < 4; l++)
            {
                xk = b + k - 1;
                yl = a + l - 1;

                if (xk < 0)
                    xk = 0;
                if (yl < 0)
                    yl = 0;
                if (xk > x.Length - 1)
                    xk = x.Length - 1;
                if (yl > y.Length - 1)
                    yl = y.Length - 1;

                e[k, l] = xk + yl * x.Length;
            }
    }
}
```

Nejprve jsou přepočteny souřadnice  $x_k$  a  $y_l$ . Podmínky *if* přiřazují při interpolování krajních oblastí teoretickým elektrodám mimo oblast hodnotu nejbližší elektrody. Hodnoty přepočítaných elektrod jsou uloženy do matice  $e[k,l]$ . Následují výpočty interpolačních koeficientů. Některé jsou příliš dlouhé, proto uvedu jen tři:

```
aij[0, 0] = values[e[1, 1]];
aij[0, 1] = -0.5 * values[e[1, 0]] + 0.5 * values[e[1, 2]];
aij[1, 0] = -0.5 * values[e[0, 1]] + 0.5 * values[e[2, 1]];

if (b == 0)
    bitmapPosition.X = 0;

if (a == 0)
    bitmapPosition.Y = 0;

if (b + 1 == x.Length - 1)
    bitmapPosition2.X = mapFiller.Width;

if (a + 1 == y.Length - 1)
    bitmapPosition2.Y = mapFiller.Height;
```

Po výpočtu koeficientů je třeba ohlídat krajní oblasti, protože nechceme interpolovat až od první a jen do poslední elektrody, ale celou interpolovanou oblast. `BitmapPosition` a `BitmapPosition2` jsou pozice od *a* do kterých interpolujeme. Díky podmínkám *if* jsou interpolovány i krajní oblasti. Pokud tedy máme správně ohraničenou oblast, můžeme interpolovat:

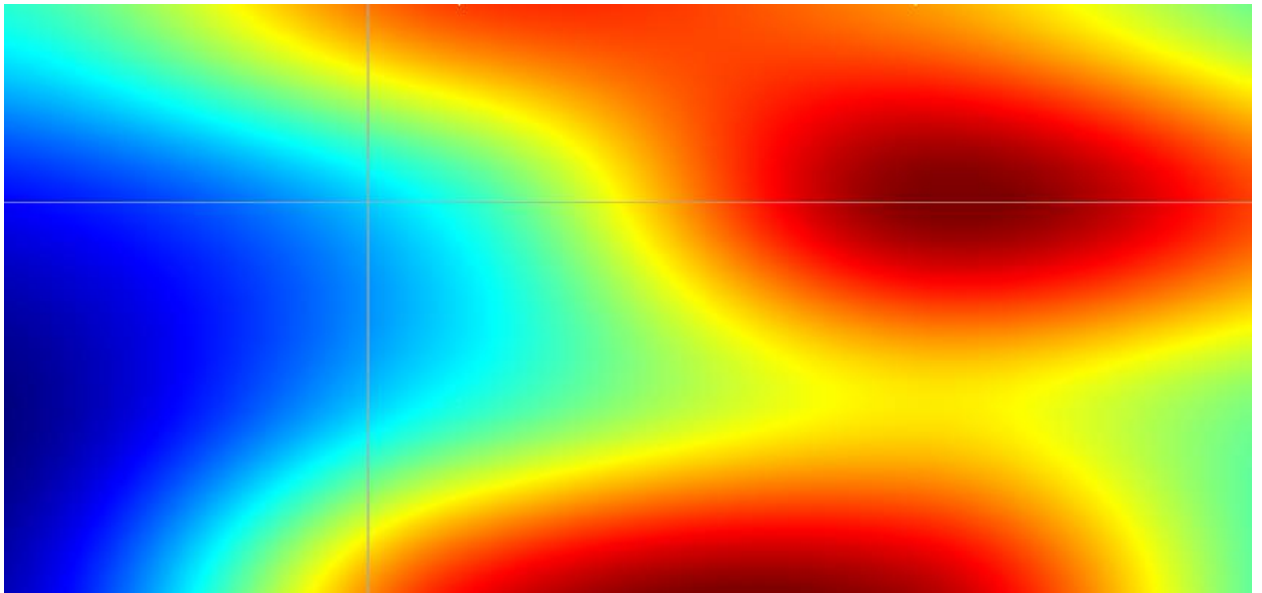
```
for (int c = bitmapPosition.Y; c < bitmapPosition2.Y; c++)
    for (int d = bitmapPosition.X; d < bitmapPosition2.X; d++)
    {
        if (mapArea.Contains(canvasPosition))
        {
            double xd = d0to1;
            double yd = c0to1;
            double xd2 = xd * xd;
            double xd3 = xd2 * xd;
            double yd2 = yd * yd;
            double yd3 = yd2 * yd;
            val = aij[0, 0] + aij[0, 1] * yd + aij[0, 2] * yd2 + aij[0, 3] * yd3 +
                aij[1, 0] * xd + aij[1, 1] * xd * yd + aij[1, 2] * xd * yd2 +
                aij[1, 3] * xd * yd3 + aij[2, 0] * xd2 + aij[2, 1] * xd2 * yd +
                aij[2, 2] * xd2 * yd2 + aij[2, 3] * xd2 * yd3 + aij[3, 0] * xd3 +
                aij[3, 1] * xd3 * yd + aij[3, 2] * xd3 * yd2 + aij[3, 3] * xd3
                * yd3;
        }
    }
```

```

    mapFiller[d, c] = ColorMap.ToColorSafe(val);
}
else
{
    mapFiller[d, c] = BackColor;
}
}

```

Pomocí dvou For cyklů projždíme danou oblast, a pokud je pixel v interpolované oblasti, přepočítáme jeho pozici do rozsahu 0 až 1 pomocí funkce *NormalizeTo01*. Tuto pozici uložíme do *xd* a *yd*, vytvoříme mocniny a spočítáme výslednou hodnotu, kterou uložíme do bitmapy.



Obr.5.1 Příklad Hermitovi interpolace v programu LiveMap

## 5.2 Implementace s použitím OpenCL

Tato implementace je téměř totožná s předchozí. Opět se interpolovaná oblast projíždí po jednotlivých částích, ale s tím rozdílem, že hodnoty pro každou interpolovanou oblast se spočtou najednou pro celou tuto oblast. K tomu slouží kernel *getvalues*.

```

__kernel void getvalues (
__global float *inbitmapy,
__global float *inbitmapx,
__global float *inbitmap2y,
__global float *inbitmap2x,
__global float *inparametr,
__global float *outvalue )
{
    int i = get_global_id(0);
    int j = get_global_id(1);

    float yd = i / (inbitmap2y - inbitmapy);
    float xd = j / (inbitmap2x - inbitmapx);
    float yd2 = yd*yd;
    float xd2 = xd*xd;
    float yd3 = yd2*yd;
    float xd3 = xd2*xd;

    outvalue[i+j*get_global_size(0)] = parametr[0] + parametr[1] * yd +
    parametr[2] * yd2 + parametr[3] * yd3 + parametr[4] * xd +
    parametr[5] * xd * yd + parametr[6] * xd * yd2 + parametr[7] * xd
    * yd3 + parametr[8] * xd2 + parametr[9] * xd2 * yd + parametr[10]
    * xd2 * yd2 + parametr[11] * xd2 * yd3 + parametr[12] * xd3 +
    parametr[13] * xd3 * yd + parametr[14] * xd3 * yd2 + parametr[15]
    * xd3 * yd3;
}

```

V proměnných *inbitmapy*, *inbitmapx*, *inbitmap2y* a *inbitmap2x* jsou načteny hodnoty *bitmapPosition.Y*, *bitmapPosition.X*, *bitmapPosition2.Y* a *bitmapPosition2.X*. V proměnné *parametr* jsou načteny interpolační koeficienty. Do proměnné *outvalue* se ukládají výsledné hodnoty. Nejprve se přepočítají souřadnice pixelu do rozsahu 0 až 1, poté jejich druhé a třetí mocniny. Nakonec je proveden výpočet interpolované hodnoty.

Interpolované hodnoty jsou uloženy do proměnné *hodnota*, která je následně převedena pomocí dvou For cyklů na matici *hodnoty*.

```

for (int i = 0; i < (bitmapPosition2.Y-bitmapposition.Y); i++)
    for (int j = 0; j < (bitmapPosition2.X-bitmapposition.X); j++)
        hodnoty[i, j] = hodnota[i + (bitmapPosition2.Y-bitmapposition.Y) * j];

```

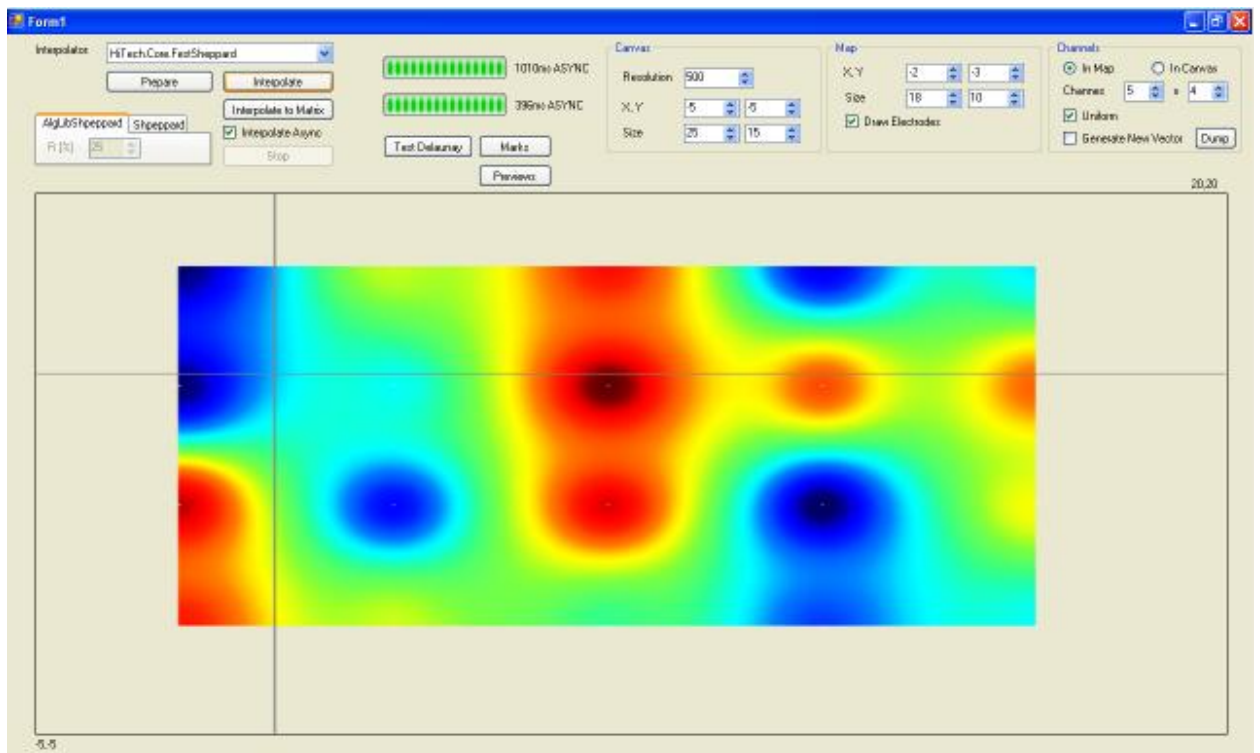
Nakonec dochází k samotnému vykreslování.

```
for (int c = bitmapPosition.Y; c < bitmapPosition2.Y; c++)
{
    for (int d = bitmapPosition.X; d < bitmapPosition2.X; d++)
    {
        if (mapArea.Contains(canvasPosition) && (c < bitmapPosition2.Y -
            bitmapPosition.Y) && (d < bitmapPosition2.X - bitmapPosition.X))
        {
            mapFiller[d, c] = ColorMap.ToColorSafe(hodnoty[c - ZeroPosition.X,
                d - ZeroPosition.Y]);
        }
        else
        {
            mapFiller[d, c] = BackColor;
        }
    }
}
```

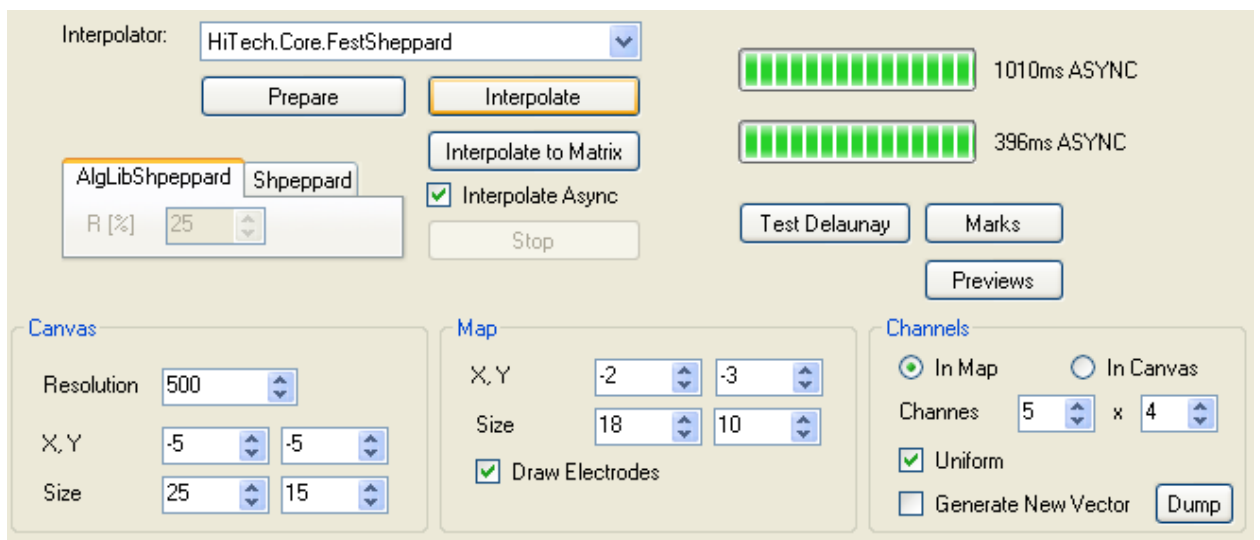
Tuto implementaci jsem bohužel nedokázal vyřešit zcela ideálně. Díky podmínce *if* se vykreslí zhruba jen čtvrtina pixelů. Nepodařilo se mi vymyslet správný posun pozic výsledných hodnot vzhledem k *ZeroPosition*. Aby program nepadal, musel jsem zavést tyto omezující podmínky. Nicméně kernel počítá interpolované hodnoty správně. Proto byla do měření zahrnuta i tato implementace. Nesprávné vykreslování nemá na čas výpočtu prakticky žádný vliv.

## 6. Testování implementací

Testování implementací jsem prováděl přímo v programu LiveMap, do kterého se mi podařilo všechny tyto metody zimplementovat. LiveMap je komplexní program sloužící pro zpracování biologických signálů. Pro testování implementací obsahuje formulář *TestInterpolators*.



Obr.6.1 Testování interpolací ve formuláři TestInterpolators



Obr.6.2 Ovládací prvky formuláře TestInterpolators

Nejprve zvolíme, jaký interpolátor použijeme. Tlačítko *Prepare* spouští funkci *Prepare* a tlačítko *Interpolate* funkci *InterpolateToBitmap*. Tlačítko *Interpolate To Matrix* spustí funkci *InterpolateToMatrix*, která je totožná s funkcí *InterpolateToBitmap*, ale místo do bitmapy ukládá interpolované hodnoty do matice. Matice je po dokončení interpolace exportována do Excelu.

Dále můžeme sledovat procentuální stav jednotlivých operací. Po skončení přípravy nebo interpolace se zobrazí čas jejich provedení v milisekundách. Horní údaj je pro přípravu a dolní pro interpolaci. Pro oblast *Canvas* lze nastavit počátek a velikost ve fyzikálních jednotkách i rozlišení v pixelech. Jelikož interpolujeme po jednotlivých pixelech, má rozlišení zásadní vliv na rychlost. Rozlišení 500 znamená, že oblast *Canvas* obsahuje 500 krát 500 pixelů. Počátek a velikost interpolované mapy ve fyzikálních jednotkách lze nastavit v okénku *Map*. Po označení *Draw Electrodes* jsou bílou barvou vykreslovány pozice elektrod. Počet kanálů (elektrod) lze nastavit v okénku *Channels*. Můžeme zde také zvolit, jestli mají být kanály rozmístěny pouze v interpolované mapě, nebo celé oblasti *Canvas*. Pokud označíme *Uniform*, budou kanály rozmístěny v pravidelné mřížce. Označením *Generate New Vector* lze vygenerovat nové kanály.

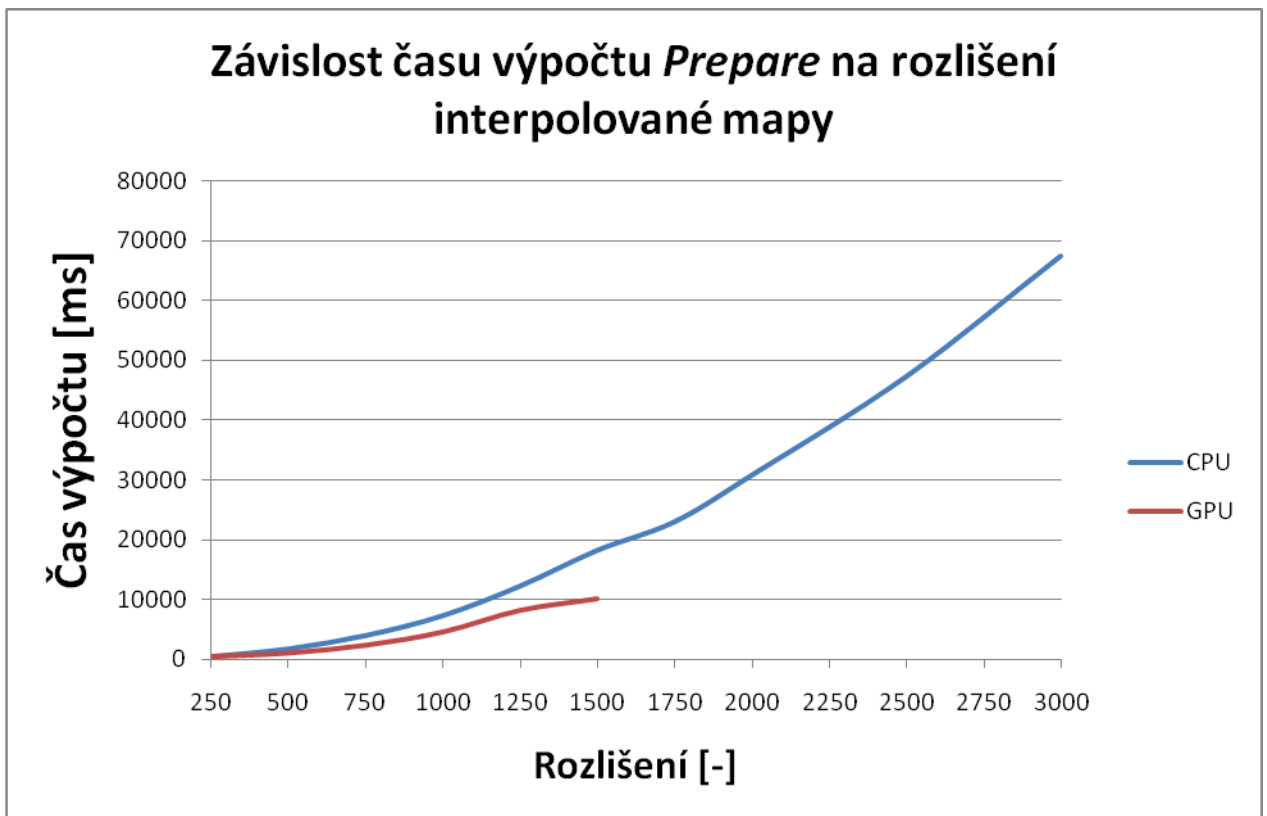
Program LiveMap umí i vizualizace na trojrozměrné modely. Z toho důvodu nebylo nutné vytvářet vizualizační plugin.

Všechny čtyři metody jsem otestoval a změřil na dvou PC sestavách. První s dvoujádrovým procesorem Intel Core2Duo 2160 1,8GHz a grafickou kartou NVidia GeForce 8600GT (Rychlost grafického čipu 560MHz, Rychlost grafických pamětí 1400MHz, Šířka paměťové sběrnice 128 bit, Počet shaderů 32), a druhou s čtyřjádrovým procesorem Intel Core i7 860 2,8GHz s podporou SMT (může zpracovávat až 8 instrukcí najednou) a grafickou kartou ATI radeon 4870 (Rychlost grafického čipu 750MHz, Rychlost grafických pamětí 3600MHz, Šířka paměťové sběrnice 512 bit, Počet shaderů 320).

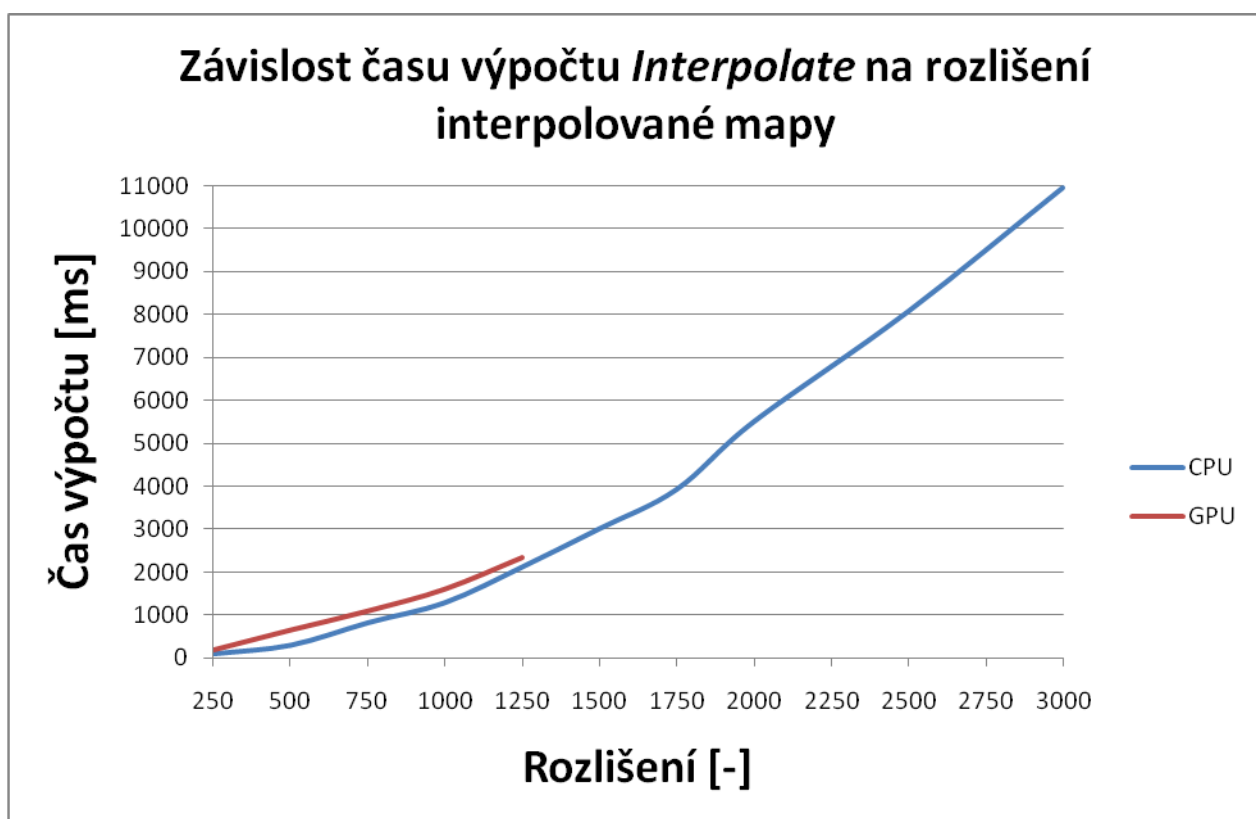
## 6.1 Výsledky měření Sheppardovi interpolace

Pro každé rozlišení bylo provedeno několik měření, z nichž byl udělán průměr. Jelikož interpolace probíhali přes celou oblast *Canvas*, jedná se přímo o rozlišení interpolované oblasti. Každé měření proběhlo pro 20 elektrod v pravidelné mřížce. V prvním měření (na první PC sestavě) byly změřeny výpočetní časy funkcí *Prepare* (graf.6.1) a *InterpolateToBitmap* (graf.6.2) na CPU i GPU.

U funkce *Prepare* grafický procesor pracoval rychleji, ale při rozlišení větším, než 1500 pixelů měl problémy s alokací paměti. To mohlo být způsobeno nedostačujícím počtem shaderů, ale i nevhodnou implementací. K výpočtům na GPU byla využívána jen globální paměť, jelikož použití lokální paměti a dalších optimalizací vyžaduje vyšší programátorské znalosti. Při rozlišení 1500 byla grafická karta rychlejší o 8177ms. Při interpolaci byl CPU rychlejší řádově o stovky ms. Grafický procesor měl při rozlišení 1250 opět problém s alokací paměti.

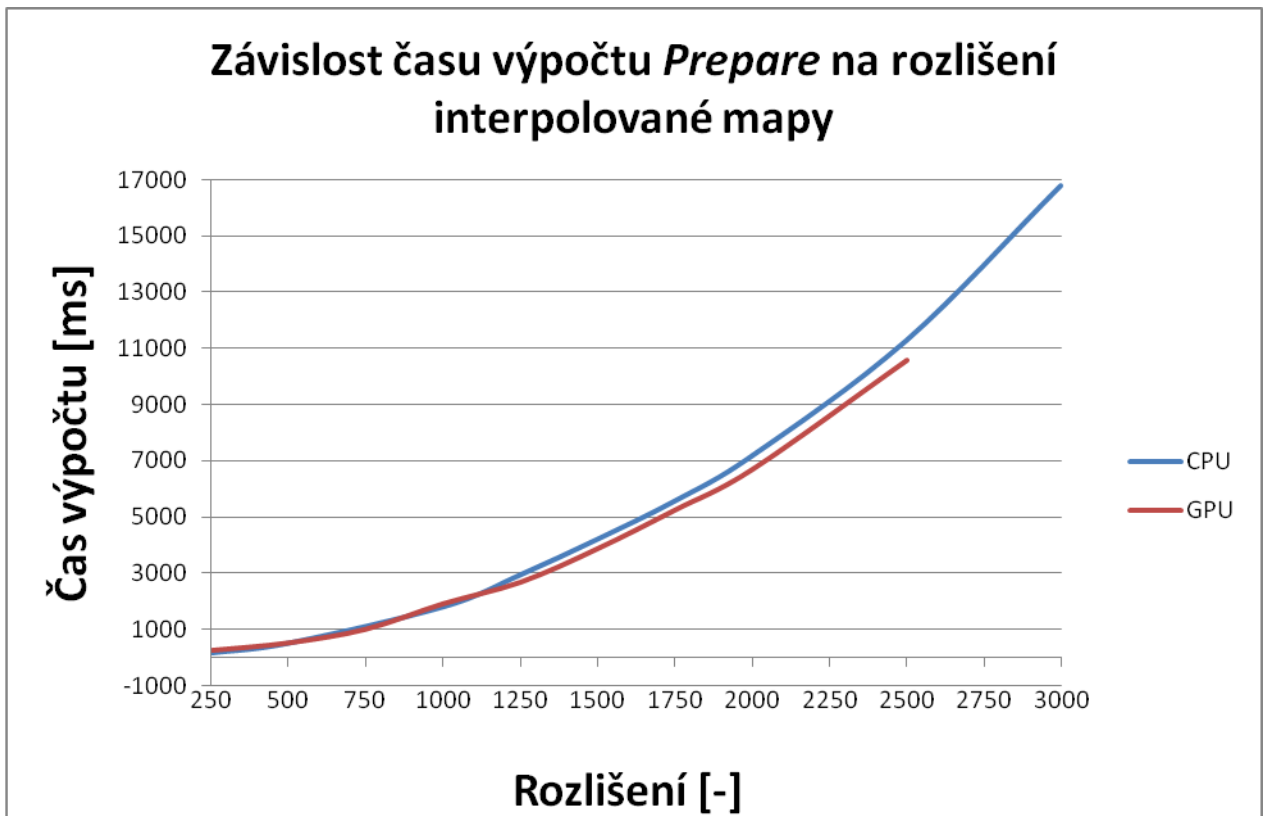


Graf 6.1 Závislost času výpočtu na rozlišení interpolované mapy, PC 1, Sheppardova interpolace, *Prepare*

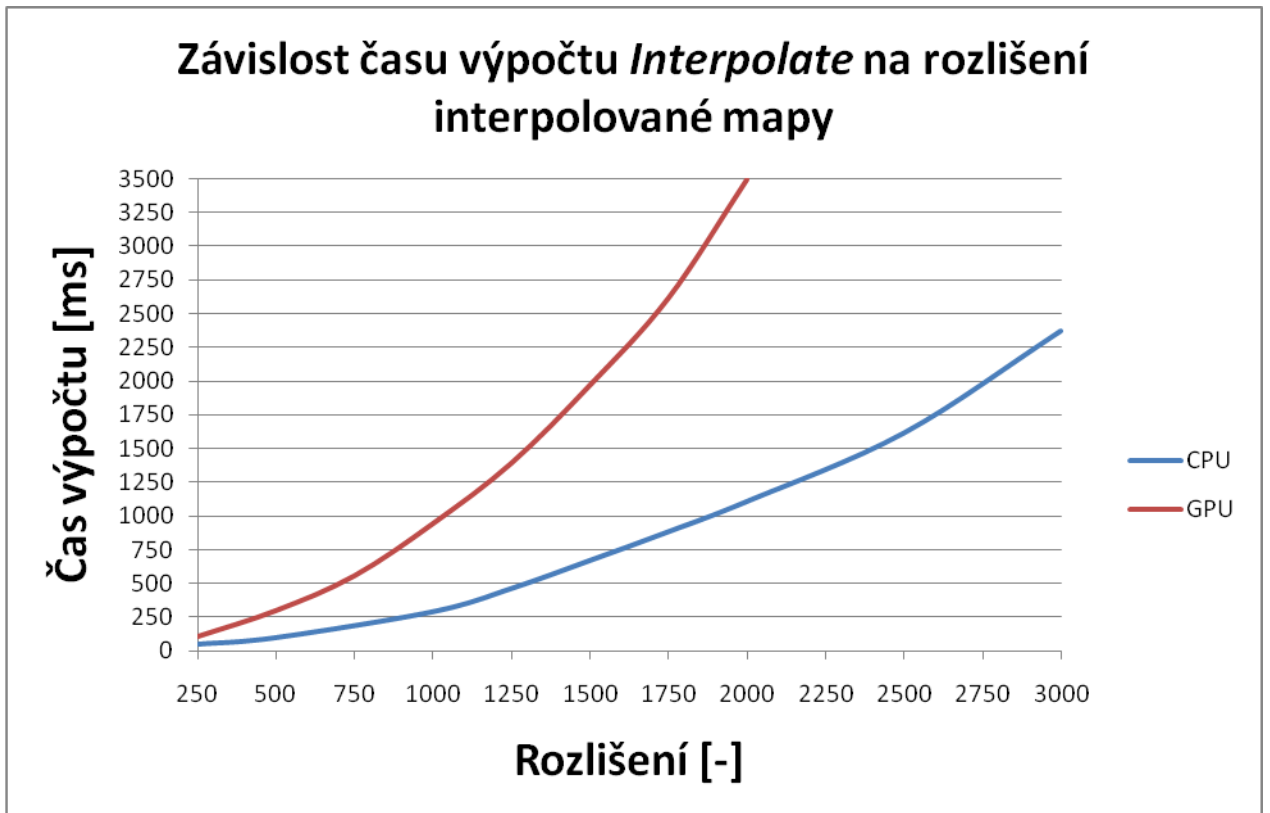


Graf 6.2 Závislost času výpočtu na rozlišení interpolované mapy, PC 1, Sheppardova interpolace, *Interpolate*

Další měření proběhlo na druhé PC soustavě. Při přípravě interpolace (graf 6.3) byl rychlejší grafický procesor cca kolem 300 – 500ms. Při rozlišení větším než 2500 nastal problém s alokací paměti. Při interpolaci (graf 6.4) však byl výrazně rychlejší CPU. Při rozlišení 2000 byl GPU o 2383ms pomalejší, pro větší rozlišení nevystačila paměť.



Graf 6.3 Závislost času výpočtu na rozlišení interpolované mapy, PC 2, Sheppardova interpolace, *Prepare*

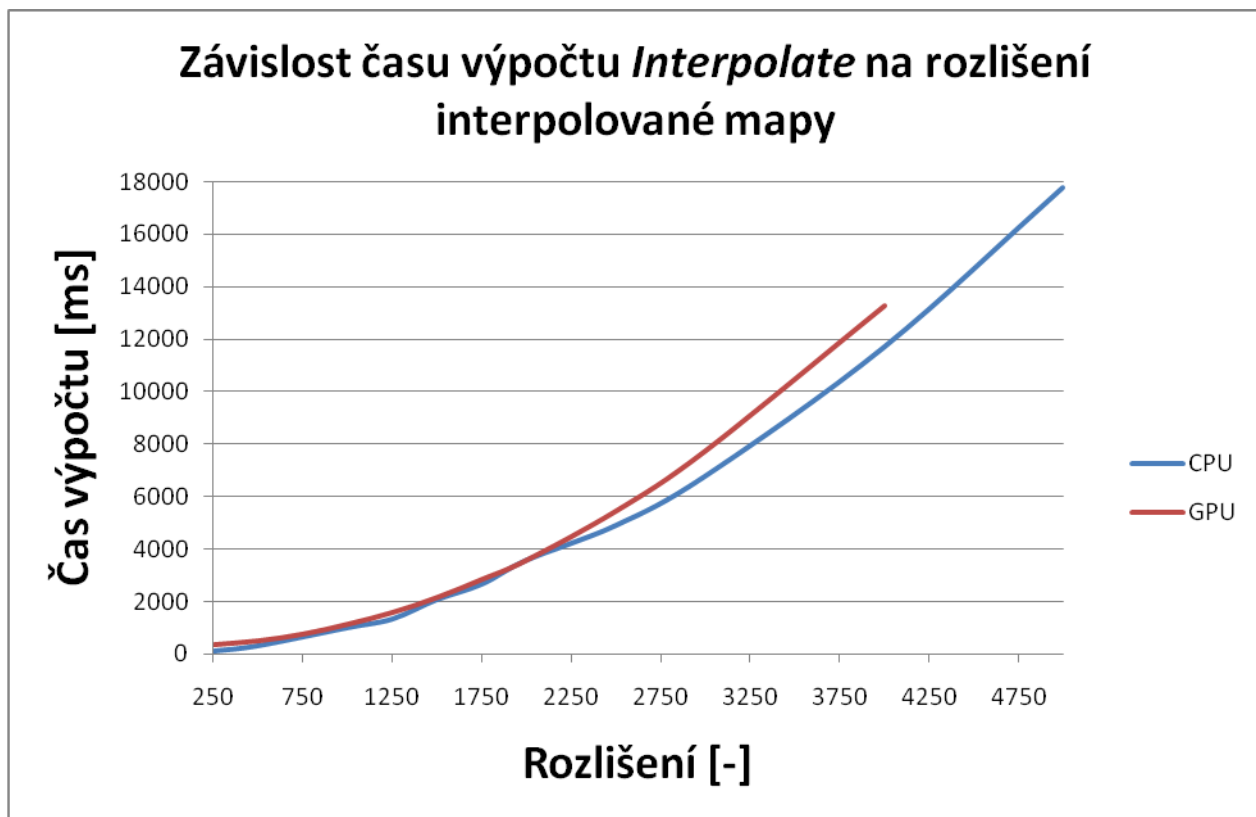


Graf 6.4 Závislost času výpočtu na rozlišení interpolované mapy, PC 2, Sheppardova interpolace, *Interpolate*

Oproti dvoujádrovému procesoru z první sestavy, byl čtyřjádrový procesor při rozlišení 3000 rychlejší v přípravě o celých 50 vteřin a v interpolaci o 8,5 vteřiny. GPU z druhé PC sestavy, byl při rozlišení 1250 v přípravě rychlejší, než GPU z první sestavy o 5,5 vteřiny, v interpolaci o 959ms.

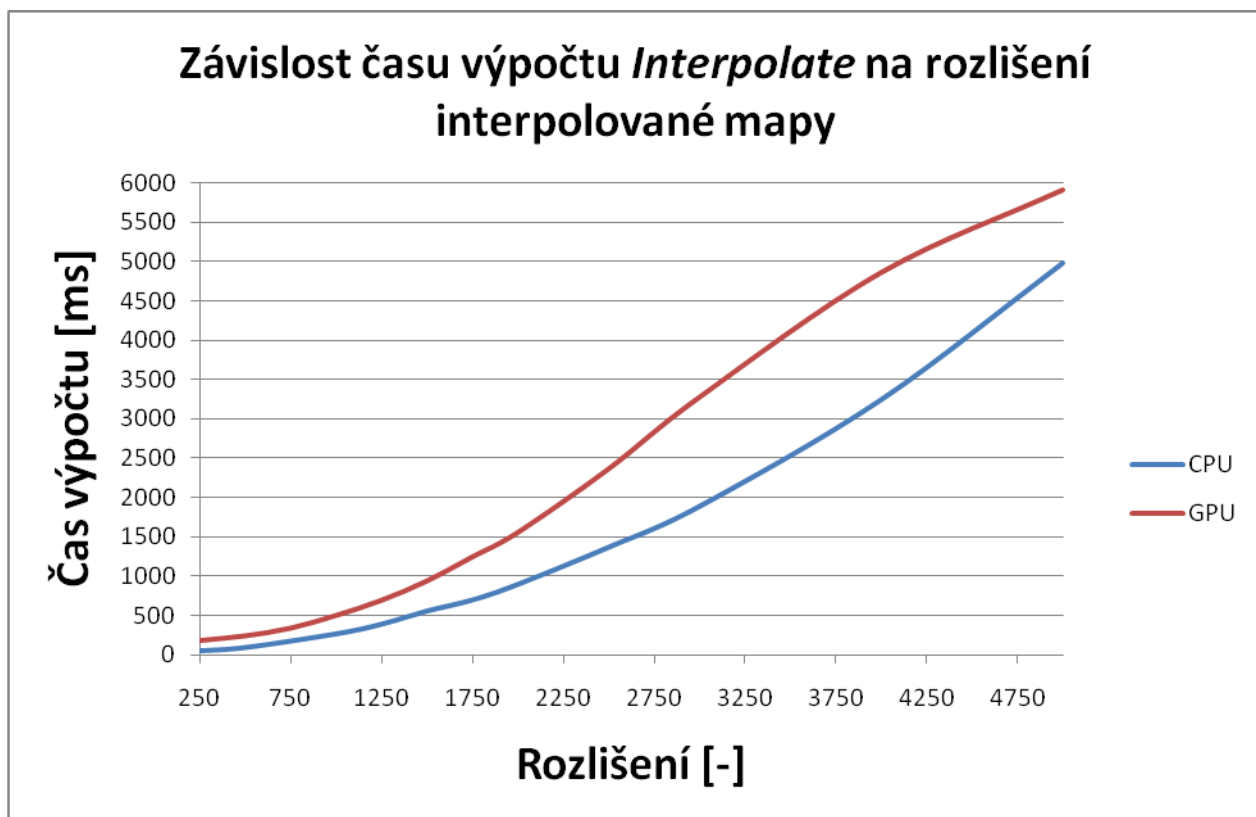
## 6.2 Výsledky měření Hermitovi interpolace

Při Hermitově interpolaci byl měřen pouze výpočetní čas interpolace. Ve funkci *Prepare* probíhají v obou implementacích stejné procesy, které se při dvaceti elektrodách pohybovaly kolem 18ms. Na první PC sestavě (graf 6.5) byl rychlejší CPU. Při rozlišení 4000 byl GPU pomalejší o 1590ms, pro větší rozlišení byl nedostatek paměti.



Graf 6.5 Závislost času výpočtu na rozlišení interpolované mapy, PC 1, Hermitova interpolace, *Interpolate*

U druhé PC sestavy (graf 6.6) byl opět rychlejší CPU, při rozlišení 5000 o 924ms. GPU měl oproti předchozím hodnotám tendenci se zrychlovat, ale pro větší rozlišení se bohužel měření nepodařilo.



Graf 6.1 Závislost času výpočtu na rozlišení interpolované mapy, PC 2, Hermitova interpolace, *Interpolate*

## 7. Shrnutí výsledků

	Rozlišení	Sheppardova Interpolace		Zrychlení		rozlišení	Hermitova Interpolace		Zrychlení
		Prepare	Interpolate	Pre	Int		Interpolate	Int	
CPU1	1250	12296	2134	1,5 GPU1	1,1 CPU1	4000	11705	1,4 CPU1	
GPU1	1250	8170	2348			4000	13295		
CPU2	2000	7158	1108	1,07 GPU2	3,15 CPU2	5000	4981	1,19 CPU2	
GPU2	2000	6695	3491			5000	5905		

Tabulka 7.1 Shrnutí výsledků

Z tabulky můžeme vyčíst, že u Sheppardovi interpolace bylo při paralelním zpracování funkce *Prepare* vždy o trochu rychlejší zpracování na GPU pomocí OpenCL. Naopak u funkce *InterpolateToBitmap* bylo vždy rychlejší zpracování na CPU pomocí knihovny TPL. Nejvyššího zrychlení dosáhl čtyřjádrový procesor s podporou SMT. Byl oproti GPU rychlejší 3,15krát.

## Závěr

Cílem mé práce bylo prozkoumat možnosti paralelního zpracování dat na standardních i grafických procesorech. Nejprve bylo vysvětleno, co jsou obecné výpočty na grafických procesorech. Následně byly uvedeny možnosti programování na GPU a některé programovací prostředky. Blíže byly představeny DirectCompute, CUDA a OpenCL. Dále bylo objasněno k čemu slouží .NET Framework 4 a zejména knihovna TPL v něm obsažená. Poté byly popsány metody Sheppardovi a Hermitovi interpolace, jejich výhody a nevýhody. Obě metody byly implementovány pomocí TPL i OpenCL. U každé implementace byla vysvětlena její funkce, případně problémy s její funkčností. Všechny implementace byly odměřeny, výsledky vyneseny do grafu a porovnány. Nakonec byly výsledky shrnuty ve formě tabulky. Vyšlo najevo, že paralelní zpracování na GPU je výhodnější pro váhování u Sheppardovi interpolace, naopak CPU pro samotné interpolování. Největšího zrychlení bylo dosaženo na procesoru Intel Core i7 860. Při interpolaci dosáhl procesor zrychlení 3,15 oproti GPU. Oproti procesoru Intel Core2Duo 2160 byl ve všech výpočtech téměř čtyřikrát rychlejší.

## Seznam použité literatury

[1] *nVIDIA OpenCL Programming Guide*. Version 3.1 URL:

<[http://developer.download.nvidia.com/compute/cuda/3\\_1/toolkit/docs/NVIDIA\\_OpenCL\\_ProgrammingGuide.pdf](http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_OpenCL_ProgrammingGuide.pdf) >

[2] About the Khronos Group [online]. [cit. 11.8. 2010]. © 2010 Khronos Group. URL:

<<http://www.khronos.org/about>>

[3] Technická podpora Microsoft [online]. [cit. 11.8. 2010]. ©2010 Microsoft Corporation. URL:

<<http://support.microsoft.com/kb/982671>>

[4] MSDN, Parallel Programming in the .NET Framework[online]. [cit. 11.8. 2010].

©2010 Microsoft Corporation. URL:

<<http://msdn.microsoft.com/en-us/library/dd460693.aspx>>

[5] MSDN, Funkční paralelismus (Task Parallel Library)[online]. [cit. 11.8. 2010].

©2010 Microsoft Corporation. URL:

<<http://msdn.microsoft.com/cs-cz/library/dd537609.aspx>>

[6] MSDN, Zpracovávání vyjímek (Task Parallel Library)online]. [cit. 11.8. 2010].

©2010 Microsoft Corporation. URL:

<<http://msdn.microsoft.com/cs-cz/library/dd997415.aspx>>

[7] SHARP, John. *Microsoft Visual C# 2008 - krok za krokem*. Brno: Computer Press, 2008.

ISBN: 978-80-251-2027-9

[8] SELLS, Chris. *C# a WinForms – programování formulářů Windows*. Brno: Zoner Press, 2005.

ISBN: 80-86815-25-0

[9] Khronos OpenCL API Registry [online]. © 2010 Khronos Group. URL:

< <http://www.khronos.org/registry/cl/>>

