

Fakulta elektrotechnická
Katedra elektroniky a informačních technologií

DIPLOMOVÁ PRÁCE

Hardwarová akcelerace klastrování dat z pixelových detektorů
ionizujícího záření

Autor práce: **Bc. Richard Sivera**
Vedoucí práce: **Ing. Petr Burian, Ph.D.**

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta elektrotechnická
Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: **Bc. Richard SIVERA**
Osobní číslo: **E21N0042P**
Studijní program: **N0714A060013 Elektronika a informační technologie**
Specializace: **Elektronika**
Téma práce: **Hardwarová akcelerace klastrování dat z pixelových detektorů ionizujícího záření**
Zadávající katedra: **Katedra elektroniky a informačních technologií**

Zásady pro vypracování

- Seznamte se s principy pixelových detektorů ionizujícího záření a s příslušnou fyzikální instrumentací.
- Prostudujte známé algoritmy pro klastrování pixelových dat. U jednotlivých algoritmů diskutujte jejich možnosti hardwarové akcelerace.
- Vybraný algoritmus (či algoritmy) implementujte v FPGA či v embedded procesoru. Detailně popište proces implementace.
- Otestujte implementovaný algoritmus na reálných datech z pixelových detektorů IZ. Výsledky testů diskutujte a zhodnoťte dosaženou výkonost HW akcelerace.
- Diskutujte začlenění implementovaného algoritmu do stávajících měřicích řetězců.

Rozsah diplomové práce: **40-60**
Rozsah grafických prací:
Forma zpracování diplomové práce: **elektronická**

Seznam doporučené literatury:

1. BURIAN, P., et al. Katherine: Ethernet embedded readout interface for Timepix3. Journal of Instrumentation, 2017, 12.11: C11001.
2. BURIAN, Petr, et al.: Ethernet embedded readout interface for Timepix2 Katherine readout for Timepix2. Journal of Instrumentation, 2020, 15.01: C01037.
3. MEDUNA, Lukáš. Detecting elementary particles with Timepix3 detector. 2019. Diplomová práce.

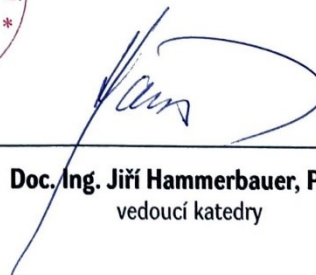
Vedoucí diplomové práce: **Ing. Petr Burian, Ph.D.**
Research and Innovation Centre for Electrical
Engineering

Datum zadání diplomové práce: **6. října 2023**
Termín odevzdání diplomové práce: **24. května 2024**



L.S.

Prof. Ing. Zdeněk Peroutka, Ph.D.
děkan



Doc. Ing. Jiří Hammerbauer, Ph.D.
vedoucí katedry

V Plzni dne 6. října 2023

Abstrakt

Práce se zabývá akcelerací klastrování pixelových dat z detektoru Timepix3 ve vyčítacím zařízení Katherine, což je odlišný přístup od klastrování na PC. Akcelerace klastrování v zařízení snižuje zatížení PC a nároky na propustnost Ethernetové sítě. První část práce je věnována teoretickému rozboru tématiky a řešení dostupných algoritmů. Druhá část práce je věnována optimalizaci vybraného algoritmu, paralelizaci klastrování a implementaci do zařízení Katherine Gen2. Vybraný algoritmus „baseline“ byl výkonově optimalizován o 124,7 %. Následně byl paralelizován až do 16 výpočetních vláken PC, kde vykázal 7x nárůst výkonu. Nakonec byl algoritmus implementován v Katherine na dvoujádrovém procesoru ARM Cortex A9 a akcelerován na jednom výpočetním vlákne. Bylo prezentováno, že algoritmus dosahuje výkonu klastrování v řádu stovek kHit/s. Výstupem implementovaného algoritmu jsou klastry nebo histogramy energií, které jsou odesílány z Katherine do PC přes TCP komunikaci. K tomu byl vytvořen obslužný PC program s GUI, který umožňuje konfiguraci způsobu klastrování a post-processing naměřených dat.

Klíčová slova

Klastrování, Algoritmus, Výkon algoritmu, Pixelová data, Ionizující záření, Timepix3, Pixelový Detektor, Katherine readout, Burdaman, Paralelizace, ARM Cortex A9, FPGA, C++, Qt framework

Abstract

This work describes clustering acceleration of Timepix3 pixel data in readout device Katherine, which is a different approach from clustering in PC. Clustering in the device lowers computational load on the PC and required bandwidth of Ethernet bus. First part of the work is dedicated to theoretical analysis and research of available algorithms. Second part of the work is dedicated to optimization of chosen algorithm, clustering parallelization and implementation of the algorithm in Katherine Gen2. The chosen algorithm “baseline” was optimized by 124,7 % and parallelized into 16 computational threads, which yielded 7x higher clustering performance. Subsequently, the algorithm was implemented in the devices dual core ARM Cortex A9 processor and accelerated in one thread. The implemented algorithm reached maximum performance in the order of hundreds of kHit/s. Resulting clusters and energy histograms are then sent from Katherine to PC via TCP communication. For this, a GUI PC program was created, which enables the user to configure the clustering and post-process the data.

Key Words

Clustering, Algorithm performance, Pixel data, Ionising radiation, Timepix3, Pixel Detector, Katherine readout, Burdaman, Parallelization, ARM Cortex A9, FPGA, C++, Qt framework

Poděkování

Tímto bych rád poděkoval vedoucímu diplomové práce Ing. Petru Burianovi, Ph.D. za zajímavé zadání a za poskytování cenných nápadů a připomínek. Také děkuji za důvěru při vypůjčení potřebné fyzikální instrumentace. V neposlední řadě děkuji své rodině a blízkým za jejich neutuchající podporu.

Obsah

Úvod	- 1 -
1 Problematika detekce ionizujícího záření.....	- 2 -
1.1 Pixelové detektory částic	- 2 -
1.2 Detektor Timepix3.....	- 3 -
1.2.1 Terminologie	- 3 -
1.2.2 Pixelová data	- 4 -
1.3 Měřicí zařízení Katherine	- 5 -
1.4 Kalibrace detektoru.....	- 6 -
2 Klustrování pixelových dat.....	- 8 -
2.1 Teorie klustrování pixelových dat	- 8 -
2.2 Dostupné algoritmy pro klustrování dat	- 9 -
2.2.1 Algoritmus „baseline“	- 10 -
2.2.2 Algoritmus „quadtree“	- 12 -
2.2.3 Algoritmus DBSCAN.....	- 13 -
2.2.4 Algoritmus k-NN.....	- 16 -
2.3 Možnosti hardwarové akcelerace klustrování.....	- 16 -
2.3.1 Akcelerace klustrování v procesoru ARM.....	- 16 -
2.3.2 Akcelerace klustrování v FPGA	- 17 -
2.4 Výběr algoritmu a způsobu akcelerace.....	- 18 -
3 Optimalizace algoritmu a paralelizace klustrování.....	- 19 -
3.1 Volba nástrojů a programovacího jazyka	- 19 -
3.1.1 Testovací sestava	- 19 -
3.2 Architektura testovacího prostředí.....	- 20 -
3.2.1 Přesnost klustrování.....	- 20 -
3.3 Datasetsy pixelových dat pro testování algoritmů	- 21 -
3.4 Optimalizace algoritmu „baseline“	- 22 -
3.4.1 Ukládání klastrů.....	- 22 -
3.4.2 Jednotlivé optimalizace algoritmu.....	- 23 -

3.4.3	Optimalizovaný algoritmus „baseline“	25 -
3.5	Optimalizace algoritmu „quadtree“	28 -
3.5.1	Implementace algoritmu „quadtree“	28 -
3.6	Porovnání výkonu, využití paměti a přesnosti algoritmů	30 -
3.7	Výkon kalibrace energií.....	32 -
3.8	Paralelizace klastrování	32 -
3.8.1	Možnosti paralelizace	33 -
3.8.2	Princip časové paralelizace.....	34 -
3.9	Možnosti paralelizace algoritmu v procesoru.....	37 -
3.10	Možnosti paralelizace algoritmu v FPGA	38 -
4	Implementace v hardware a dosažený výkon klastrování	40 -
4.1	Požadavky na funkce	40 -
4.2	Použité nástroje a jazyky	41 -
4.3	Architektura systému	42 -
4.3.1	Architektura Pluginu	43 -
4.3.2	Architektura Obslužného programu	45 -
4.4	Komunikační protokol.....	47 -
4.5	Implementace klastrování v Katherine	49 -
4.5.1	Maximální dosažitelný výkon klastrování.....	50 -
4.6	Dynamické řízení výkonu klastrování.....	51 -
4.7	Výsledný výkon klastrování	53 -
4.7.1	Nestabilita při vysokých intenzitách IZ.....	54 -
4.7.2	Finální výkon klastrování	55 -
4.8	Začlenění implementovaného algoritmu do stávajících měřících řetězců.....	56 -
4.9	Možnosti budoucího zvýšení výkonu klastrování	57 -
	Zhodnocení a závěr.....	59 -
	Literatura	61 -
	Seznam obrázků.....	I
	Seznam tabulek.....	III
	Příloha A: Zdrojové kódy.....	IV

Příloha B: GUI Obslužného programu	V
Příloha C: Snímek elektrické svorkovnice pořízený režimem Pixel Counting	VI

Seznam symbolů a zkratek

<i>ASIC</i>		<i>Integrovaný obvod pro specifickou aplikaci</i>
<i>closedClusters</i>		<i>Pole kompletních klastrů</i>
<i>CPU</i>		<i>Centrální procesorová jednotka</i>
<i>D_{max}</i>	<i>ns</i>	<i>Maximální zpoždění vyčítací elektroniky</i>
<i>E</i>	<i>eV</i>	<i>Energie částice nebo hitu</i>
<i>FPGA</i>		<i>Programovatelné hradlové pole</i>
<i>GUI</i>		<i>Grafické uživatelské rozhraní</i>
<i>hit</i>		<i>Detekce IZ jedním pixelem detektoru (zásah detektoru)</i>
<i>hitrate</i>	<i>hit/s</i>	<i>Počet hitů za jednotku času (intenzita IZ)</i>
<i>H_{max}</i>	<i>hit/s</i>	<i>Maximální dosažitelný hitrate</i>
<i>IZ</i>		<i>Ionizující záření</i>
<i>LE</i>		<i>Programovatelný logický blok v FPGA</i>
<i>LUT</i>		<i>Vyhledávací tabulka (Look-up-Table)</i>
<i>O</i>		<i>Asymptotická složitost algoritmu</i>
<i>openClusters</i>		<i>Pole nekompletních klastrů</i>
<i>RAM</i>		<i>Paměť s náhodným přístupem</i>
<i>std</i>		<i>STL knihovna C++</i>
<i>ToA</i>	<i>ns</i>	<i>Čas při detekci hitu (Time-of-Arrival)</i>
<i>ToT</i>	<i>ns</i>	<i>Čas uplynutý detekováním hitu (Time-over-Threshold)</i>
<i>Δt</i>	<i>ns</i>	<i>Časové rozpětí klastru</i>

Úvod

Detekce ionizujícího záření je letitý problém, který se zjednodušil s příchodem pixelových detektorů částic jako je Timepix3. Díky těmto detektorům je možno analyzovat jednotlivé částice, jejich energie a čas jejich detekce. Detektory a vyčítací elektronika neustále zrychlují, fyzikální experimenty produkují hodně dat a nastává problém jejich zpracování a ukládání. Doposud byla surová data přenášena do obslužných PC, které zpracovávaly a ukládaly i TB dat. Jedním způsobem řešení tohoto problému je surová pixelová data nepřenášet a neukládat, ale rovnou zpracovávat ve vyčítacím zařízení, např. v Katherine. Proto vznikla tato práce, která si klade za cíl akcelarovat zpracování pixelových dat přímo ve vyčítacím zařízení Katherine. Výzvou tohoto řešení je vytvořit dostatečně rychlý algoritmus pro omezené výpočetní a paměťové prostředky hardware Katherine. Dostupný hardware tvoří dvoujádrový procesor ARM Cortex A9 s přidruženým FPGA a 2 GB paměti RAM.

V práci bude nejprve představena problematika detekce IZ a detektor Timepix3. Také bude představeno vyčítací zařízení Katherine, na kterém bude klastrování akcelarováno.

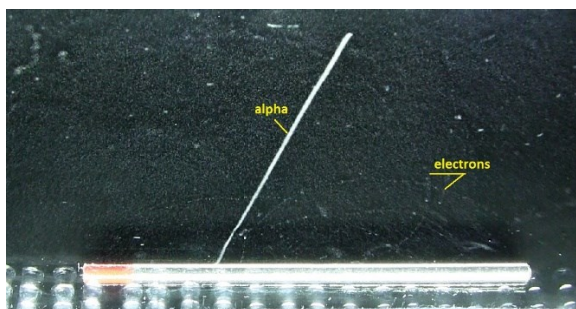
Následně budou podle definice klastrování představeny dva algoritmy „baseline“ a „quadtree“ [4], které budou doplněny o dva algoritmy strojového učení DBSCAN [7][8][10] a k-NN [9]. Bude diskutován jejich výkonový potenciál a vhodnost pro dílčí způsoby hardwarové akcelerace.

Vybrané algoritmy budou dále optimalizovány pro maximální výkon. Algoritmu „baseline“ bude optimalizacemi mnohonásobně zvýšen výkon klastrování. Optimalizovaný algoritmus „quadtree“ navazuje na diplomovou práci Lukáše Meduny [4]. Dále bude diskutována problematika paralelizace klastrování, která se opírá o vlastní experimentálně zjištěné výsledky a teoretické výpočty z diplomové práce Tomáše Čelka [16]. Poté budou představeny možnosti paralelizace klastrování na procesoru a FPGA.

Poslední kapitola popíše implementaci optimalizovaného algoritmu „baseline“ v procesoru zařízení Katherine Gen2 programem napsaném v jazyce C++. Celkem budou vytvořeny dva programy, první pro Katherine a druhý pro PC obsluhu. Bude popsána architektura implementace a budou diskutovány překážky a faktory, které ovlivňují výkon klastrování. Poté bude prakticky změřen finální dosažený výkon klastrování a bude potvrzena jeho přesnost. Závěrem jsou diskutovány široké možnosti využití vytvořeného programu v existujících měřicích řetězcích a budoucí možnosti zvýšení výkonu klastrování.

1 Problematika detekce ionizujícího záření

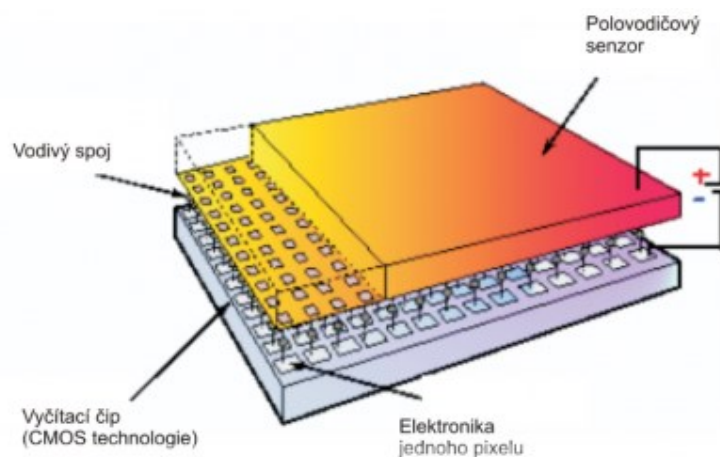
Ionizující záření (IZ) je běžným fyzikálním jevem. Je způsobeno pohybem elementárních částic, jakou jsou například částice Alfa (Obr. 1), Beta, Gama, nebo nabitých iontů. Zvláště Gama záření může být pro člověka škodlivé, zároveň ale může IZ sloužit k vyšetření pacientů pomocí RTG nebo CT, ale také může sloužit ke zkoumání fyzikálních zákonů, k materiálovému výzkumu, léčbě, testování materiálů atd. Jedním ze způsobů, jak IZ detekovat a analyzovat, je použití tzv. pixelových detektorů částic.



Obr. 1: Jedna alfa částice a elektrony zachycené v mlžné komoře [5]

1.1 Pixelové detektory částic

Pixelové detektory detekují částice pomocí pixelové matice podobně, jako CCD snímáče pořizují fotografické snímky. Tyto detektory lze využívat jak pro výzkum, tak pro lékařské účely. Struktura detektoru je popsána na následujícím obrázku. Detektor Timepix3 (v této práci používaný) je hybridní polohově citlivý polovodičový detektor IZ vyvinutý v Evropském centru pro jaderný výzkum CERN v rámci kolaborace Medipix [1].



Obr. 2: Struktura hybridního pixelového detektoru Medipix2, ze stejné rodiny zařízení jako Timepix3 [1]

Na Obr. 2 můžeme vidět dvě hlavní části detektoru: aktivní senzor (nejčastěji křemík) a vyčítací (readout) ASIC (čip). První část, polovodičový senzor (kromě nejčastěji

používaného křemíku se dále používá například CdTe, GaAs, SiC a další), je struktura s maticí 256 x 256 pixelů (případně 128x128). Senzor má vlastnosti jako povrchový PN přechod, ke kterému je přivedeno inverzní předpětí, senzor má tedy v klidovém stavu vyprázdňenou oblast. Vyprázdňená oblast polovodiče poté představuje citlivou část detektoru, kde po dopadu IZ (hitu) vzniká elektrický náboj. Náboj je poté díky předpětí odveden do elektrody příslušného pixelu, kde po zpracování v nábojovém zesilovači vznikne napěťový impuls úměrný velikosti náboje, ten je dále diskretizován a dále zpracován vyčítací elektronikou v pixelu readout čipu. Readout čip obvykle umí měřit délku trvání tohoto impulsu (Time-over-Threshold), čas vzniku události (Time-of-Arrival) či čítač prostý počet pulzů (counting mode). Dřívější generace readout čipů uměly poskytovat jen jeden typ této informace. Detektor Timepix3 je výjimečný v tom, že umí poskytovat informaci o ToT i ToA současně. [1]

1.2 Detektor Timepix3

Jedná se o hybridní pixelový detektor vhodný pro rychlé kontinuální detekování IZ. Jak již bylo zmíněno v předchozí kapitole, detektor se skládá z polovodičového senzoru a vyčítacího ASIC čipu. ASIC je vyroben na 130 nm CMOS technologii a obsahuje přesně 65536 kanálů pro zpracování 256x256 pixelové matice senzoru. Maximální časové rozlišení detektoru je 1,56 ns. [2]

Zařízení je schopno pracovat ve dvou základních readout režimech (modech). První je „data-driven“ mode, který po dopadu částice odešle informaci o ToA a ToT detekovaného hitu (pixel částice). V tomto modu vyčítání funguje kontinuálně a po detekci hitu je ten stejný pixel schopen detekovat další hit po přibližně $ToT + 475$ ns. Výhodou tohoto módu je právě kontinuální detekce dopadlých částic a možnost vyčítat až v rychlosti 40 MHit/s/cm². Druhým režimem je „frame-based“ mode, který vyčte stav všech pixelů v jednom časovém okamžiku a odešle ho jako snímek. V tomto případě může Timepix3 vyčítat až 1300 snímků za sekundu. [2]

1.2.1 Terminologie

Kolem detektorů částic se často používá anglická terminologie. Pro upřesnění definujeme pojmy *hit* a *hitrate*.

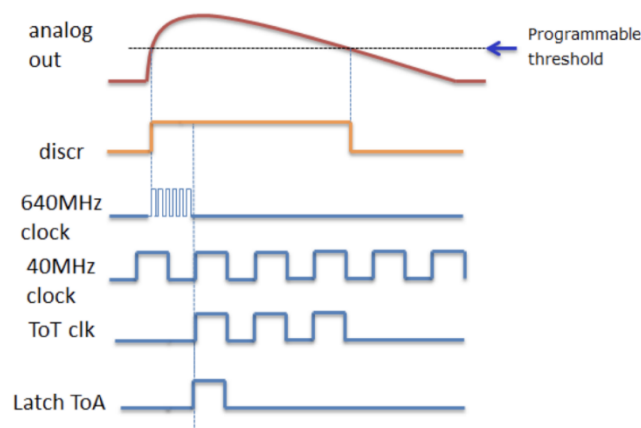
Hit je záznam dopadlé částice na jednom pixelu detektoru (zásah pixelu). Pokud způsobí jedna částice zásah 5 pixelů detektoru naráz, mluvíme o záznamu pěti hitů, i když jde zároveň o jednu částici. Po záznamu hitu vyčítací elektronikou je nutné jej převést na

PixelHit (zkr. pixel). Při zpracování pixelových dat (např. klastrování) znamená zpracování hitu nebo pixelu totéž.

Hitrate představuje intenzitu záznamu hitů detektorem za čas a jeho jednotkou je *hit/s*. Často se používá pro popsání maximální rychlosti (nebo výkonu algoritmu), jakou je detektor nebo elektronika schopen detekovat dopadlé hity. Pokud tedy říkáme, že Timepix3 může detekovat maximálně 40 MHit/s/cm², znamená to, že maximální hitrate na jednom centimetru čtverečním detektoru je 40 milionů hitů za sekundu. Později v této práci bude prezentován maximální hitrate (výkon) dosažený navrženým algoritmem klastrování.

1.2.2 Pixelová data

Pixelová data jsou výstupem pixelového detektoru. Jak již bylo zmíněno, detektor Timepix3 je schopen měřit dvě základní informace, a to *ToT* a *ToA*. Tyto veličiny je detektor schopen měřit současně. Na Obr. 3 je vidět, jak se pomocí dvou hodinových signálů o frekvenci 40 MHz a 640 MHz určuje *ToT* a *ToA*.



Obr. 3: Průběh měření detektoru [4]

ToA (Time-of-arrival) je čas měřený v nanosekundách, který určuje čas dopadu částice na detektor s rozlišením 1,56 ns. Rozlišení je dáno frekvencí hodinových signálů. Čas dopadu částice je určen v čase, kdy napěťová úroveň na analogovém výstupu pixelu překročí prahovou úroveň (threshold). Po překročení prahové úrovně jsou spuštěny 640 MHz hodiny, jež jsou zastaveny v momentě náběžné hrany 40 MHz hodin. Počet hodinových cyklů 640 MHz hodin se nazývá *fineToA* (jemné *ToA*). Počet hodinových cyklů 40 MHz hodin od konce 640 MHz hodin až do poklesu analogového výstupu pod prahovou hodnotu se nazývá *coarseToA* (hrubé *ToA*). [4]

Pomocí *fineToA* a *coarseToA* lze poté vypočítat výsledný čas *ToA* (1.1):

$$ToA = coarseToA \cdot 25 - fineToA \cdot 1,5625 \text{ (ns)}, \quad (1.1)$$

kde konstanta 25 je perioda 40 MHz hodin v ns a 1,5625 je perioda 640 MHz hodin v ns.

ToT (Time-over-Threshold) je čas strávený detekováním hitu (přímo úměrný energii hitu), který je definován jako počet hodinových cyklů 40 MHz hodin od detekce překročení prahové hodnoty (Threshold) až do poklesu analogového výstupu pod prahovou hodnotu. Pro přepočítání *ToT* na *E* hitu se používá kalibrační přepočítání (kapitola 1.4).

Spolu s *ToT* a *ToA* je výstupem detektoru ještě informace o souřadnicích zasaženého pixelu v kartézském souřadnicovém systému, a to souřadnice *x* (0 až 255) a *y* (0 až 255). Dohromady jsou data (*x*, *y*, *ToT* a *ToA*) nazývána jako tzv. *pixelová data*.

1.3 Měřicí zařízení Katherine

Pro detekci IZ detektorem Timepix3 je nutné mít zařízení, které výstupní signály detektoru zpracuje a převede je do čitelné podoby. Tzv. vyčítacích (readout) zařízení je na výběr více, v této diplomové práci bude používáno zařízení Katherine.



Obr. 4: Vyčítací zařízení Katherine Gen1 s připojeným Timepix3 detektorem nalevo

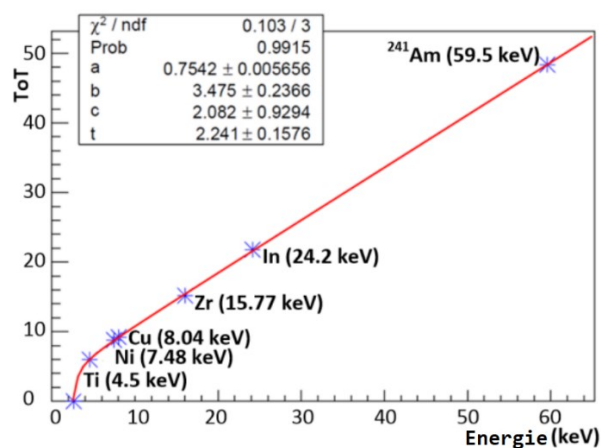
Katherine je zařízení navržené pro měření s připojením k PC. Výhodně lze využívat více zařízení zároveň pro měření IZ na zkušebním paprsku (nebo jiném experimentu). Měřicí řetězec se sestává z detektoru Timepix3 připojeného k zařízení Katherine (viz Obr. 4), které je následně přes Ethernet připojeno ke vzdálenému obslužnému PC. Na přední straně zařízení je konektor VHDCI s 68 piny sloužící k připojení detektoru Timepix3, zatímco LEMO konektor slouží k připojení předpětí (tzv. bias voltage) v rozmezí -300 V až +300 V. Z druhé strany je vidět 1 Gbit Ethernet konektor, DC napájení a GPIO piny pro dodatečné funkce. Díky gigabitovému Ethernetu je maximální možná propustnost sítě stanovena na ekvivalentní hitrate 16 MHit/s. [2]

Hardware Katherine obsahuje FPGA SoC, který je složen z FPGA s 8000 LE a dvoujádrový procesor ARM Cortex-A9 s 1 či 2 GB DDR3 RAM. FPGA implementuje gigabitový Ethernet, vyčítání dat z detektoru, veškeré zpracování dat a ukládání dat do bufferů. Na procesoru běží pod operačním systémem Linux řídicí program, který implementuje řízení logiky v FPGA a výsledné odeslání dat z bufferů na Ethernet k obslužnému PC. V PC lze poté přes program Burdaman zapínat či vypínat měření, kontrolovat stav Katherine nebo nastavovat předpětí detektoru, to vše přes připojení peer-to-peer UDP a daný komunikační protokol. Program také implementuje kalibraci energií pomocí kalibračních souborů, které kompenzují interní nedokonalosti detektoru (např. rozdílná citlivost pixelů). Burdaman také umožňuje nejrůznější post-processing naměřených dat, jako např. klastrování dat, prohlížení částic, nebo zobrazení histogramu energií částic. [2]

Je nutné zmínit, že tato diplomová práce se bude zabývat akcelerací klastrování dat na novější verzi zařízení Katherine Gen2. Druhá generace zařízení se od původní Katherine liší přidáním druhého VHDCI konektoru pro druhý detektor, zvětšení RAM na 2 GB DDR3, a přidáním rozhraní USB 3.0 doplňujícím původní gigabitový ethernet [3]. Druhá generace Katherine se liší ve více ohledech, ale zmíněné rozdíly jsou pro účely této diplomové práce nejzásadnější.

1.4 Kalibrace detektoru

Pro přesný výpočet energie dopadlé částice z ToT , ale také kompenzaci chyb a nehomogenity pixelů v matici detektoru je nutné provést kalibraci detektoru.



Obr. 5: Závislost ToT na energii (kalibrační křivka) pro jeden pixel detektoru [2]

Provedení kalibrace spočívá v ozařování detektoru charakteristickým rentgenovým zářením různých prvků (indium, zirkonium, měď atd.) a radioaktivním izotopem

²⁴¹Am [2]. Ze znalosti známých energií prvků je poté sestavena závislost známé energie na ToT . Z grafu (Obr. 5) jsou poté určeny kalibrační konstanty pro každý pixel, které lze poté použít pro přepočítání ToT na energii v keV (1.2). Jedná se o aproximaci ToT na energii, nikoliv přesný výpočet.

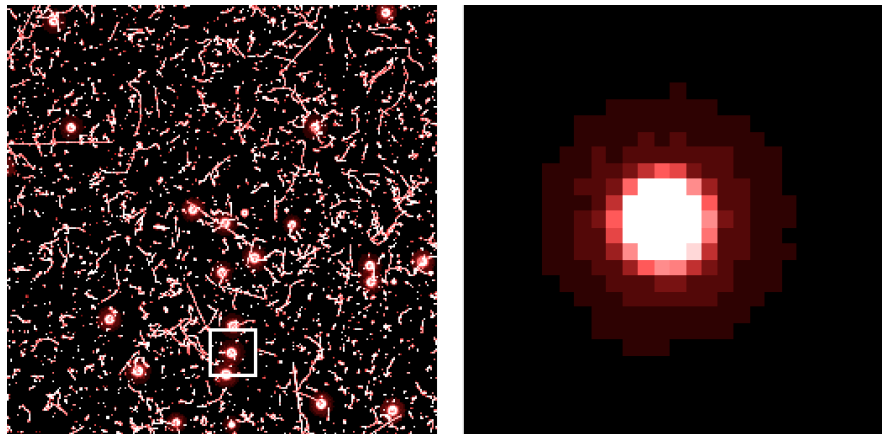
$$ToT = a \cdot E + b - \frac{c}{E - t} \quad (-), \quad (1.2)$$

kde a je první kalibrační konstanta, b je druhá kalibrační konstanta, c je třetí kalibrační konstanta, E je energie hitu v keV, a t je korekce prahové hodnoty pixelu.

Nutno dodat, že kalibrační konstanty z výpočtu (1.2) jsou pro každý pixel jiné a vycházejí z kalibračních křivek (Obr. 5) pro každý pixel. To znamená, že detektor o velikosti 256x256 pixelů musí mít kalibrační soubor obsahující matici 256x256 kalibračních konstant pro každý druh konstanty.

2 Klastrování pixelových dat

Pixelová data z detektoru Timepix3 je pro analýzu částic nutno zpracovat klastrováním. Pixelová data jsou výstupem detektoru a nesou informace o detekované částici: souřadnice pixelu; ToT; čas dopadu ToA. Jedna detekovaná částice obvykle (neplatí však vždy) aktivuje více pixelů (viz Obr. 6), a tak je výsledný záchyt částice definován skupinou pixelů. Klastrování pixelových dat je proces, který jednotlivé aktivní pixely seskupí do skupin, které jsou definovány prostorovou a časovou blízkostí.



Obr. 6: Zachycené částice na celé ploše detektoru (nalevo) a jedna vyznačená alfa částice (klastr) po dokončení klastrování dat (napravo)

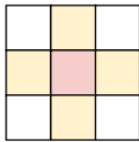
Na Obr. 6 vidíme, jak mohou být všechna naměřená pixelová data (nalevo) po klastrování rozdělena na jednotlivé částice (napravo), které nazýváme klastry pixelů. Tento konkrétní klastr má velikost 201 pixelů (hitů) s celkovou energií po kalibraci 4,2 MeV. Celkově se ve snímku nalevo ukrývá 2570 klastrů/detekovaných částic. Při větším množství dopadlých částic je pro online analýzy (jako v případě této práce) nezbytně nutné, aby byl algoritmus klastrování dostatečně rychlý. V této kapitole budou prozkoumány známé algoritmy pro klastrování dat a budou analyzovány jejich možnosti a složitost implementace pro akceleraci v Katherine.

2.1 Teorie klastrování pixelových dat

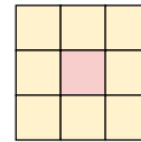
Vstupem algoritmu pro klastrování jsou pixelová data neboli hity z detektoru. Můžeme mít tedy data o tisících pixelů s jejich naměřenými parametry (x ; y ; ToT ; ToA), a úkolem klastrování je určit, zdali existují nějaké další pixely, tvořící dohromady s daným pixelem jeden klastr. Pixel musí v tomto případě splnit dvě podmínky, aby mohl být považován za součást nějakého klastru. Podmínky jsou definovány následovně.

- **Podmínka sousedství pixelů:**

Aby mohly dva pixely tvořit jeden klastr, musí se v matici detektoru 256x256 jednat o sousední pixely. Sousední pixely mohou být definované jako skupina 4 sousedních pixelů v horizontálním a vertikálním směru, nebo jako 8 sousedních pixelů, které mohou být navíc sousední také v diagonálním směru.



(a) 4-neighbors pixels



(b) 8-neighbors pixels

Obr. 7: Typy sousedních pixelů. Hlavní pixel je červený, zatímco sousední pixely jsou žluté [4]

- **Podmínka časová:**

Časová podmínka určuje maximální možné časové rozpětí (časovou deltu) jednoho klastru. Jelikož jednotlivé pixely obsažené v klastru mohou být zaznamenány v různých časových okamžicích (různé ToA), je nutné určit maximální možné časové rozpětí. Časové rozpětí klastru Δt je definováno od pixelu p_1 , který má nejnižší ToA , po pixel p' , který má největší ToA [4]. Pokud je pixel p' uvnitř okna Δt dle (2.1), splnil časovou podmínku. Běžně může být Δt klastru rovno 200 ns. Svoji podstatou je Δt způsobeno propagací náboje ve hmotě detektoru po dopadu částice, nebo nelinearitou vstupních zesilovačů.

$$|ToA_{p_1} - ToA_{p'}| < \Delta t, \quad (2.1)$$

kde ToA_{p_1} je ToA pixelu s nejnižším ToA v klastru, $ToA_{p'}$ je ToA kontrolovaného pixelu na časovou podmínku, a Δt je nastavené maximální časové rozpětí klastru.

Pro zařazení pixelu do klastru je nutné splnění sousedské i časové podmínky zároveň.

2.2 Dostupné algoritmy pro klastrování dat

V této kapitole budou představeny některé z existujících algoritmů pro klastrování. Kapitoly 2.2.1 a 2.2.2 představí algoritmy klastrování přímo podle definice klastrování, zatímco kapitoly 2.2.3 a 2.2.4 představí alternativní řešení klastrování pomocí algoritmů strojového učení.

2.2.1 Algoritmus „baseline“

První možností klastrování dat je jednoduchý algoritmus „baseline“, který vychází přímo z definice klastrování. Algoritmus postupně rozděluje pixely do pole klastrů *openClusters* (pole nekompletních klastrů), a když je to možné odešle kompletní klastry z *openClusters* do *closedClusters* (pole kompletních klastrů). Algoritmus je definován následovně:

- Načíst nový pixel;
- Načíst jeden klastř C z pole *openClusters*;
- Odeslat kompletní klastř C do *closedClusters*;
- Kontrola časové a sousedské podmínky s každým pixelem v klastř C ;
- Pokud jsou splněny obě podmínky, zařadit pixel do klastř C ;
- Pokud není splněná alespoň jedna z podmínek, opakovat od druhého bodu;
- Pokud nebyl pixel zařazen do *openClusters*, vytvořit nový klastř.

V algoritmu se může vyskytnout několik speciálních situací.

- **První speciální situací** je, že pixel splnil obě podmínky pro více klastrů zároveň. To znamená, že více klastrů společně s pixelem tvoří dohromady jeden klastř. Je nutné klastry spojit do jednoho;
- **Druhou speciální situací** je latence vyčítací elektroniky, která může způsobit, že dva pixely se stejným ToA budou zpracovány v jiný okamžik. Např. pixely seřazené podle ToA : p_1, p_2, p_3 mohou být zaznamenány v pořadí: p_2, p_3, p_1 , protože pixel p_1 byl zaznamenán s velkou latencí. V takovém případě je nutno definovat maximální zpoždění pixelu předtím, než můžeme klastř prohlásit za kompletní. Maximální zpoždění D_{max} může být v tomto případě např. 200 us [4], ale závisí na mnoha faktorech při měření (např. velikost dopadlé částice) a musí být nastavitelný obsluhou. Pro prohlášení klastř za kompletní je tedy nutné, aby maximální ToA obsažené v klastř C bylo o D_{max} menší, než je ToA aktuálně přijímaného pixelu (viz Algoritmus 1, řádek 6);
- **Třetí speciální situací** je, že dvě částice při dopadu na detektor dopadnou ve stejném časovém okně Δt a navzájem se překryjí (splní časovou a sousedskou podmínku). V tomto případě nelze částice jednoduše odlišit a dojde v algoritmu k nepřesnosti. Řešením tohoto problému by mohlo být použití algoritmu strojového učení KD-means [6], který by byl schopen při správných hyperparametrech rozlišit

dva překrývající se klastry, ale naopak by neměl vysokou průměrnou přesnost vyplývající z jeho nedeterministické povahy.

Celý algoritmus včetně řešení první a druhé speciální situace je následovný.

Algoritmus 1: Klastrování „baseline“

Vstup: pixel p ; pole nekompletních klastrů $openClusters$ (obsahující klastry C); D_{max}

Výstup: pole kompletních klastrů $closedClusters$ (obsahující klastry C_k)

```
1:   přidán ← False
2:   if ( $openClusters = 0$ )
3:        $openClusters \leftarrow VytvořNovýKlastr(p)$ 
4:   else
5:       foreach (klastr  $C$  in  $openClusters$ )
6:           if ( $p.ToA - C.ToA > D_{max}$ )
7:                $closedClusters \leftarrow OdešliKompletníKlastr(C)$ 
8:               continue
9:           foreach (pixel  $pix$  in  $C$ )
10:              if ( $SplněnéPodmínky(pix, p)$ )
11:                  if (přidán)
12:                       $SpojKlastry(posledníKlastr, C)$ 
13:                       $openClusters = openClusters - C$ 
14:                  else
15:                       $C \leftarrow p$ 
16:                      přidán ← true
17:                       $posledníKlastr \leftarrow C$ 
18:              if (přidán = false)
19:                   $openClusters \leftarrow VytvořNovýKlastr(p)$ 
```

Algoritmus 1 obsahuje všechny dříve zmíněné operace nezbytné pro klastrování pixelových dat. Na řádce „6“ můžeme vidět dříve zmíněnou druhou situaci pro kompenzaci zpoždění elektroniky, kde můžeme klastr C prohlásit za kompletní a přemístit ho do $closedClusters$. Na řádce „10“ dále probíhá kontrola časové a sousedské podmínky ve funkci $SplněnéPodmínky(C, p)$, která porovnává data všech pixelů klastru C s novým pixelem p . Pokud byl pixel do nějakého klastru přidán („15“), je ukazatel na tento klastr uložen do $posledníKlastr$. Pokud pixel následně splní podmínky i s dalším klastrem (první speciální situace), je klastr C spojený (merge) do klastru $posledníKlastr$ („12“), načež C je následně odstraněn z $openClusters$. Pokud nebyl pixel p přidán do žádného klastru, je do $openClusters$ vložen nový klastr funkcí $VytvořNovýKlastr(p)$.

Popsaný algoritmus je pro malý počet klastrů výpočetně nenáročný a přesný. Algoritmus chybí pouze v případě třetí speciální situace. Nevýhody tohoto algoritmu nastanou v případě velkého počtu klastrů *openClusters*, kdy bude algoritmus procházet každý pixel každého klastru (dvě smyčky foreach) pro kontrolu podmínek. Dvě vnořené smyčky jsou časově a výpočetně náročné ($O(n^2)$).

2.2.2 Algoritmus „quadtree“

Algoritmus původně navržený v diplomové práci [4] má základy v algoritmu „baseline“, navíc implementuje základní principy „quadtree“ neboli kvadrantového stromu. Rozdíl mezi těmito algoritmy není ve hledání sousedů, ale ve způsobu ukládání a prohledávání klastrů. Kvadrantový strom je datová struktura s efektivním ukládáním bodů v 2D prostoru, hojně je využívána v 2D grafických programech a počítačových hrách [4].

Kvadrantové stromy rekurzivně rozdělují 2D prostor do čtyř kvadrantů. V první řadě máme kořen stromu, který představuje celou 2D plochu. Kořen má potom 4 potomky (kvadranty), které se nazývají uzly. Také každý uzel může mít své vlastní 4 potomky. Takto je strom definován rekurzivně až po nejmenší uzel, který se nazývá list. Do listů se poté ukládají pixely. Pro jednoduchost se dá říct, že list zabírá plochu 1x1 pixelu. Struktura kvadrantového stromu je názorně zobrazena na Obr. 8. [4]

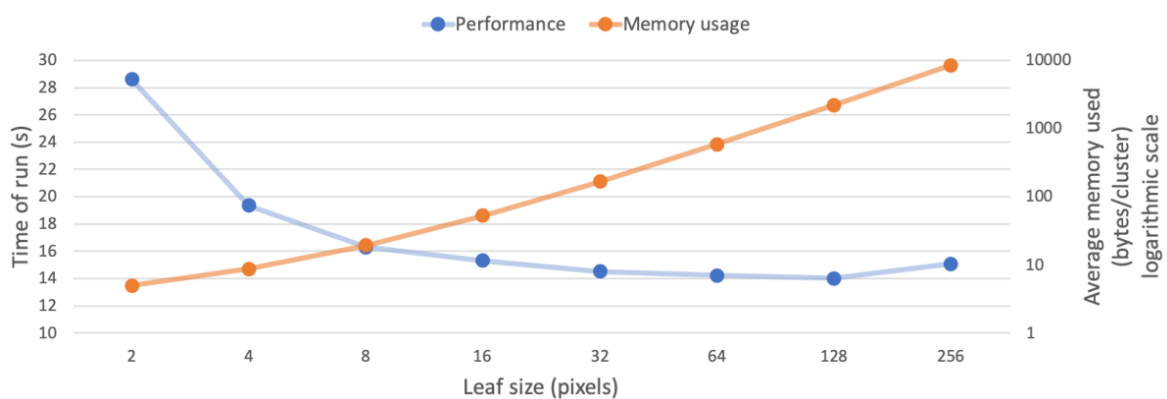
Rekurze ve stromu je definována následovně:

- Rozdělení aktuální plochy do kvadrantů na 4 potomky (uzly);
- Obsahuje-li uzel v sobě pixel, je v něm vytvořen další potomek;
- Rekurzivní vytváření potomků až do definované hloubky, kde je vytvořen poslední potomek stromu (list), do kterého je pixel uložen [4].



Obr. 8: Znárodnění rekurzivního vytváření stromu s velikostí listu 32x32 pix [4]

Autor diplomové práce [4] pomocí popsaného algoritmu vytvořil ukládání možných sousedů pixelů do kvadrantového stromu *neighbors*, se kterými je následně porovnáván na podmínky sousedství každý nový pixel. Díky hledání ve stromové struktuře je efektivně a rychle zkontrolován každý klastr na sousedské podmínky. Odpadá tím nutnost použití druhé smyčky `foreach` (Algoritmus 1 řádek „9“), která je efektivně nahrazena rychlým prohledáváním pole *neighbors*. Tím je kompenzována nevýhoda algoritmu „baseline“, kde při velkém počtu klastrů *openClusters* s velkým množstvím pixelů užíváme velkou část výpočetních prostředků jejich prohledáváním. Asymptotická složitost hledání n sousedů v „quadtree“ je $O(n \log(n))$ [9][16], namísto $O(n^2)$ pro algoritmus „baseline“. Tato výhoda je u algoritmu „quadtree“ vykoupena většími nároky na paměť a složitějším algoritmem hledání sousedů. Nevýhodou je také nutnost přesouvat klastry z nižších uzlů do vyšších v případě, že velikost klastru přesáhla velikost původního uzlu. Část výpočetních prostředků je tedy tracena na správu stromové struktury.



Obr. 9: Závislost výkonu/paměti na velikosti listu [4]

Z Obr. 9 můžeme vidět, že nejkratší čas klastrování (největší výkon) byl dosažen s velikostí listu 128x128 pixelů. Autor práce [4] nakonec zvolil velikost listu 16x16 pixelů jako kompromis mezi rychlostí a využitím paměti. Je nutno podotknout, že práce se zaměřovala na klastrování dat přímo v PC. Náš případ je jiný, jelikož v zařízení Katherine pracujeme s omezenými paměťovými prostředky (1 či 2 GB DDR3), malé velikosti listu by nepřipadaly v úvahu. Výhody a nevýhody „quadtree“ vs „baseline“ budou porovnány dále viz Kapitola 3.

2.2.3 Algoritmus DBSCAN

Prozatím byly představeny algoritmy založené na kontinuálním rozdělování pixelů do jednotlivých klastrů. Existují však také tzv. hierarchické algoritmy, které rozdělí pixely do jednotlivých klastrů z kompletních naměřených dat naráz. Takovým algoritmem je

DBSCAN [7], v celém anglickém znění „Density-based spatial clustering of applications with noise“. Tento algoritmus provádí klastrování pixelů na základě hustoty bodů v prostoru s možností detekce šumu. [7]

V případě DBSCANu záleží na tom, co považujeme za šum. Reálně je šum přítomen při měření vysokoenergetických částic a zpravidla se nachází po okrajích detektoru Timepix3. Takový šum lze lehce odfiltrovat podmínkou pro filtrování pixelů po okrajích detektoru. Naproti tomu DBSCAN filtruje šum podle velikosti klastru, což není vhodné, protože mnohdy jsou předmětem zájmu právě klastry malé velikosti (1 až 4 pixely). Parametrem určujícím filtrování šumu bude dále zmíněný *minPts*.

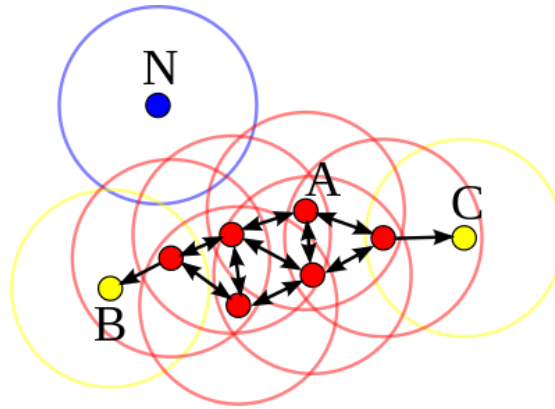
DBSCAN byl vyvinut pro klastrování s následujícími požadavky:

- Minimální znalost oboru, ze kterého pocházejí měřená data;
- Nalezení klastrů libovolného tvaru;
- Nevadí nedeterministické výsledky;
- Vysoká efektivita pro velké množství dat [7].

Z požadavků, pro které byl DBSCAN navržen je vidět, že kvality algoritmu nedokážeme v našem případě plně ocenit. První dva požadavky už splňují algoritmy „baseline“ a „quadtree“, čtvrtý požadavek je ale výhodou. Zatímco pro algoritmus „baseline“ je časová náročnost hledání sousedů přímo úměrná počtu přichozích pixelů na druhou, pro DBSCAN je tato úměra příznivější $O(n \log(n))$ [7].

Nyní popíšeme základní myšlenku algoritmu. Nechť ε je parametr specifikující rádius jednoho pixelu (v našem případě roven 1 pixelu) a nechť *minPts* je minimální počet pixelů v klastru. S těmito definicemi je sousedství definováno následovně:

- Pixel p je *středový pixel* klastru, pokud je v jeho okolí alespoň *minPts* *bezprostředních pixelů* v maximální vzdálenosti ε ;
- Pixel q je *bezprostřední pixel*, pokud se nachází v maximální vzdálenosti ε od *středového pixelu*;
- Pixel q je *dosažitelný pixel*, pokud se nachází v maximální vzdálenosti ε od jakéhokoliv pixelu, který je součástí klastru;
- Všechny ostatní pixely, které se nenachází v maximální vzdálenosti ε od jakéhokoliv bodu libovolného klastru, jsou buď *jedno-pixelové klastry* nebo *šum* [7].

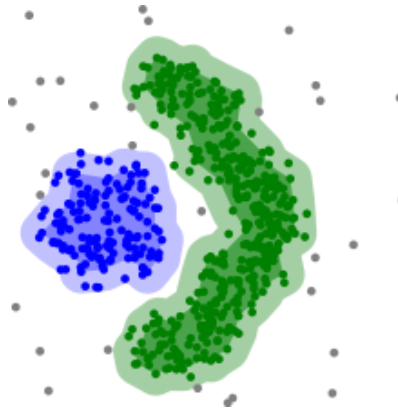


Obr. 10: Znázornění definice sousedství. Každý bod má definovaný rádius ϵ . Body A jsou tzv. *bezprostředními pixely* a tvoří jádro klastru. Body B a C jsou tzv. *dosažitelnými pixely* a stále jsou součástí klastru s body A. Bod N je *šum*, protože se nenachází v blízkosti žádného jiného bodu [8]

Z definice sousedství je zřejmé, že algoritmus DBSCAN definuje pouze podmínku sousedství, zatímco na podmínku časovou zapomíná. Nicméně algoritmus by mohl být vhodný pro rychlé klastrování s omezenou přesností, pokud byl parametr ϵ nastaven na větší hodnotu než 1 pixel. To by mohlo být vhodné pro velké částice jako např. částice alfa (viz Obr. 6). Zjednodušený algoritmus je poté následovný:

- Najít pixely ve vzdálenosti max. ϵ pro každý pixel;
- Identifikovat *středové pixely* které mají alespoň *minPts* sousedů;
- Najít všechna *jádra klastrů* které splňují podmínky sousedství a ignorovat pixely které nejsou v bezprostřední blízkosti *jádra klastrů*;
- Rozřadit všechny zbývající pixely buď k jednotlivým klastrům (pokud splňují podmínku vzdálenosti), nebo je označit jako šum [7].

DBSCAN má asymptotickou složitost hledání n sousedů jen $O(n \log(n))$ [7], namísto $O(n^2)$ pro algoritmus „baseline“. Algoritmus má však svoje nevýhody, v podobě složitějšího algoritmu a definici šumu, která není vhodná pro všechna měření. Přesnost algoritmu je také nižší. Zatímco „baseline“ hledá sousedy podle známé definice vycházející ze znalosti funkce detektoru, DBSCAN nezná časovou podmínku a aproximuje neznámá data podle parametrů a je nedeterministický [8]. Nevýhodou je také nutnost provádět algoritmus na kompletních pixelových datech. Kontinuální klastrování pixel po pixelu algoritmem DBSCAN by se muselo provádět v předem určených časových oknech, což by zvýšilo nároky na paměť. Názorný výsledek klastrování algoritmem DBSCAN je na Obr. 11.



Obr. 11: Příklad výsledku algoritmu DBSCAN na smyšlených datech se dvěma klastry a šumem [8]

2.2.4 Algoritmus k-NN

Algoritmus k-nejbližších sousedů (dále k-NN), je jednoduchý algoritmus pro klastrování nebo kategorizaci dat. Využívá se zejména na poli strojového učení, kde dokáže rozeznávat vzorce v neznámých datech.

Vstupním parametrem algoritmu jsou vstupní pixelová data a poté parametr k . Parametr k značí počet nejbližších pixelů k vyhodnocení podmínky sousedství. V algoritmu je používán k rozhodování, do kterého klastru pixel p náleží. Algoritmus pixel p přiřadí do klastru, který má největší počet pixelů v k nejbližších sousedech pixelu p . [9]

Nevýhodami k-NN je v našem případě fakt, že parametr k zásadně určuje přesnost klastrování [9], a že algoritmus nebere v úvahu maximální povolenou vzdálenost 1 pixelu pro slnění podmínky sousedství, a nezná časovou podmínku. k-NN je také náročný na paměť a s rostoucím počtem dat rychle klesá jeho rychlost [9]. Zmíněné negativní vlastnosti algoritmu jsou zejména nevhodné pro akceleraci na zařízení Katherine. Existuje však mnoho evolucí a vylepšení základního k-NN algoritmu, které jsou blíže popsány v článku [9]. Algoritmus je ze své podstaty nepřesný a nebudeme se jím dále zabývat.

2.3 Možnosti hardwarové akcelerace klastrování

Máme možnost akcelerace na dvoujádrovém procesoru ARM Cortex-A9 nebo v FPGA. V této kapitole budou diskutovány možnosti akcelerace klastrování na těchto výpočetních jednotkách.

2.3.1 Akcelerace klastrování v procesoru ARM

Akcelerace na procesoru dává velkou volnost ve výběru algoritmu a programovacího jazyka, protože na zařízení běží operační systém Linux. Akcelerace na procesoru má také výhodu v jednoduchosti, jelikož z programu lze pixelová data přímo přijímat a data po

klastrování lze odesílat do obslužného PC přes vhodný komunikační protokol. Výhodou je také jednoduchá implementace jakéhokoliv z diskutovaných algoritmů.

Procesor má omezené prostředky. Dvě jádra běžící na frekvenci 1,1 GHz neumožňují velkou paralelizaci klastrování, v tom případě dává smysl klastrování provádět v maximálně dvou souběžných vláknech. Paměť 2 GB DDR3 v Katherine Gen2 omezuje maximální možný počet bufferovaných klastrů. Klastry musejí být bufferovány po minimální čas D_{max} do pole *openClusters* (Algoritmus 1 řádek „7“). Nejdříve po tomto čase můžou být klastry odeslány po Ethernetu do obslužného PC. Maximální hitrate daný velikostí paměti bude:

$$H_{max} = \frac{P}{C_{Bytes} \cdot t_o} \text{ (Hit/s)}, \quad (2.1)$$

kde H_{max} je maximální hitrate, P je velikost dostupné paměti RAM pro ukládání hitů do klastrů, C_{Bytes} je velikost jednoho klastru s jedním hitem v bajtech, a t_o je perioda odesílání kompletních klastrů do PC.

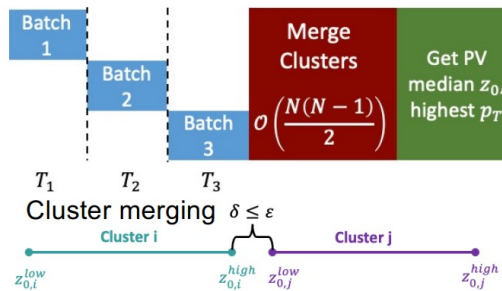
Z výpočtu je vidět, že pro dosažení vysokého maximálního hitrate je třeba mít krátkou periodu odesílání klastrů do PC (nemůže být kratší než D_{max}). Pokud budeme brát v úvahu paměť 2 GB (ignorujeme režii OS a běh ostatních programů) a velikost jednoho klastru s jedním pixellem 36 B, bude při odesílání kompletních klastrů každých 100 ms H_{max} roven přibližně 500 MHit/s. Vzhledem k omezeným výpočetním prostředkům procesoru je jasné, že limitujícím faktorem výkonu klastrování bude spíše procesor než paměť. Nicméně režie ostatních programů a samotného operačního systému dostupnou paměť a tedy H_{max} snižují.

2.3.2 Akcelerace klastrování v FPGA

Největší výhodou akcelerace na hradlovém poli je zejména možnost masivní paralelizace a velkého výkonu, pokud je k dispozici dostatečný počet LE. Nevýhodou je omezený počet dostupných LE, které jsou v FPGA zároveň využívány pro kompletní zpracování dat z Timepix3, odesílání dat do PC a gigabitový Ethernet. Každá úprava algoritmu vede ke zdlohavému procesu implementace a testování v FPGA v porovnání s úpravou kódu běžícím v procesoru. Velkou nevýhodou je také nutnost ukládání dočasných pixelových dat do bufferů. Při vysokých intenzitách IZ by buffery musely být úměrné hitrate, což by představovalo velké nároky na embedded paměť v FPGA.

Výběr algoritmu je také omezený. Zatímco jednoduchý algoritmus jako „baseline“ by mohl být do FPGA poměrně jednoduše implementován, složitější algoritmy jako je „quadtree“

by zabíraly více LE a umožnily by menší paralelizaci. FPGA tedy nedává takovou volnost ve výběru algoritmu jako akcelerace klastrování na procesoru, ale teoreticky umožňuje vyšší rychlost při velké paralelizaci. Přesto je možné implementovat do FPGA i složitý algoritmus viz Obr. 12. Bloky „Batch“ si lze představit jako přijaté hity v časových oknech, na kterých se paralelně provádí akcelerace klastrování. Po dokončení klastrování je nutné klastry z paralelních vláken spojit (Merge) [10].



Obr. 12: Příklad architektury algoritmu DBSCAN v FPGA [10]

2.4 Výběr algoritmu a způsobu akcelerace

Pro účely diplomové práce klastrování pixelových dat bude vhodnější akcelarovat klastrování na procesoru ARM z důvodu omezených prostředků FPGA. Nevýhody implementace do FPGA, např. limitovaný počet LE, velké provázání s existující logikou v FPGA a složitější debuggování, jsou pro účel prototypování a ladění klastrování příliš velké. Procesor naproti tomu umožňuje rychlé prototypování různých algoritmů, volný výběr typu algoritmu a také výborné možnosti debuggování. Proto se dále v práci budeme zabývat výhradně implementací algoritmu do procesoru, přičemž možnost implementace v FPGA bude jen diskutována.

Dále v práci budou blíže prozkoumány algoritmy „baseline“ a „quadtree“, které naproti ostatním zmíněným algoritmům mají deterministické výsledky a vysokou přesnost.

3 Optimalizace algoritmu a paralelizace klastrování

V této kapitole budou porovnány algoritmy „baseline“ a „quadtree“. Bude vytvořeno testovací prostředí pro výkonnostní testování a ladění algoritmu pro dosažení co nejvyššího výkonu klastrování, aby byl optimalizovaný algoritmus připraven na implementaci do Katherine. Důraz bude kladen na přesnost algoritmu a kalibraci energií. Nakonec budou prezentovány možnosti paralelizace algoritmu na procesoru a FPGA.

3.1 Volba nástrojů a programovacího jazyka

Pro vytvoření prostředí pro testování a ladění algoritmu na PC je nutné vybrat vhodné nástroje. Volba programovacího jazyku padla z důvodu rychlosti na jazyky C/C++. Byl vybrán jazyk C++ pro svoje bohaté STL knihovny s garantovanou bezpečností výjimek, nebo použití šablon (ang. template). C++ také umožňuje snadné vytvoření GUI pomocí dostupných frameworků. Volba dialektu jazyka byla závislá na podporovaných dialektech kompilátoru ARM, který bude při implementaci do embedded procesoru použit. Momentálně ARM kompilátor nemá plnou podporu všech dialektů C++ ani knihoven, proto byl zvolen nejlépe podporovaný dialekt C++14, který implementuje většinu knihoven STL [11]. Jediným nedostatkem je, že ARM kompilátor implementuje podporu vláken a paralelizace `std::thread` ve verzi alfa.

Dále bylo nutné zvolit mezi konzolovým, nebo GUI programem. Prostor pro testování s vlastním GUI je výhodné pro snadnou vizualizaci pixelových dat, klastrů, výpisu jejich vlastností a snadné orientaci v informacích. Proto byl zvolen Qt framework [12], který umožňuje jednoduchou tvorbu GUI, programování v C++ a rozličné možnosti zobrazování grafů pomocí `QtCharts`. Framework je multiplatformní a široce rozšířený.

3.1.1 Testovací sestava

Pro testování a ladění výkonu algoritmu byl použit notebook **Lenovo Ideapad 5 Pro 16ACH6**. Procesor s 16 vlákny a TDP 45 W bude vhodný jak na jedno-vláknové, tak paralelní testy výkonu. Zařízení má dimenzované chlazení pro stálou zátěž.

Tabulka 1: Specifikace testovací sestavy

CPU:	AMD Ryzen 7 5800H (8 jader, 16 vláken, 4,4 GHz, 16 MB L3 cache, 45 W TDP)
RAM:	16 GB DDR4, 2 kanály, 3200 MHz
OS:	Windows 10 Home

3.2 Architektura testovacího prostředí

Testovací prostředí musí co možná nejlépe simulovat reálné měření na zařízení Katherine. Klastrování na PC je ale rozdílné od klastrování při reálném měření viz Tabulka 2.

Tabulka 2: Rozdíly v klastrování na Katherine vs PC

	Zařízení Katherine	PC
Vstupní data	Přijímána kontinuálně v náhodných časech během měření	Pixelový .txt soubor reálného měření načten naráz z disku
Zpracování vstupních dat	Příjem zprávy o hitu, převedení hitu na pixel se všemi přepočty	Parsování čísel z textového souboru, výpočty souřadnic a ToA
Klastrování	Totožný algoritmus pro obě zařízení	
Finalizace	Odeslání kompletních klastrů do PC	Zobrazení výsledků v GUI

Klastrování bude provedeno jedno-vláknově v pipeline (Obr. 13) pro izolaci samotného algoritmu klastrování od ostatních úkonů pro přesné měření výkonu (popř. kalibrace energie). Výkon bude měřen pomocí knihovny `std::chrono::high_resolution_clock` pro vysokou přesnost měření času. Pro výpočet dosaženého výkonu (hitrate) v MHit/s stačí čítat počet zpracovaných hitů, který je následně vydělen časem stráveném klastrováním.



Obr. 13: Pipeline testovacího prostředí pro měření výkonu klastrování

Po načtení pixelového souboru a parsování textových dat na hity proběhne volitelná kalibrace energie podle (1.2) a poté samotný algoritmus klastrování. Ze změřeného času klastrování bude vypočten výsledný výkon klastrování. Výsledné klastry budou zobrazeny v GUI pro vizuální kontrolu s možností řazení podle velikosti klastru nebo *ToA*.

3.2.1 Přesnost klastrování

Přesnost algoritmu definujeme jako počet nalezených klastrů algoritmem klastrování ke skutečnému počtu klastrů. Při tisících přijatých klastrů z jednoho měření nelze získat skutečný počet klastrů jinak než vizuální kontrolou, protože měření může být zatíženo šumem a překryvem částic. Jako přibližný skutečný počet klastrů poslouží výsledný počet klastrů po provedení klastrování v programu Burdaman [12] (program vyvinutý na ZČU pro

ovládání Katherine a klastrování pixelových dat na PC). Výpočet přesnosti výsledků je následovný:

$$Acc = \frac{n_B - |n_B - n_A|}{n_B} \cdot 100 (\%), \quad (3.1)$$

kde Acc je přesnost výsledků klastrování, n_A je počet klastrů po provedení klastrování naším algoritmem, a n_B je počet klastrů nalezený programem Burdaman.

3.3 Datasets pixelových dat pro testování algoritmů

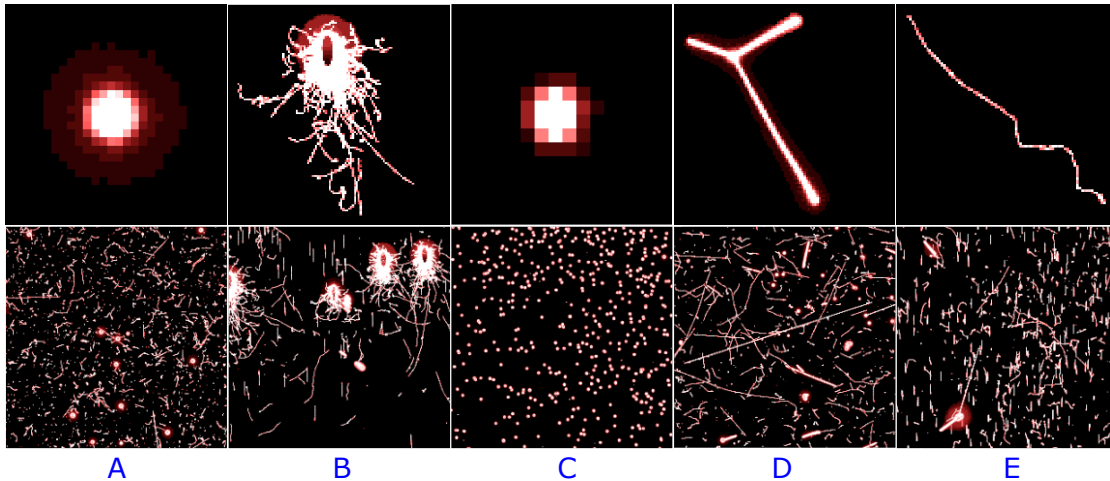
Pro porovnání výkonu jednotlivých algoritmů budou použity datasety obsahující pixelová data co možná nejrůznějších typů. Při praktickém měření IZ jsou radiační pole různá a tomu odpovídají také různé typy částic. Pro účely měření rychlosti algoritmu klastrování je tedy nutné mít data s různými typy částic. Pixelová data se odlišují např.:

- Průměrný hitrate (intenzita IZ);
- Typem detekovaných částic (ovlivněno zdrojem IZ při měření);
- Velikostí a charakterem detekovaných klastrů;
- Energií detekovaných částic;
- Úhlem dopadu částic na detektor;
- Δt klastru.

Kompletní seznam pixelových dat pro testování a porovnávání výkonu je v následující tabulce. Data jsou co nejrozmanitější pro co největší otestování algoritmů.

Tabulka 3: Datasets pro testování algoritmů. Typy částic: A: Mix; B: Ionty olova s úhlem dopadu 60°; C: Protony; D: Mix; E: Elektrony s úhlem dopadu 30°. Velikost klastrů a ToT jsou uváděny ve formátu maximální hodnota / aritmetický průměr

Dataset	Vel. klastrů (pix) (max/avg)		Hitrate (Hit/s)	Hit (MHit)	ToT (max/avg)		Lokalita
A	214	6	1	0,037	409	17	Doma
B	5695	27	90 000	5,449	1022	33	Cern
C	420	11	790 000	7,914	565	34	Cern
D	2063	13	20 000	234,291	980	20	Cern
E	1281	14	2 110	3,168	466	20	Cern



Obr. 14: Snímky datasetů pro testování. Nahoře typické klastry, dole ukázkové snímky dat

Pro referenci byla data klastrována také v programu Burdaman, s jehož výsledky budou navržené algoritmy porovnávány výpočtem (3.1).

Tabulka 4: Výsledky klastrování datasetů pomocí programu Burdaman

Dataset	Čas klastrování (s)	Počet klastrů
A	1	2 764
B	82	202 811
C	35	700 661
D	1 099	17 259 396
E	15	218 844

3.4 Optimalizace algoritmu „baseline“

Základní algoritmus pro klastrování „baseline“ byl popsán v kapitole 2.2.1. Tato kapitola definovaný algoritmus optimalizuje pro zvýšení výkonu algoritmu. Také bude definován způsob ukládání klastrů a budou vyhodnoceny výsledky jednotlivých optimalizací.

3.4.1 Ukládání klastrů

Definujeme, jakým způsobem budou klastry ukládány. Klastry musí být ukládány ve dvou etapách, do polí *openClusters* (nekompletní klastry) a *closedClusters* (kompletní klastry). Pro ukládání dat definujeme následující datové struktury:

- **OnePixel** (**x**, **y**, **ToT**, **ToA**) ukládá základní informace detekovaného pixelu. Datový typ pro **x** a **y** byl zvolen dvoubytový typ *uint16_t* pro budoucí větší detektory. Dále definujeme **ToT** jako *uint32_t* a **ToA** jako *double*;
- **OneCluster** (**vector<OnePixel>**, **minToA**, **maxToA**, **xMax**, **xMin**, **yMax**, **yMin**) ukládá jeden klastř pixelů. Ukládá pixely klastru do pole *std::vector* z STL

knihovny C++. Proměnné s koncovkou „ToA“ typu *double* ukládají nejvyšší a nejnižší *ToA* všech pixelů klastru. To je výhodné pro kontrolu časové podmínky viz 2.1. Proměnné s předložkou *x* nebo *y* typu *uint16_t* ukládají hranice klastru;

- **Clusters (vector<OneCluster>)** ukládá všechny klastry do jednoho pole. Tento typ se používá pro ukládání klastrů *openClusters* a *closedClusters*.

Využití pole *std::vector* je výhodné, protože umožňuje dynamicky alokovat pole na pozadí a ukládat jakoukoliv datovou strukturu/typ pomocí šablon. Statická alokace paměti pro ukládání klastrů by sice umožnila rychlejší prohledávání a kontrolování podmínek, ale nebylo by možné alokovat dostatek paměti pro velké množství klastrů.

3.4.2 Jednotlivé optimalizace algoritmu

Něž přejdeme k optimalizacím algoritmu, uvedeme počáteční výkon základního algoritmu „baseline“. Ten bude změřen v testovacím prostředí na představených datasetech.

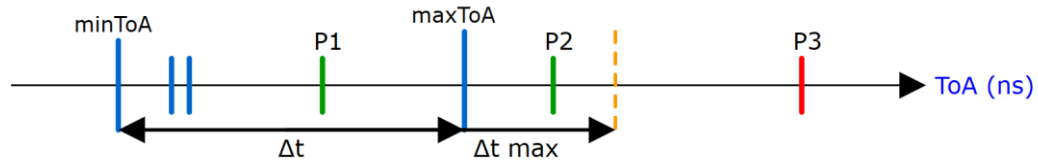
Tabulka 5: Výkon základního algoritmu „baseline“ (aritmetický průměr z 5 měření)

Dataset	A	B	C	D	E
Čas (ms)	4	19 475	34 192	14 931	544

Výkon byl změřen s optimalizacemi kompilátoru (MSVC) pro maximální rychlost algoritmu */O2* a prioritou rychlosti algoritmu před velikostí programu */Ot*. Z výsledků je zřejmé, že počet hitů v datasetu není jediným kritériem pro rychlost algoritmu. Např. čas klastrování datasetu C je cca dvakrát delší než klastrování datasetu D, přestože dataset C obsahuje přibližně 29x méně hitů. To znamená, že důležitou roli hraje typ částic a jejich velikost.

- **1. optimalizace: Vyjmutí časové podmínky**

Algoritmus je ve své základní podobě neefektivní. Algoritmus 1 na řádku „10“ kontroluje podmínky sousedství pro všechny pixely klastrů *openClusters*, přestože to není třeba. Procházení všech pixelů v tisících dynamicky alokovaných polích je výpočetně náročné, proto by bylo výhodné časovou podmínku kontrolovat jen jednou pro celý klaster. Toho lze docílit za pomoci ukládání *minToA* a *maxToA* celého klastru (**OneCluster**). Při nesplnění podmínky se vyhneme výpočetně náročnému kontrolování časové podmínky všech pixelů daného klastru viz Obr. 15.



Obr. 15: Znázornění pixelů na časové ose pro kontrolu časové podmínky klastru. Modré pixely jsou součástí otevřeného klastru. Δt je časová hranice klastru. Pixely $P1$ a $P2$ časovou podmínku splní, zatímco pixel $P3$ nikoliv.

Na Obr. 15 lze vidět, že v případě kontroly časové podmínky pro $P3$ výrazně ušetříme čas, protože $P3$ nebude kontrolován na podmínku sousedství. Pixel $P1$ a $P2$ by časovou podmínku splnil, ale přesto po úspěšné kontrole bude muset být zkontrolován každý pixel celého klastru na podmínky sousedství. Důležitý je čas Δt_{max} , podle kterého je rozhodováno o vynechání kontroly sousedství. Aplikací první optimalizace tedy ušetříme čas v případech, kdy pixel jednoznačně nemůže splnit časovou podmínku pro celý klaster. Výsledky vylepšeného algoritmu jsou v následující tabulce.

Tabulka 6: Výkon po 1. optimalizaci algoritmu (aritmetický průměr z 5 měření)

Dataset	A	B	C	D	E
Čas (ms)	4	16 101	3 050	13 271	539
Zrychlení (%)	0	17	91	11	1

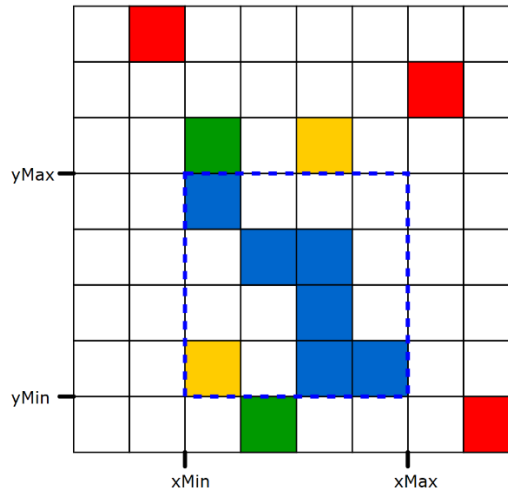
Nejvyššího zrychlení bylo dosaženo logicky u datasetu C, který má nejvyšší hitrate, a tedy nejvyšší časovou hustotu hitů.

• 2. optimalizace: Rozdělení sousedské podmínky

Druhou logickou optimalizací je rozdělením podmínky sousedství. Stejně jako $minToa$ a $maxToa$ v případě časové podmínky lze ukládat také 2D **hranice klastru**: max. a min. souřadnice klastru. K tomu slouží proměnné $xMax$, $xMin$, $yMax$ a $yMin$.

Kontrola sousedství se rozdělí na dvě etapy:

- V první etapě je pixel kontrolován vůči **hranicím klastru** a je rozhodnuto, zda se může stát součástí klastru či nikoliv (Obr. 16);
- V druhé etapě, pokud pixel prošel první podmínkou, je na sousedské podmínky finálně kontrolován se všemi pixely kontrolovaného klastru.



Obr. 16: Znáornění pixelů v 2D prostoru pro kontrolu podmínky sousedství. Modré pixely jsou součástí otevřeného klastru. Zelené pixely podmínku plní a mohou se stát součástí klastru. Žluté pixely podmínku plní, ale nemohou se stát součástí klastru. Červené pixely podmínku neplní

Z Obr. 16 je zřejmé, že představená optimalizace v určitých případech zvýší výkon klastrování. Pro *červený pixel* ke zrychlení dojde, protože bude stačit jedna kontrola sousedství vůči hranicím klastru abychom věděli, že pixel nemůže být jeho součástí. Pro *žluté pixely* ale zrychlení nepřijde, první etapa je podmínkou nutnou ale nepostačující, po úspěšné kontrole vůči hranicím klastru dále musí být pixel projít druhou etapou. *Žluté pixely* tedy prvotní podmínkou sousedství projdou, ale při kontrole vůči všem pixelům klastru je zřejmé, že s žádným nesousedí. Naopak *zelené pixely* prvotní podmínkou i následnou podmínkou projdou. Výkon klastrování s první a druhou optimalizací je následující.

Tabulka 7: Výkon po 2. optimalizaci algoritmu (aritmetický průměr z 5 měření)

Dataset	A	B	C	D	E
Čas (ms)	3	5 760	2 911	11 260	456
Zrychlení (%)	33	64	5	15	16

3.4.3 Optimalizovaný algoritmus „baseline“

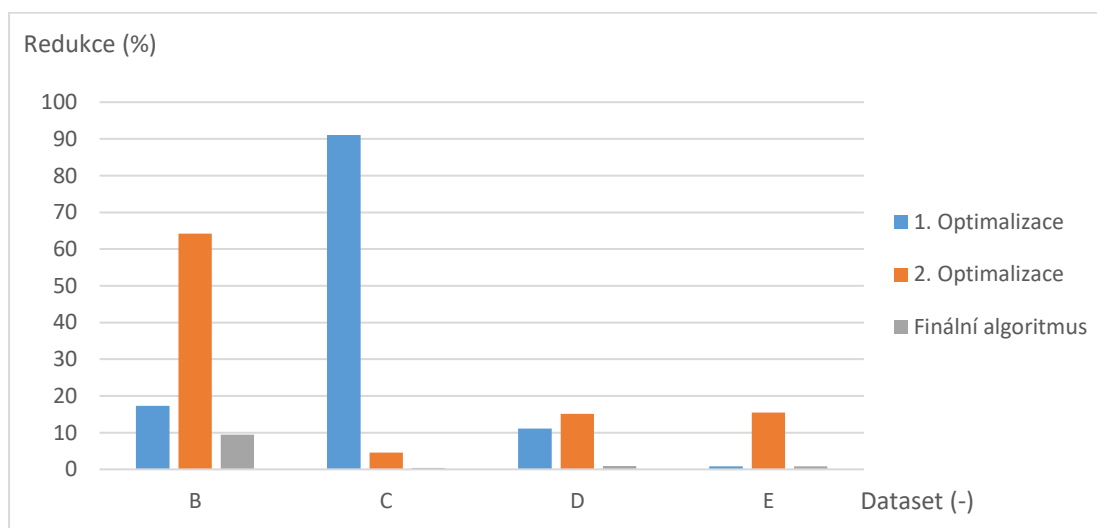
Aplikací představených optimalizací algoritmu došlo k výraznému nárůstu výkonu klastrování všech datasetů. Mimo představené optimalizace ještě došlo na optimalizaci sémantiky ukládání pixelů do klastru. V C++ mohou být pixely do klastru buď uloženy jako kopie a původní pixely smazány, nebo mohou být od C++11 pixely přesunuty bez vytváření kopie pomocí funkce `std::move` [14]. Tato funkce kompilátoru indikuje, že data

jednoho pixelu už nebudou po použití *std::move* dále potřeba a je tedy možné je přesunout do cílového klastru v paměti. Výsledný výkon je v následující tabulce.

Tabulka 8: Finální optimalizace algoritmu pomocí *std::move* (aritmetický průměr z 5 měření)

Dataset	A	B	C	D	E
Čas (ms)	3	5 216	2 744	11 156	452
Zrychlení (%)	0	9	0,4	1	1
Kumulativní zrychlení (%)	33	272	1 083	33	24
Výkon (MHit/s)	5,3	1,0	2,7	5,7	7,0

V následujícím grafu lze vidět procentuální redukce výpočetního času klastrování pro jednotlivé datasety. Dataset A byl z porovnání vyřazen, neboť ani finální redukce času o 1 ms není prakticky měřitelná a prokazatelná hodnota.



Obr. 17: Procentuální redukce času klastrování pro jednotlivé datasety oproti předchozí verzi algoritmu

Procentuální redukce času klastrování se u všech datasetů různí (Obr. 17). Největší redukci lze sledovat u datasetů B a C, které mají ze všech dat nejvyšší hitrate. Vysoký hitrate znamená časově hustě koncentrované hity, kde pomůže nejvíce 1. optimalizace pomocí vyjmutí časové podmínky. Toho je důkazem redukce času klastrování datasetu C s nejvyšším průměrným hitrate 0,79 MHit/s ze všech dat. Dataset B obsahuje největší klastry s maximální velikostí 5695 pixelů, kde došlo k největšímu posunu pomocí 2. optimalizace.

Celkové kumulativní zrychlení algoritmu (Tabulka 8) ukazuje velkou závislost výkonu na typu dat. Na základě výsledků z Obr. 17 lze prohlásit, že pro dosažení vysokých hitrate

u malých klastrů je nejdůležitější první optimalizace. Druhá optimalizace urychluje algoritmus v případě velkých klastrů. Finální algoritmus je následující.

Algoritmus 2: Optimalizovaný algoritmus „baseline“

Vstup: pixel p ; pole *openClusters* (obsahující klastry C); max. zpoždění klastru D_{max}

Výstup: pole *closedClusters* (kompletní klastry C_k)

```
1:   přidán ← False
2:   if (openClusters = 0)
3:       openClusters ← VytvořNovýKlastr( $p$ )
4:   else
5:       foreach (klastr  $C$  in openClusters)
6:           if ( $p.ToA - C.maxToA > D_{max}$ )
7:               closedClusters ← OdešliKompletniKlastr( $C$ )
8:               continue
9:           if ( $abs(p.ToA - C.minToA) > \Delta t$ )
10:              continue
11:          if (PixelNeniUvnitřHranicKlastru( $C, p$ ))
12:              continue
13:          foreach (pixel  $pix$  in  $C$ )
14:              if (SplněnáPodmínkaSousedství( $pix, p$ ))
15:                  if (přidán)
16:                      SpojKlastry(posledníKlastr,  $C$ )
17:                      AktualizujMinMaxToA(posledníKlastr,  $C$ )
18:                      AktualizujMinMaxSouřadnice(posledníKlastr,  $C$ )
19:                      openClusters = openClusters -  $C$ 
20:                  else
21:                       $C \leftarrow p$ 
22:                      AktualizujMinMaxToA( $C, p$ )
23:                      AktualizujMinMaxSouřadnice( $C, p$ )
24:                      přidán ← true
25:                      posledníKlastr ←  $C$ 
26:          if (přidán = false)
27:              openClusters ← VytvořNovýKlastr( $p$ )
```

Algoritmus 2 rozšiřuje Algoritmus 1 o dvě optimalizace algoritmu. Řádek „9“ implementuje 1. optimalizaci. Pokud není *ToA* kontrolovaného pixelu p v časovém okně daného klastru C , algoritmus se přesune k dalšímu klastru. Řádek „11“ funkcí *PixelNeniUvnitřHranicKlastru*(C, p) implementuje 2. optimalizaci. Tato funkce za pomoci série podmínek zkontroluje, zda se pixel p nenachází uvnitř hranic klastru C . Posledními změnami jsou funkce *AktualizujMinMaxToA*() a *AktualizujMinMaxSouřadnice*(), které

aktualizují uložené parametry daného klastru ($minToA$, $maxToA$, $xMax$, $xMin$, $yMax$, $yMin$) pro správnou funkci algoritmu.

Závěrem lze prohlásit, že dosažený výkon pro algoritmus „baseline“ je velmi závislý na typu pixelových dat. Zatímco pro dataset E (elektrony) je výkon 7,0 MHit/s, dataset B (ionty olova) dosahují výkonu pouze 1,0 MHit/s. Je to tím, že klastry datasetu B mají v průměru největší velikost (Tabulka 3). Tím se ve vnořené smyčce na řádce „13“ projeví asymptotická složitost algoritmu $O(n^2)$.

3.5 Optimalizace algoritmu „quadtree“

Algoritmus „quadtree“ staví své základy na algoritmu „baseline“, pouze pozměňuje způsob ukládání klastrů (viz Kapitola 2.2.2). 2D plocha detektoru je rekurzivně rozdělována na kvadranty, čímž efektivně zužujeme výběr potenciálních klastrů C , kterých by mohl být pixel p součástí. Jelikož tato práce částečně navazuje na diplomovou práci [4], máme k dispozici již optimalizovaný algoritmus „quadtree“. V této kapitole bude popsána jeho implementace do testovacího prostředí a následně bude změřen a analyzován jeho výkon.

3.5.1 Implementace algoritmu „quadtree“

Klastry jsou u tohoto algoritmu ukládány obdobně, jako u algoritmu „baseline“. Datové struktury pro ukládání dat jsou následující:

- **Neighbors(vector<node>, root)** ukládá možné *sousední pixely* v kvadrantovém stromu. Ukládá kořen stromu **root** a pole všech uzlů **node**. Na rozdíl od „baseline“ se při kontrole sousedství nevypočítává, zda je pixel p *soused*, ale kontroluje se, zda **Neighbors** obsahuje *sousední pixel* na souřadnici pixelu p . Při vložení pixelu do klastru je nutné **Neighbors** aktualizovat;
- **ClusterData(vector<OnePixel>, minToA, Neighbors)** je obdobou datové struktury **OneCluster**, avšak zde chybí informace o **maxToA**, a také informace o hranicích klastru **xMax**, **xMin**, **yMax**, **yMin**. To v tomto případě nevádí, protože způsob kontroly sousedství se provádí pomocí uložených sousedů **Neighbors**;
- **ClusterDataTree(vector<ClusterData>)** ukládá všechny klastry do polí v kvadrantovém stromu. Jedná se o datovou třídu, která svými členskými metodami realizuje rekurzi a správu stromu. Jeden klastr **ClusterData** je ukládán do několika různých polí, které vždy budou odpovídat danému uzlu ve stromu. Tato struktura se používá pro ukládání klastrů *openClusters* a *closedClusters*.

Z datových struktur vidíme odlišný způsob ukládání klastrů a kontroly sousedství. Samotný algoritmus už je podobný, ve smyčce jsou proti přichozímu pixelu p zkontrolovány všechny klastry *openClusters* na časovou a sousedskou podmínku, a pokud je splněná, pixel je uložen do správného klastru.

Algoritmus 3: Klastrování „quadtree“

Vstup: pixel p ; pole *openClusters* (obsahující klastry C); max. zpoždění klastru D_{max}

Výstup: pole *closedClusters* (kompletní klastry C_k)

```
1:   přidán ← False
2:   if (openClusters = 0)
3:       openClusters ← VytvořNovýKlastr( $p$ )
4:   else
5:       foreach (klastr  $C$  in openClusters)
6:           if ( $p.ToA - C.ToA > D_{max}$ )
7:               closedClusters ← OdešliKompletniKlastr( $C$ )
8:               AktualizujStrom()
9:               continue
10:          if (ČasováPodmínka( $C$ ,  $p$ ) AND  $C.NeighborsObsahuje(p)$ )
11:              if (přidán)
12:                  SpojKlastry(posledniKlastr,  $C$ )
13:                  AktualizujMinToA( $C$ ,  $p$ )
14:                  AktualizujStrom()
15:              else
16:                   $C \leftarrow p$ 
17:                  přidán ← true
18:                  posledniKlastr ←  $C$ 
19:                  AktualizujMinToA( $C$ ,  $p$ )
20:                  PřidejNovéSousedy( $C.Neighbors$ )
21:                  AktualizujStrom()
22:          if (přidán = false)
23:              openClusters ← VytvořNovýKlastr( $p$ )
24:              AktualizujStrom()
```

Původní algoritmus, ze kterého Algoritmus 3 vychází, je detailně popsán v diplomové práci [4]. Výhodou řešení je jednoduchá kontrola sousedství $C.NeighborsObsahuje(p)$, což umožňuje ve stromové struktuře efektivně hledat sousedy. Namísto prohledávání všech pixelů všech klastrů tomu u algoritmu „baseline“ je ve stromové struktuře rekurzivně vyhledán uzel, do kterého pixel p spadá. Tím je počet klastrů na kontrolu sousedství snížen. Toho je dosaženo ukládáním sousedů do pole $PřidejNovéSousedy(C.Neighbors)$.

Nevýhodou je, že algoritmu přibyla nutnost neustálé aktualizace kvadrantového stromu pomocí *AktualizujStrom()*, protože každá změna ve stromu může znamenat nutnost jeho aktualizace (např. pokud pixely uložené v uzlu přetečou jeho hranice).

Výkon algoritmu „quadtree“ byl testován se stejným nastavením a optimalizacemi jako algoritmus „baseline“.

Tabulka 9: Finální výkon algoritmu „quadtree“ (aritmetický průměr z 5 měření)

Dataset	A	B	C	D	E
Čas (ms)	3	4 970	5 007	17 744	834
Výkon (MHit/s)	5,3	1,1	1,6	3,9	3,8

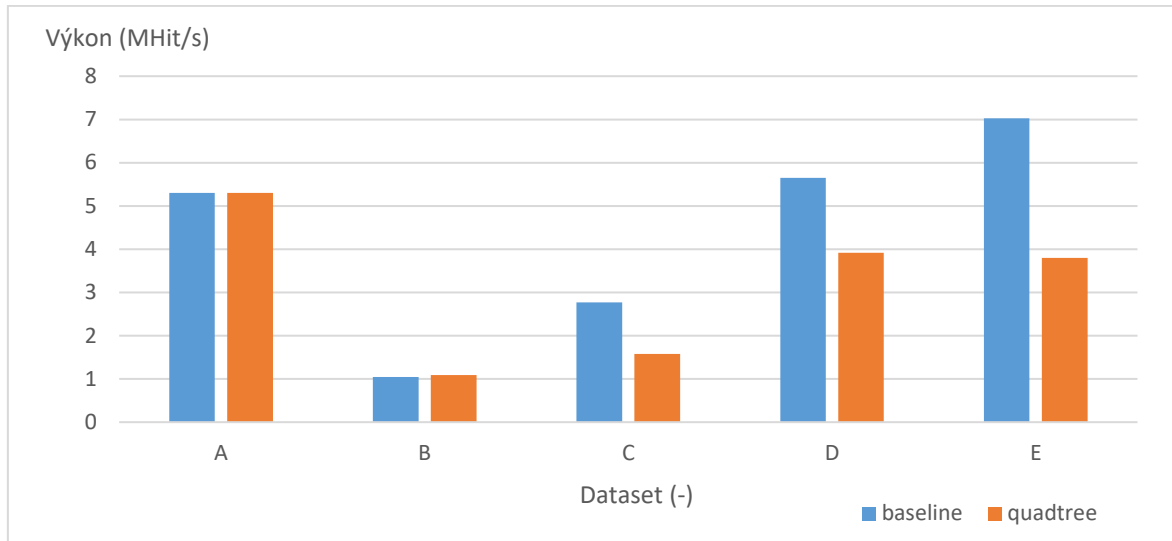
3.6 Porovnání výkonu, využití paměti a přesnosti algoritmů

V této kapitole budou porovnány výkony klastrování obou algoritmů, ale také jejich využití paměti, což je pro klastrování dat na Katherine zásadní. Dále bude porovnána přesnost obou algoritmů s výsledky klastrování programu Burdaman.

Z grafu viz Obr. 18 vidíme, že algoritmus „quadtree“ v klastrování datasetů C, D a E výrazně zaostává za výkonem algoritmu „baseline“. Jde to proti logice, protože by algoritmus „quadtree“ měl mít teoreticky efektivnější hledání sousedů ve 2D prostoru vyplývající z jeho podstaty. Mělo by tomu tak být zejména v případech, kdy klastry *openClusters* obsahují velké množství pixelů.

Pro vysoký hitrate by výkon „baseline“ oproti „quadtree“ měl klesat, jak ukazuje komparativní analýza metod pro detekci kolizí [15]. Z analýzy vyplývá, že čas potřebný pro vyhledání sousedů měl u algoritmu „baseline“ kvadratickou závislost v závislosti na počtu objektů, zatímco „quadtree“ měl závislost přibližně lineární [15]. To v našem případě díky četným optimalizacím algoritmu „baseline“ neplatí. Optimalizovaný algoritmus „baseline“ ve své podstatě má stále stejnou asymptotickou složitost, ale díky 2. optimalizaci se marginálně snižuje počet kontrolovaných pixelů na sousedské podmínky. V kombinaci s 1. optimalizací a odesíláním kompletních klastrů je počet pixelů na kontrolu výrazně snížen.

Výhody „quadtree“ se projevily u velkých klastrů z datasetu B. Tento dataset s ionty olova obsahuje největší klastr o velikosti 5695 pixelů. Zde je „quadtree“ výkonnější, neboť u algoritmu „baseline“ se projevuje asymptotická složitost. U ostatních datasetů jsou klastry menší a zrychlení pomocí prohledávání sousedů v kvadrantech nekompensuje čas nutný na četné aktualizace stromové struktury.



Obr. 18: Porovnání výkonů klastrování algoritmů baseline a quadtree v závislost na datasetu

Algoritmus „quadtree“ může být rychlejší, ale jen v případě extrémně velkých částic. Pro jednoznačné určení předností algoritmů pro jednotlivá měření by bylo nutné více druhů pixelových dat a hlubší analýza. Z dostupných dat, která pokrývají velkou část běžných měření lze usoudit, že algoritmus „baseline“ dosahuje vyšších výkonů.

Paměťové využití obou algoritmů je podobné. Pro dataset E bylo využití paměti „quadtree“ 194 MB, zatímco pro „baseline“ to bylo 193 MB. Největší rozdíl byl u velkého datasetu D, kde bylo využití 3800 MB, respektive 4000 MB RAM. Rozdíl paměťového využití byl tedy 5 %. Když vezmeme v potaz vyšší rychlost klastrování algoritmu „baseline“, jde o přijatelný rozdíl. Algoritmus „baseline“ by dodatečně ušetřil paměť použitím datového typu *float* namísto *double* u *ToA* a pomocných proměnných, popřípadě změny *ToT* z *uint32* na *uint16*.

Nakonec porovnáme přesnosti obou algoritmů. Ty byly u obou algoritmů naprosto totožné. Přesnost podle vzorce (3.1) pro datasety v pořadí od A do E jsou: 99,1 %; 97,2 %; 95,6 %; 26,4 %; 98,7 %. Nízká přesnost u datasetu D 26,4 % neznamena nutně špatný výsledek. Po vizuální kontrole náhodných klastrů jsou klastry kompletní. Nižší přesnost může být způsobena rozdílným nastavením maximálního zpoždění pixelů D_{max} a časového okna Δt v programu Burdaman. Proto může u některých klastrů chybět pixel z důvodu latence.

Na základě výše uvedených výsledků bude v zařízení Katherine implementován algoritmus „baseline“. Algoritmus v určitých případech využívá více paměti, ale je pro většinu datasetů výkonnější. Jeho potenciální budoucí implementace v FPGA bude nesrovnatelně jednodušší, neboť implementace rekurze a aktualizování stromové struktury „quadtree“ by v FPGA nebyla triviální.

3.7 Výkon kalibrace energií

Kalibrace energií hitů je nutná pro přesný výpočet energie klastru z údaje ToT . Kalibrace je také nutná pro kompenzaci chyb a nehomogenity pixelové matice detektoru. Výpočet energie podle (1.2) vyžaduje načtení kalibračních matic A , B , C , a t ze souborů do paměti zařízení. Nevýhodou je nutnost provádění výpočtu pro každý hit, což zpomaluje algoritmus. Pro dataset B algoritmus klastrování zpomalil vlivem počítání kalibrace o **18 %** (aritmetický průměr pěti měření).

Zvýšení výkonu kalibrace lze dosáhnout pomocí předem vytvořené LUT (vyhledávací tabulky). LUT má tu výhodu, že pouze přečteme jednu z předem vypočtených hodnot. Zpomalení algoritmu s využitím LUT bylo pro datasety v průměru **7 %**, což je oproti zpomalení **18 %** klasickým výpočtem velké zlepšení.

Kalibrační LUT musí mít tři rozměry: osu x ; osu y ; ToT . Pomocí těchto tří rozměrů je vytvořeno trojrozměrné pole o velikosti $256 \times 256 \times H$. H v tomto případě označuje hloubku (maximální ToT), pro které lze skutečnou energii v tabulce vyhledat. Bohužel, hloubkou se lineárně zvyšuje paměťové využití, což může být v zařízení Katherine problém. Největší ToT v rámci datasetů je hodnota 1022 v datasetu B. Tomu odpovídá nutnost ukládat LUT o velikosti 267 MB (pole $256 \times 256 \times 1022$) v paměti RAM, což už je u verze Katherine s 1 GB RAM nepřijatelné.

Velikost maximálního ToT při měření nelze předem určit. Vytvoření LUT o hloubce 2000 by už znamenalo využití RAM téměř 550 MB. Z tohoto důvodu bude na zařízení Katherine implementován pomalejší výpočet kalibrace podle (1.2).

LUT lze jednoduše optimalizovat na poloviční využití paměti použitím 2 B typu `uint16_t` namísto 4 B typu `uint32_t`. Další optimalizací by mohlo být dynamické zvětšování LUT, konkrétně v rozměru H , podle potřeby.

3.8 Paralelizace klastrování

Do této chvíle byla tato kapitola věnována hlavně testům a optimalizacím základních algoritmů. Následující kapitoly budou věnovány možnostem paralelizace vybraného algoritmu „baseline“.

Paralelizace zvyšuje výkon klastrování, ale má své nevýhody. Na procesoru lze paralelizovat algoritmus do více vláken za předpokladu, že máme k dispozici dostatek jader/vláken. Výkon ale nemůže stoupat počtem vláken lineárně, protože více běžících vláken vyžaduje synchronizační a řídicí úkony.

3.8.1 Možnosti paralelizace

Paralelizace klastrování je možná za předpokladu, že tok hitů od detektoru jsme schopni vhodně rozdělit na více částí, aniž bychom porušili časovou souslednost hitů. Doposud bylo představeno pouze klastrování v jednom vlákne (Obr. 13). Nabízí se následující možnosti paralelizace:

- **Časová paralelizace** – tok dat je časově rozdělen na menší časové úseky, na kterých je zvlášť prováděno klastrování v paralelních vláknech. Po dokončení klastrování jednotlivých úseků je nutné provést jejich opětovné spojení, neboť časově hraniční klastry mohou postrádat zbytek klastru v následujícím vlákne;
- **Prostorová paralelizace** – tok dat je rozdělován prostorově na kvadranty. Pro paralelizaci do 4 vláken se prostor detektoru rozdělí na kvadranty. Přibývá nutnost pátého vlákna, které provádí spojování hraničních klastrů z jednotlivých kvadrantů. Také se komplikuje uzavírání klastrů, neboť část klastrů v jednom kvadrantu už splnila podmínku pro kompletizaci klastru, zatímco druhá ještě nespojená část klastru v druhém kvadrantu nikoliv;
- **Paralelní pipeline (sériová paralelizace)** – tok dat zůstává kontinuální, jednotlivé bloky z pipeline klastrování (Obr. 13) jsou realizovány jako paralelní vlákna. Žádné spojování okrajových klastrů není nutné, nevýhodou je nižší očekávaný výkon.

Z představených možností mají největší potenciál na výkon (škálovatelnost) Časová a Prostorová paralelizace. Časová paralelizace má méně nevýhod. Časově lze tok dat rozdělit na libovolný počet vláken, zatímco prostorová paralelizace vyžaduje vhodný počet vláken pro rozdělení na stejné velké kvadranty. U časové paralelizace také odpadá nevýhoda zmíněného uzavírání klastrů. Nevýhodou může být také nerovnoměrné radiační pole, které dopadá jen na jeden kvadrant detektoru a zatěžuje jedno vlákno více než ostatní [16]. Tento problém u časové paralelizace může nastat snad jen u pulzujícího pole (proměnný hitrate v čase), což se dá kompenzovat tím, že každé vlákno dostane stejný počet hitů.

K rozhodnutí, jaká paralelizace je výhodnější, poslouží výsledky z diplomové práce [16], ve které autor provedl teoretické výpočty pravděpodobnosti nutnosti spojování hraničních klastrů přímo pro detektor Timepix3.

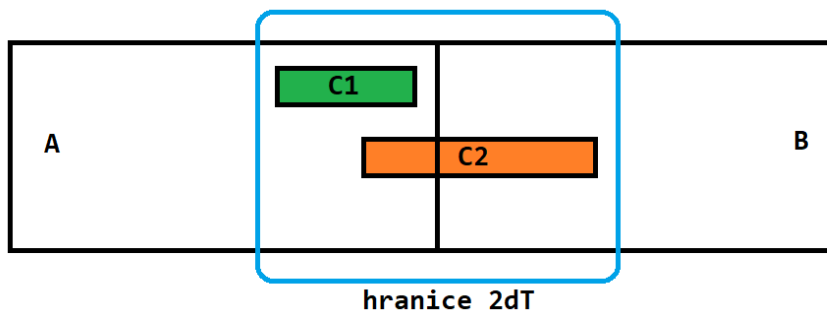
Z práce vyplývá, že pravděpodobnost potřeby spojování hraničních klastrů může být zásadně menší u Časové paralelizace oproti Prostorové. Pro velmi krátké časové okno 10 us (která se přidělí jednomu vláknu) byla pravděpodobnost spojování vypočtena pouze

na 0,25 %, zatímco pro prostorovou paralelizaci byla pravděpodobnost 2 % (s rozdělením prostoru na čtyři kvadranty). Nejzásadnější ale je, že časové okno 10 us je velice krátké a reálná délka okna pro paralelizaci bude řádově delší, čímž se dále sníží pravděpodobnost nutnosti spojovat rozdělené klastry. [16]

Na základě závěrů z práce [16] budou představená řešení paralelizace stavět na základech Časové paralelizace.

3.8.2 Princip časové paralelizace

Jak již bylo zmíněno, časová paralelizace rozděluje tok dat na krátké časové úseky (okna), které zpracují paralelní vlákna provádějící klastrování. Pokud bychom např. chtěli provádět paralelní klastrování na pixelových datech trvajících 400 ms na 4 vláknech procesoru, každé vlákno by zpracovávalo 100 ms okno pixelových dat. Po dokončení klastrování v jednotlivých vláknech ale narážíme na problém spojování klastrů. Protože byla data rozdělena na čtyři stejně dlouhé časové rámce, může se stát, že některý klastr je rozdělený napůl mezi dvě vlákna. To například znamená, že pokud obsahuje klastr 10 pixelů, tak 4 pixely může zpracovávat vlákno 1, zatímco zbylých 6 pixelů zpracovává vlákno 2. Takový případ je na Obr. 19.



Obr. 19: Hranice dvou časových oken A a B. Klastr C1 je uvnitř hranice, ale spojení nebude potřeba. Klastr C2 byl rozdělen mezi dvě vlákna, a bude muset být následně spojen

Zatímco klastr C1 spojení dat z vláken nepotřebuje, klastr C2 má své pixely rozděleny do dvou časových oken (do dvou vláken). Oba klastry jsou ovšem hraniční, protože některé jejich pixely zasahují do hranice časových oken. Velikost hraniční oblasti je rovna délce časového okna klastru Δt na každou stranu od hranice.

Pokud je libovolný klastr hraniční, je třeba provádět spojování klastrů (merging). Znamená to, že jednotlivá vlákna musí počkat na výsledek všech hraničních vláken, a s těmi si vyměnit hraniční klastry, které musí být spojeny:

- Vlákna, která dokončí klastrování svého časového okna dříve než ostatní vlákna, musí na tato vlákna čekat;
- Provést spojování klastrů po předání hraničních klastrů mezi vlákny. Spojování klastrů spočívá v tom, že všechny pixely obou hraničních klastrů jsou proti sobě zkontrolovány na sousedské podmínky. I zde jsou aplikovatelné všechny optimalizace uvedené v kapitole 3.4.2.

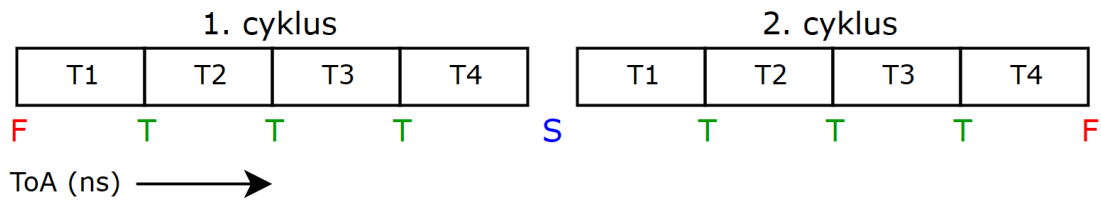
Algoritmus 4: Algoritmus jednoho vlákna klastrování se spojováním hraničních klastrů

Vstup: časové okno **Frame** obsahující pixely **p**; pole *openClusters* (obsahující klastry **C**)

Výstup: pole *closedClusters* (kompletní klastry **C_k**)

```
1:  vláknoDokončeno ← False
2:  foreach (pixel p in Frame)
3:      closedClusters ← Baseline(p, openClusters)
4:      OdešliKompletniKlastry(closedClusters)
5:  vláknoDokončeno ← True
6:  while (ostatníVláknaDokončena = false)
7:      sleep()
8:  hraničníC = HraničníKlastrySousedníchVláken()
9:  closedClusters ← SpojKlastry(openClusters, hraničníC)
```

Algoritmus 4 popisuje funkci vlákna klastrování se spojováním hraničních klastrů. Vláknu je přiděleno časové okno **Frame** obsahující pixely **p**. Poté je prováděn algoritmus „baseline“ než jsou všechny pixely **p** přiřazené k *openClusters* nebo *closedClusters*. Zatímco část klastrů je odeslána *OdešliKompletniKlastry()*, část klastrů *openClusters* je ponechána jako hraniční klastry (protože se nacházely v hraniční oblasti viz Obr. 19). Poté vlákno spí, dokud ostatní vlákna nedokončí své klastrování. Následně jsou funkcí *HraničníKlastrySousedníchVláken()* získány hraniční klastry sousedních vláken, které jsou s hraničními klastry aktuálního vlákna spojeny funkcí *SpojKlastry()*. Výsledkem je pole kompletních klastrů *closedClusters*. Po spojování mohou být vláknům přiřazena nová časová okna.



Obr. 20: Navazující výpočetní cykly s vyznačením nutnosti spojování. T_n jsou vlákna se svými indexy

Pro úplnost je na Obr. 20 (rozvinutí Obr. 19) uveden příklad časové osy klastrování při rozdělení dat do 4 vláken. Klastrování je rozděleno do dvou výpočetních cyklů. Je třeba rozlišit hraniční situace:

- Situace „F“ – Začátek a konec měření. Hraniční klastry jsou kompletní a neexistují pixely, které by hraničním klastrům chyběly;
- Situace „T“ – Je nutné spojit hraniční klastry. Klastry jsou spojeny mezi dvěma sousedními vlákny (např. T1 a T2);
- Situace „S“ – Speciální případ spojování klastrů. Na časové ose se nacházíme mezi prvním a druhým výpočetním cyklem. Samotné spojování klastrů probíhá stejně jako u situace „T“, ale s rozdílem, že zde spojujeme hraniční klastry nesousedních vláken (T4 a T1).

Nastává otázka délky jednotlivých časových oken. Ta závisí na více faktorech:

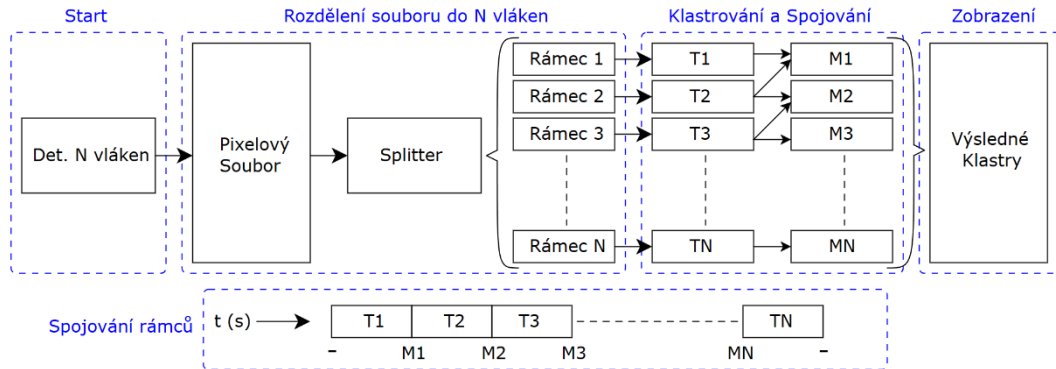
- Počtu vláken;
- Délce časového okna klastru Δt ;
- Maximální zpoždění pixelů D_{max} ;
- Maximální akceptovatelná latence výsledků klastrování (pro uživatele).

Minimální délka časového okna musí být větší nebo rovna D_{max} , protože při klastrování tok hitů přichází časově zpřeházený a jeden hit se může zpozdít o maximální latenci D_{max} .

S krátkou délkou časového okna se musí provádět přerozdělování pixelů do vláken a opětovné spojování hraničních klastrů, což jsou časové náročné operace. Je tedy snaha mít co nejdelší časové okno. To představuje kompromis, znamená to větší nároky na paměť RAM. Zároveň se zvyšuje latence pro uživatele. Na druhou stranu dlouhé časové okno snižuje pravděpodobnost spojování klastrů [16]. Volba délky časového okna je tedy kompromisem mezi výpočetní rychlostí (výkonem klastrování) a pamětí/latencí.

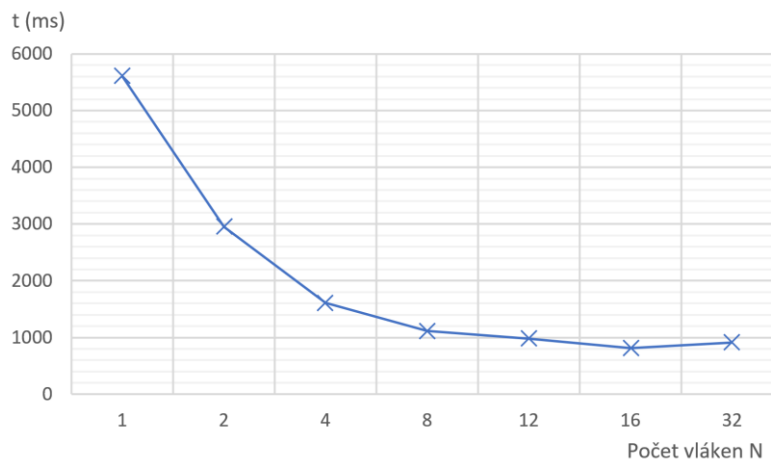
3.9 Možnosti paralelizace algoritmu v procesoru

Algoritmus 4 byl pro test výkonu paralelizace klastrování implementován v testovacím prostředí. Na Obr. 21 je vidět architektura implementovaného algoritmu.



Obr. 21: Architektura časové paralelizace klastrování na PC. Výpočetní cyklus (nahore); spojování (dole)

Při startu programu je detekován počet vláken (nebo jader) procesoru. Poté je pixelový soubor rozdělen do N stejně dlouhých časových rámců (velké soubory je nutné zpracovat po částech). Následně jsou časové rámce předány N vláknům (T_n) provádějícím klastrování „baseline“. Po dokončení klastrování ve všech vláknech jsou hraniční klastry spojeny (M_n). Spojování hraničních klastrů je naznačeno na spodní části Obr. 21.



Obr. 22: Čas klastrování v závislosti na počtu vláken pro dataset B

Z výsledků na Obr. 22 vidíme 90 % nárůst výkonu ve dvou vláknech. Další 82 % nárůstu při čtyřech vláknech. Při přechodu z 8 na 16 vláken už je nárůst výkonu nižší, a to 37,5 %. Navýšení na 32 vláken na 16 vláknovém stroji výkon logicky o 10 % snižuje. To znamená, že s přibývajícím počtem vláken vzrůstají nároky na režii vláken a spojování klastrů, čímž klesá efektivita paralelizace. Výsledný výkon klastrování při 16 vláknech je

6,7 MHit/s (nárůst výkonu 609 %). Po kontrole přesnosti vyplývá, že algoritmus je při dodržení podmínky minimální velikosti časového okna D_{max} deterministický.

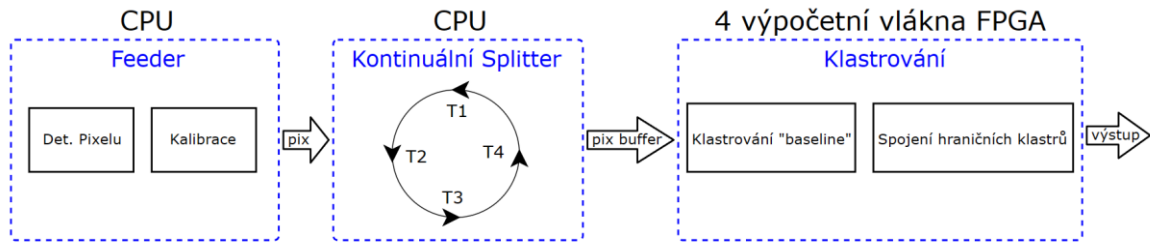
Prezentované výsledky byly testem škálovatelnosti výkonu s počtem jader. Test byl proveden offline, tedy ne za běhu reálného měření. Algoritmus ale může být vhodný pro budoucí verze Katherine s větším počtem jader. Verze Katherine Gen2 má dvě jádra a nezanedbatelné režijní náklady na běh Linuxu a příjem hitů z FPGA, paralelizace na tomto zařízení tedy nedává smysl. V budoucnu by se případný vyšší počet vláken procesoru mohl na časovou paralelizaci klastrování využít. Možností by byla také implementace více jádrového softwarového procesoru do FPGA, který by paralelní klastrování prováděl.

3.10 Možnosti paralelizace algoritmu v FPGA

V minulé kapitole padla zmínka o implementaci paralelního klastrování softwarovým procesorem v FPGA. V této kapitole bude prozkoumána možnost implementace paralelního klastrování v FPGA do logických bloků přímo navržených pro klastrování.

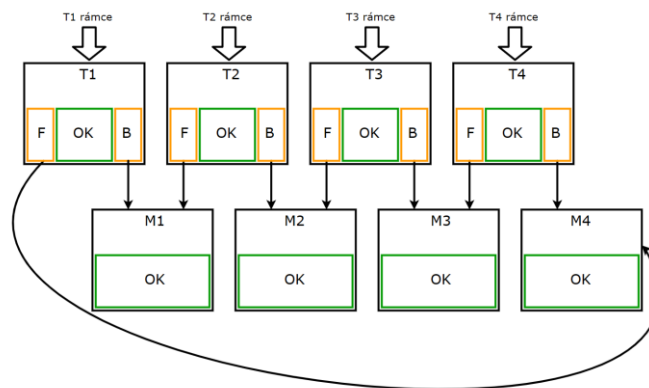
Nejprve pojďme prozkoumat nutné prostředky pro implementaci v FPGA. Algoritmus „baseline“ obsahuje množství podmínek, buffery pro ukládání pixelů, klastrů *openClusters* a *closedClusters*. Podmínky nejsou pro realizaci v FPGA překážkou, ale ukládání klastrů do bufferů ano. Klastrů může být při vysokém hitrate velké množství a nároky na paměť v řádu stovek MB. Jelikož struktura **OneCluster** obsahuje mnoho čísel a pole pixelů, není ukládání a čtení takového klastru z bufferu triviální. Zároveň velikost klastru není statická, ale dynamická, protože se počet pixelů v klastrů během klastrování navyšuje. Dynamická změna velikosti klastru poté vyžaduje realokaci paměti v bufferu, což může být časově náročná a netriviální záležitost. FPGA by si muselo ukládat adresy jednotlivých klastrů a pixelů do speciálního bufferu. V opačném případě by musely být buffery prohledávány a musely by být vloženy extra informace pro popis dat, což by dále navýšilo nároky na paměť a výpočetní prostředky. Implementací takového algoritmu paralelně by nároky na návrh hardwaru ještě vzrostly.

Nicméně i přes všechny překážky by měla být implementace při dostatečné velikosti paměti a počtu LE v FPGA možná. Paměť by se dala ušetřit přesunutím co nejvíce možné zátěže na procesor a FPGA by provádělo jen samotný algoritmus klastrování a bufferování nezbytných dat. Jako návrh potenciálního řešení byla navržena následující architektura (Obr. 23), jejíž model byl implementován v testovacím prostředí PC.



Obr. 23: Architektura paralelního klastrování v SoC

Nejprve jsou pixely přijaty blokem „Feeder“, ve kterém procesor přijme pixel a provede kalibraci ToT na energii. Blok „Kontinuální Splitter“ rozděljuje přijatý datový tok pixelů na N časových rámců do výpočetních vláken FPGA. Následně vlákna provádějí algoritmus „baseline“ a provádějí spojování hraničních klastrů podle Obr. 20. Výstupní klastry jsou FPGA odeslány přes Ethernet nebo uloženy do bufferu.



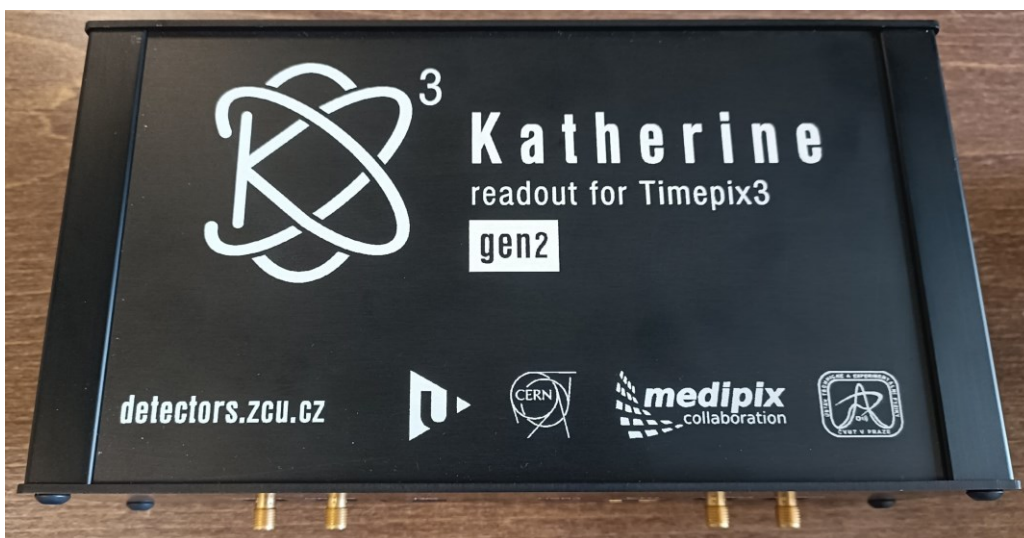
Obr. 24: Model vláken klastrování v FPGA. Kompletní klastry (OK); přední hraniční klastry (F); zadní hraniční klastry (B)

Obr. 24 poskytuje bližší pohled do navržené architektury v FPGA. Obrázek ukazuje případ pro 4 výpočetní vlákna. Bloků je 8, protože kromě 4 vláken pro klastrování T_n jsou nutné výpočetní vlákna pro spojování hraničních klastrů ve stejném počtu M_n . Funkce algoritmu je následovná:

- Přidělení časových rámců do výpočetních vláken pro klastrování „T“;
- Provedení klastrování v jednotlivých vláknech;
- Kompletní klastry „OK“ jsou odeslány do PC, hraniční klastry (F a B) jsou předány výpočetním vláknům pro spojování klastrů „M“;
- Čekání na synchronizaci a dokončení klastrování ve všech vláknech „T“;
- Výpočetní jednotky „M“ provedou spojení klastrů, klastry „OK“ odeslány;
- Vlákna „T“ přijímají další časové rámce pixelových dat a cyklus se opakuje.

4 Implementace v hardware a dosažený výkon klastrování

V této kapitole bude popsána implementace programu užívajícího optimalizovaný algoritmus „baseline“ (Algoritmus 2) na procesoru v zařízení Katherine Gen2 (Obr. 25). Půjde o jakýsi Plugin, protože program bude doplňovat funkce existujícímu řídicímu programu v Katherine ovládaného přes program Burdaman. Popsán bude také vytvořený obslužný PC program, pomocí kterého bude možno klastrování v Katherine ovládat a měnit jeho parametry. Plugin a obslužný program budou tvořit dohromady systém, který umožní akceleraci klastrování dat přímo v Katherine s možností post-processingu a ukládání naměřených dat v PC.



Obr. 25: Vyčítací zařízení Katherine Gen2

4.1 Požadavky na funkce

Hlavním požadavkem celého systému je akcelerace klastrování v Pluginu a možnost zobrazovat výsledky v Obslužném programu.

Tabulka 10: Funkce Pluginu implementované v zařízení Katherine

Název funkce	Popis	Výpočetní náročnost
Klastrování	Klastrování pixelových dat přímo v zařízení	Vysoká
Akvizice histogramů	Ušetření propustnosti Ethernetu v případě, že uživatelé zajímají pouze energie částic	Vysoká+
Pixel Counting	Příjem souřadnic pixelů bez ToT a ToA . Vhodné pro pořizování rentgenových snímků	Nízká
Příjem pixelů	Příjem pixelů se zpracováním klastrování v PC	Nízká+
Idle	Nečinný stav Pluginu	Zanedbatelná

Společně s klastrováním budou v Pluginu implementovány i další metody pro analýzu IZ (Tabulka 10). Zejména Akvizice histogramů je přínosná a jedná se o klastrování, kde je výstupem pouze energie nebo *ToT* klastru. Obslužný program bude mít následující požadavky.

Tabulka 11: Funkce Obslužného programu

Název funkce	Popis
Ovládání Pluginu	Výběr aktivních funkcí Pluginu
Prohlížení dat	Prohlížení jednotlivých klastrů a zobrazování jejich parametrů (energie, čas akvizice, časové rozpětí, histogramu energií)
Ukládání dat	Uložení a načtení naměřených dat pro možnost další analýzy
Zobrazení statistik	Statistika o běhu Pluginu (okamžitý hitrate, klastry za sekundu)
Nastavení parametrů	Parametry klastrování (filtrování velikosti klastrů, Δt , D_{max})

4.2 Použité nástroje a jazyky

Obslužný program bude založený na testovacím prostředí. Plugin bude mít podobné požadavky jako Obslužný program, nicméně půjde o program bez vlastního GUI. Plugin bude stejně jako obslužný program napsán v jazyce C++ a bude použitý stejný dialekt C++14, který je podporovaný kompilátorem ARM [11]. Podpora tohoto dialektu je kompilátorem „arm-linux-gnueabi“ velice rozsáhlá. Nevýhodou je podpora vícevláknových tříd jako *std::thread* ve verzi alfa. Více vláken bude potřeba pro plynulé zpracování pixelových dat viz další kapitoly.

Zmíněný kompilátor je součástí toolchainu GCC 4.x, který má zabudovanou podporu pro ARM Development Studio 5 (DS-5). V tomto IDE bude kód Pluginu napsán a po sestavení bude vygenerován jako „executable“, bude tedy samostatně spustitelným programem.

Aby byl kompilátor schopen přeložit zmíněné alfa funkcionality pro více vláken, je nutné při kompilaci zadat extra argumenty. Pro C++ kompilátor (přesněji jeho preprocesor) a C++ linker je nutno zadat argument „-pthread“, aby došlo k úspěšnému sestavení programu. Pro co největší portabilitu je klíčové, aby byl argument zadán jak u Linkeru, tak u Kompilátoru [17].

Pokud Linux nemá nainstalované potřebné C++ dynamické knihovny DLL, zahlásí po spuštění program chybu „GLIBCXX_3.4.19 not found“. To byl případ u Katherine Gen1, která neměla potřebné C++ knihovny nainstalované. Řešením problému bylo nalinkovat knihovny staticky už při sestavení. Proto bylo nutné pro C++ linker zadat následující argument: „-static-libstdc++“. Soubor .exe se zvětší o velikost knihovny (663 kB vs

53 kB), ale poté je jisté, že program půjde spustit na kterémkoliv zařízení Katherine (Gen1 a Gen2) a je zajištěna největší možná portabilita [17].

Pro nejvyšší možný výkon Pluginu je samozřejmostí optimalizační flag „-O3“. Kompletní příkaz pro Kompilátor je následující:

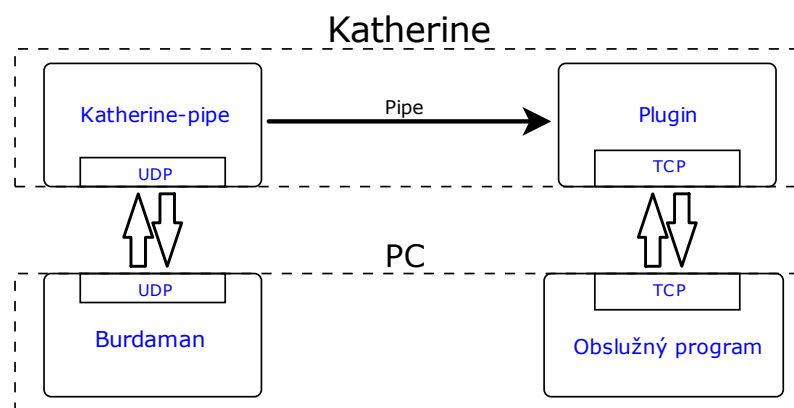
```
arm-linux-gnueabi-g++ -std=c++1y -D_cplusplus=201103L -I"PATH"  
-I"PATH" -O3 -Wall -c -fmessage-length=0 -pthread -fPIC
```

Kompletní příkaz pro Linker:

```
arm-linux-gnueabi-g++ -static-libstdc++ -flto -pthread
```

4.3 Architektura systému

Systém je z hlediska připojení založen na architektuře Klient-Server. Obslužný program (Server) je spuštěn první a Plugin (Klient) se k němu připojí. Použitým transportním protokolem je TCP, který zaručuje spolehlivé doručení zpráv adresátovi, což je pro spolehlivé ovládání Pluginu a doručení všech výstupních dat klastrování klíčové. Model počítačové komunikace je v tomto případě Master/Slave. Obslužný program (Master) má jednosměrné řízení nad Pluginem (Slave).

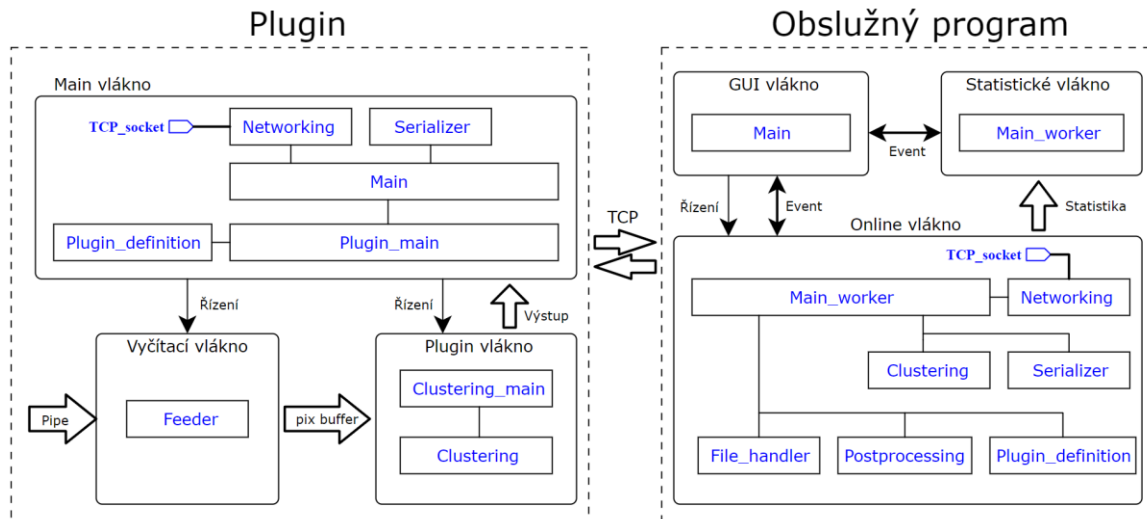


Obr. 26: Závislost Pluginu a Obslužného programu na stávajícím měřicím řetězci

Jak již bylo zmíněno, Plugin sám o sobě nespouští měření, ale chová se jako doplněk k řídicímu programu Katherine-pipe, který je ovládán PC programem Burdaman (Obr. 26). Pro funkci pluginu je tedy nutno mít zapnutý program Burdaman, kterým je možno spouštět měření a ovládat parametry měření. Následně je spuštěn Obslužný program a Plugin. Program Plugin s Katherine-pipe vytvoří spojení typu „Named Pipe“ (jednosměrná komunikace mezi procesy typu FIFO). Přes Pipe poté Plugin přijímá všechny zprávy, které Katherine-pipe zároveň odesílá programu Burdaman.

Pro zahájení měření stačí, aby byly všechny programy spuštěné a následně lze programem Burdaman zahájit nové měření. Plugin přes Pipe přijme data a vybraným způsobem je zpracuje. Zpracovaná data Plugin odesílá přes TCP a dále popsany komunikační protokol (Kapitola 4.4) do Obslužného programu, který data následně zobrazí. Uživatel může data dále zobrazovat, analyzovat a ukládat.

Architektura Pluginu a Obslužného programu je detailně popsána v Obr. 27, kde lze prohlédnout jednotlivá výpočetní vlákna se svými třídami.



Obr. 27: Architektura systému. Plugin běžící v Katherine komunikuje s Obslužným programem přes TCP spojení

4.3.1 Architektura Pluginu

V této podkapitole budou popsány jednotlivé bloky Pluginu z Obr. 27. Bloky budou popsány jak z hlediska vláken, tak z hlediska jednotlivých tříd.

Dle vláken:

- **Main vlákno** je hlavní vlákno, jehož hlavní smyčka obstarává vstupní příkazy z komunikace, podle příkazů řídí ostatní vlákna a odesílá výstupy Plugin vlákna Obslužnému programu. Hlavní smyčka je periodicky uspávána, aby vlákno zbytečně nezatěžovalo dvoujádrový procesor ARM. Výsledky se tedy odesílají periodicky po částech;
- **Vyčítací vlákno** obstarává připojení na program Katherine-pipe a přijímá všechny zprávy (např. začátek měření, konec měření, hity atd). Následuje zpracování zpráv, hity jsou přepočteny na pixely a umístěny do Pixelového bufferu, ze kterého je přijímá Plugin vlákno;

- **Plugin vlákno** provádí samotné funkce Pluginu. Data z bufferu jsou vybraným způsobem zpracována (např. klastrována) a umístěna na výstup, ze kterého Main vlákno odesílá data do Obslužného programu.

Dle tříd:

- **Networking** spravuje TCP připojení, neblokující čtení/odesílání zpráv a připojení na Pipe. TCP připojení je spravováno přes Socket API [18]. Po vytvoření file deskriptoru přes *socket()* se Plugin připojí k Obslužnému programu přes *connect()*;
- **Serializer** převádí data do vhodného formátu pro přenos TCP protokolem;
- **Main** inicializuje Networking a obsahuje hlavní smyčku viz „Main vlákno“;
- **Plugin_definition** definuje všechny funkcionality programu, obsahuje definici příkazů a statusu. Obsahuje také funkce pro převod hitu na pixel;
- **Plugin_main** obsahuje funkce sloužící jako rozhraní, přes které Main vlákno ovládá Vyčítací a Plugin vlákno. Např. obsahuje funkci *plugin_start(plugin)*, která spustí/zastaví vlákna pro vybranou funkci;
- **Feeder** ve své smyčce provádí operace viz Vyčítací vlákno. Přijaté zprávy mají 6 B. Zároveň hlídá množství přijatých informací a vyhazuje zprávy o přijatých hitech v případě zahlcení Plugin vlákna daty. Výzvou je také neustálý odběr zpráv z Pipe, která má omezenou kapacitu. Zaplněním Pipe by se ztratily důležité zprávy (např. o ukončení měření). Jelikož není kapacita Pipe dělitelná 6, její zaplnění 6 B zprávami způsobí chybu rámce. Řešením je zvětšení kapacity Pipe, neblokující zápis pixelů do pixelového bufferu a hlídání stavu Pipe programem Katherine-pipe;
- **Clustering_main** obsahuje hlavní smyčku Plugin vlákna. Přijímá pixely z pixelového bufferu, vykonává vybranou funkci a závěrem umísťuje data do výstupního bufferu. V případě klastrování nebo akvizice histogramů volá algoritmus klastrování ze třídy „Clustering“;
- **Clustering** provádí samotnou akceleraci klastrování pixelových dat. Obsahuje optimalizovaný algoritmus „baseline“ (viz Algoritmus 2), který je navíc obohacen o akvizici histogramů (z kompletního klastru je odeslána pouze jeho energie).

Pro bezpečnost komunikace mezi vlákny (thread-safety) Pluginu je užito standartních knihoven *std::mutex* a *std::atomic*. Zatímco proměnou typu *bool* stačí označit flagem „atomic“, pro pixelový buffer a výstupní buffer je nutné použití uzamykání prostředků

pomocí `std::mutex::lock`. Důmyslným použitím „locků“ je garantováno, že nemůže dojít k datovému konfliktu mezi vlákny (data race).

Pixelový buffer (pix buffer): Užití „locků“ může být časově náročné, pokud dvě vlákna frekventovaně přistupují ke společnému bufferu jako v případě Pixelového bufferu. Pro zvýšení výkonu byl navržen Pixelový buffer se dvěma vnitřními buffery: vstupní a výstupní buffer. Třída má přes C++ template (šablona) definovanou velikost vstupního bufferu N . Po naplnění vstupního bufferu je jeho obsah přesunut do výstupního bufferu, ze kterého čte Plugin vlákno. Čas je ušetřen, jelikož Vyčítací a Plugin vlákno musí čekat na synchronizaci „lock“ jen jednou za N zápisů.

4.3.2 Architektura Obslužného programu

V této podkapitole budou popsány jednotlivé bloky Obslužného programu. Bloky budou popsány jak z hlediska vláken, tak z hlediska jednotlivých tříd.

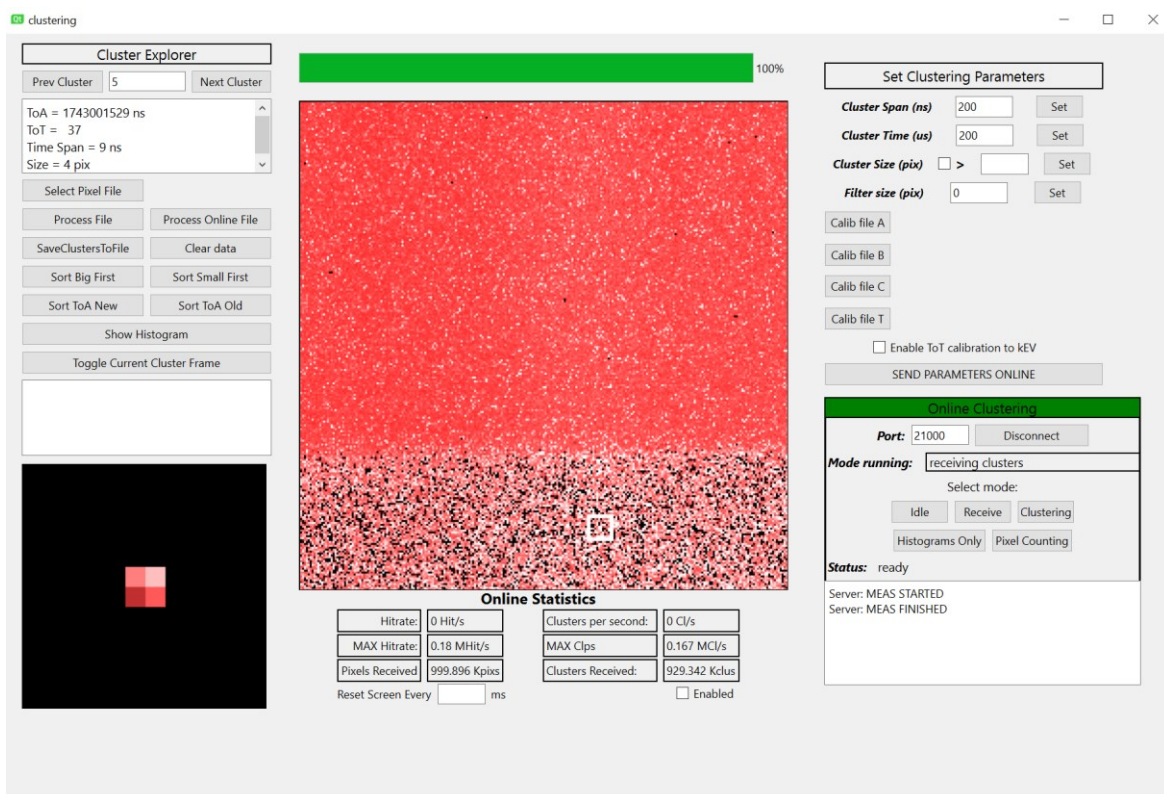
Dle vláken:

- **GUI vlákno** je hlavní vlákno programu a zároveň grafické vlákno, a také spouští další vlákna. Pro responzivní ovládání programu je důležité, aby vlákno nebylo vytěžováno, proto vykonává jen minimum operací. Framework Qt nabízí efektivní aktualizování grafických elementů skrze tzv. Eventy, takže je GUI responzivní i při plném vytížení;
- **Online vlákno** vykonává všechny funkce Obslužného programu (viz Tabulka 11) až na zobrazení statistik. Spravuje TCP připojení k Pluginu a obstarává veškerou komunikaci a řízení. Ukládá a načítá pixelové soubory, nastavuje parametry klastrování, nebo provádí klastrování offline z vybraného souboru;
- **Statistické vlákno** periodicky každou 1 s aktualizuje online statistiku o běžícím Pluginu (hitrate, klastry/s atd.).

Dle tříd:

- **Main** obsahuje Event handlersy všech interaktivních grafických prvků a spouští další vlákna;
- **Main_worker** obsahuje hlavní smyčku Online vlákna. Nejprve inicializuje TCP připojení a poté vykonává hlavní smyčku. Hlavní smyčka přijímá data od Pluginu, která následně zobrazuje. Také odesílá příkazy a parametry klastrování. Periodicky každých 200 až 1000 ms vyvolává Eventy pro vykreslení dat;

- **Networking** obsahuje funkce pro vytvoření serverového TCP připojení. Podobně jako Plugin používá Socket API [18] s tím rozdílem, že platformou může být také OS Windows. Z toho důvodu musí být použity jiné knihovny a načítání dynamických knihoven DLL specifických pro Windows viz [18];
- **Serializer** je totožná třída se stejnojmennou třídou v Pluginu;
- **File_handler** zajišťuje ukládání a načítání pixelových/klastrových dat do/z PC;
- **Postprocessing** umožňuje připravení snímků pro vykreslení v GUI, ale také přepočty *ToT* a počtu pixelů (Pixel Counting) na barvu pixelu. Pro vykreslování klastrů a pixelů je použita knihovna Qt *QPixmap*, zatímco pro zobrazení histogramů energií je použita knihovna *QtCharts*;
- **Plugin_definition** je totožná třída se stejnojmennou třídou v Pluginu;
- **Clustering** obsahuje algoritmy „baseline“ a „quadtree“ pro provádění klastrování offline na PC v případě, že si uživatel přeje z Pluginu přijímat pouze pixely.



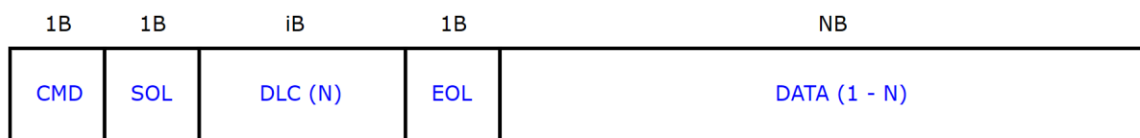
Obr. 28: GUI Obslužného programu s výsledky klastrování charakteristického rentgenového záření železa. Dosažený hitrate 0.18 MHit/s je vidět ve spodní sekci „Online statistics“

Bezpečnost komunikace mezi vlákny je řešena obdobným způsobem jako u Pluginu. Zpomalení z důvodu synchronizace vláken „locky“ se neprojevuje, jelikož zde neběží více stále vytížených na sobě závislých vláken. Jediným případem synchronizace je předávání dat mezi GUI a Online vláknem, což je elegantně vyřešeno Eventy Qt frameworku.

Navržené GUI Obslužného programu je vidět na Obr. 28.

4.4 Komunikační protokol

Komunikační protokol je založen na modelu Master/Slave. Každá zpráva přenášená přes TCP připojení má následující formát.



Obr. 29: Formát datové zprávy

Každá zpráva začíná bajtem CMD, který určuje účel zprávy. Zprávy mají flexibilní délku danou bajty DLC, přičemž DLC může být dlouhé 1 až 8 B. DLC je opatřeno bajty SOL (začátek DLC) a EOL (konec DLC), díky čemuž algoritmus pozná, kde začíná a končí DLC. To je klíčové z důvodu, že DLC může mít samo o sobě proměnnou délku. Na základě DLC je poté čteno N bajtů dat. Samotná DATA už mohou obsahovat například kompletní klastry odesílané z Pluginu do Obslužného programu.

Možné CMD jsou následující:

- 'K' – Příkaz (změna zvolené funkce) pro Slave;
- 'A' – Acknowledge od Slave (jen v reakci na příkaz);
- 'V' – Nastavení parametrů klastrování pro Slave;
- 'E' – Chybová zpráva;
- 'M' – Textová zpráva (např. výpis do logu pro uživatele);
- 'C' – Výstup klastrování (kompletní klastry) od Slave;
- 'P' – Výstup pixelů od Slave;
- 'H' – Výstup histogramu energií od Slave;
- 'N' – Výstup čítače pixelů pro „Pixel Counting“ režim od Slave.

Uveďme příklad spuštění online klastrování. Uživatel se připojí přes Obslužný program k Pluginu a stiskne tlačítko pro klastrování. Následně se odešle zpráva s CMD = 'K' a polem DATA obsahujícím příkaz pro spuštění klastrování "-CLSTR". Plugin zprávu

přijme a spustí režim klastrování, po jehož spuštění odešle zprávu Obslužnému programu s $CMD = 'A'$ a polem $DATA = "-CLSTR"$. Tím Plugin potvrzuje spuštění režimu klastrování a Obslužný program notifikuje uživatele o úspěšném spuštění. Po spuštění měření následují zprávy s výsledky klastrování, tedy $CMD = 'C'$ a $DATA$ obsahující serializované klastry. Samotnou serializaci klastrů popisuje navržený serializační Algoritmus 5.

Algoritmus 5: Algoritmus pro serializaci kompletních klastrů

Vstup: pole $closedClusters$ (obsahující kompletní klastry C_k)

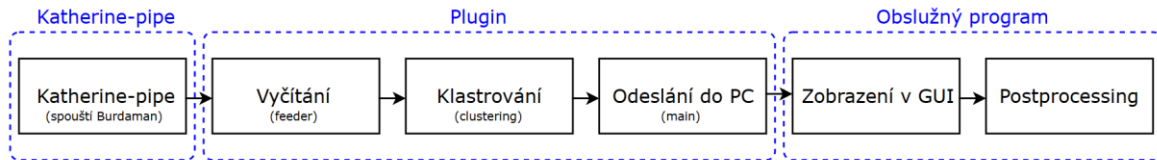
Výstup: znakový řetězec $std::string S$

```
1:  foreach (klastr  $C_k$  in  $openClusters$ )
2:      foreach (pixel  $p$  in  $C_k$ )
3:           $S.append(p.X + "\t")$ 
4:           $S.append(p.Y + "\t")$ 
5:           $S.append(p.ToT + "\t")$ 
6:           $S.append(p.ToA)$ 
7:          if ( $p \neq C_k.end()$ )
8:               $S.append(",")$ 
9:       $S.append(";")$ 
```

Serializace převádí pole $closedClusters$ na znakový řetězec S , kde jsou klastry C_k separovány středníkem ";", pixely p klastru jsou separovány čárkou ",", a vlastnosti pixelu jsou separovány tabulátorem "\t". Znaky se přidávají pomocí funkce $append()$. Takto serializovaná data odešle Plugin do Obslužného programu, který provede deserializaci (inverzní operaci vůči Algoritmu 5). Serializace výstupů ostatních funkcí (Pixel Counting, Akvizice Histogramů atd.) je podobná, vždy je totiž zachováno pravidlo oddělování úrovní jednotlivých vlastností pixelů, celých pixelů, a klastrů. Serializací do binárního formátu namísto textového lze dále optimalizovat výkon serializace.

4.5 Implementace klastrování v Katherine

Na Obr. 30 je vidět postup akcelerace klastrování v Katherine od detekce pixelů po post-processing v Obslužném programu.



Obr. 30: Postup akcelerace klastrování v Katherine. Program Katherine-pipe je spuštěn programem Burdaman, Plugin provede klastrování a odešle data Obslužnému programu

Nejprve je hit převeden na pixel v bloku „Vyčítání“. Následně je po filtraci umístěn do Pixelového bufferu a odeslán ke klastrování. Zpracování hitu popisuje Algoritmus 6.

Algoritmus 6: Zpracování příchozího hitu

Vstup: zpráva m z Pipe obsahující hit; parametry klastrování K ; časový offset $time_offset$

Výstup: pixelový buffer P

- 1: **if** ($PixelBufferFull() = true$)
 - 2: Zahod'Zprávu(m)
 - 3: **return**
 - 4: $pixel = Převed'HitNaPixel(m, time_offset)$
 - 5: **if** ($pixel \neq K.oblastFiltrování$)
 - 6: $P \leftarrow pixel$
-

Algoritmus 6 ukazuje, že hit musí splnit dvě podmínky pro umístění do pixelového bufferu. Nejprve nesmí být pixelový buffer plný $PixelBufferFull()$, jinak je hit zahozený (řízení výkonu klastrování viz Kapitola 4.6). Poté je hit převeden na pixel funkcí $Převed'HitNaPixel(m)$, která převede 6 B zprávu na pixel. Následně pokud není v oblasti filtrování (uživatelé definovaná oblast, kde pixely nejsou přijímány), je umístěn do pixelového bufferu P . Převedení hitu na pixel je následovné.

```
uint32_t x = (m >> 28) & 0xFF;  
uint32_t y = (m >> 36) & 0xFF;  
uint32_t ToT = (m >> 4) & 0x3FF;  
uint64_t course_toa = ((m >> 14) & 0x3FFF) + (time_offset * 16384);  
uint32_t fine_toa = m & 0xF;  
double ToA = (course_toa * 25) - (fine_toa * 1.5625);
```

Převedení hitu, resp. zprávy na pixel je docíleno bitovými posuvy a maskováním bitů. Finálně je pomocí $course_toa$ a $fine_toa$ vypočteno výsledné ToA pixelu (1.1). Do $course_toa$ je započítán časový offset $time_offset$, který je průběžně aktualizován zprávami z Katherine-pipe.

Po vyčítání následuje blok „Klastrování“. Klastrování provádí optimalizovaný algoritmus „baseline“ popsany dříve. Z hlediska architektury se algoritmus nachází ve třídě „clustering“ (Obr. 27) a je volán z nekonečné smyčky Plugin vlákna.

Proti dříve popsanému algoritmu „baseline“ je navíc implementováno filtrování klastrů podle velikosti. Uživatel si volí, jaká velikost klastrů ho nezajímá, čímž šetří paměťové prostředky spolu s propustností Ethernetu. Proces je popsán viz Algoritmus 7, přičemž podmínka viz řádek „1“ nahrazuje podmínku na řádku „6“ viz Algoritmus 2.

Algoritmus 7: Filtrování klastrů podle velikosti

Vstup: pole *openClusters* (obsahující klastry *C*); parametry klastrování *K*

Výstup: pole *closedClusters* (obsahující kompletní klastry *C_k*)

```
1:   if (p.ToA – C.ToA > K.Dmax)
2:       if (K.FilterOn)
3:           if (C.ClusterSize < K.FilterSize)
4:               ZahodKlastr(C)
5:               continue
6:       closedClusters ← OdešliKompletniKlastr(C)
7:       continue
```

Kompletní seznam nastavitelných parametrů klastrování je pro úplnost následující: filtrování pixelů mimo definovanou oblast; filtrování klastrů podle velikosti; Δt ; D_{max} .

4.5.1 Maximální dosažitelný výkon klastrování

Dvoujádrový ARM procesor v Katherine Gen2 bude zcela jistě dosahovat nižšího výkonu klastrování, než testovací sestava viz Kapitola 3.6.

Pro zjištění maximálního dosažitelného výkonu Katherine by nebylo vhodné, aby běžela všechna tři vlákna Pluginu (Obr. 27), která by procesor zbytečně zatěžovala. Je tedy možné načíst pixelová data z předem připraveného datasetu a provést offline měření výkonu.

K měření výkonu budou použita stejná pixelová data viz Tabulka 3. Kvůli nedostatku paměti RAM budou datasety B, C a D zredukovány na maximálně 100 MB (datasety B a C zredukovány na cca 70 % původní velikosti, dataset D zredukován na 1,7 % své původní velikosti). Nejlepší porovnatelnost lze čekat od datasetů A a E.

Tabulka 12 potvrzuje předchozí hypotézu, dvoujádrový procesor ARM dosahuje i se všemi optimalizacemi přinejlepším 12 % výkonu stejného algoritmu běžícího na PC. Sledujeme stejný trend výkonu, výkon algoritmu klesá s velikostí jednotlivých klastrů (datasety B a C). Přesnost klastrování lze porovnat jen u nezredukovaných datasetů (A a E) a shoduje se s předchozími výsledky.

Tabulka 12: Naměřený jedno vláknový výkon klastrování na Katherine Gen2 (aritmetický průměr 5 měření)

Dataset	A	B	C	D	E
Čas (ms)	24,4	56 969	32 455	6 758	4 762
Výkon (MHit/s)	0,64	0,07	0,15	0,59	0,67
Výkon oproti PC (%)	12	7	5	11	9

Dále bude cílem dosáhnout výkonu klastrování za běhu měření co nejvíce se přibližující **0,67 MHit/s**. Jelikož Plugin poběží ve třech vláknech, a další vlákno užívá program Katherine-pipe, lze u dvou-jádrového procesoru ARM čekat dle mého odhadu přinejlepším 50 % výkonu oproti maximálnímu.

4.6 Dynamické řízení výkonu klastrování

Výsledky maximálního výkonu ukázaly, že výkon klastrování na Katherine Gen2 bude pod 1 MHit/s. Teoretická propustnost Katherine Gen2 je ovšem až 16 MHit/s [2]. To znamená nutnost řízení výkonu, aby nebyl Plugin zahlcen příliš velkým množstvím hitů. Plugin musí být schopen řídit propustnost hitů nehledě na okolnostech měření.

Výkon lze řídit hlavními výpočetními vlákny programu, tedy vlákem Vyčítacím a Plugin vláknem. Řídící veličinou bude v tomto případě počet pixelů v Pixelovém bufferu mezi vlákny (Obr. 27), přičemž Vyčítací vlákno musí zajistit neustálý odběr hitů z Pipe.

Vyčítací vlákno umísťuje pixely do pixelového bufferu, a tím dokáže kontrolovat maximální počet pixelů v bufferu. To zajišťuje funkce *PixelBufferFull()* viz Algoritmus 6. Tím je zajištěno, že Plugin vlákno nebude zahlceno narůstajícím počtem pixelů. Tímto je zároveň kontrolováno využití RAM pixelovým bufferem. Implementace funkce *PixelBufferFull()* je následovná:

- Přečtení počtu pixelů v Pixelovém bufferu;
- Jeli v bufferu více než N_{max} pixelů, je přidávání pixelů do bufferu zakázáno do doby, než Plugin vlákno stihne zpracovat aspoň $N_{max}/2$ pixelů;
- Po opětovném povolení zápisu do bufferu funkce *PixelBufferFull()* vrátí *false* a další pixel je zapsán do bufferu.

Číslo N_{max} je zvoleno tak, aby byl Plugin šetril paměť RAM. Při vyhrazení max. 30 MB na pixelový buffer bylo N_{max} zvoleno na 1 966 080 pixelů (jeden pixel má 16 B).

Ze zvoleného N_{max} můžeme vypočítat využití paměti poli *openClusters* a *closedClusters* v Plugin vláknem. Nejhorší scénář využití paměti bude v případě, pokud z každého pixelu

vznikne pouze jeden klastr. Jeden klastr s jedním pixelem je 36 B velký, tudíž bude využití paměti klastry 67,5 MB.

Celkové využití paměti Pix. bufferem a klastry bude tedy menší než 100 MB. Díky tomu Katherine zůstane velká část prostředků na ostatní programy. Vytknutím t_o z výpočtu (2.1) můžeme určit maximální periodu odesílání klastrů do Obslužného programu za předpokladu výkonu 1 MHit/s (4.1).

$$t_o = \frac{P}{C_{Bytes} \cdot H_{max}} = \frac{100 \text{ MB}}{36 \text{ B} \cdot 1 \text{ MHit/s}} = 2,91 \text{ s}, \quad (4.1)$$

kde H_{max} je maximální hitrate, P je velikost dostupné paměti RAM pro ukládání klastrů, C_{Bytes} je velikost jednoho klastru s jedním pixelem v bajtech, a t_o je perioda odesílání kompletních klastrů do PC.

Výpočtem jsme opět potvrdili, že limitujícím faktorem výkonu bude procesor. To je proto, že uživatel bude chtít nová data vidět na obrazovce několikrát za sekundu. Pokud určíme periodu odesílání na 20 ms, se zvoleným N_{max} bude garantovaná maximální spotřeba paměti pluginu menší než 100 MB (při ignorování ostatních bufferů a kódu programu). V případě Katherine Gen2 půjde o pouhé 5 % využití RAM, což přispěje k robustnosti a použitelnosti Pluginu.

Podobně lze vypočítat, jak dlouho může v případě nečinnosti Vyčítací vlákno spát, tak aby nezatěžovalo zbytečně procesor a zároveň stíhalo vyčítat veškeré zprávy z pipe. Pipe má omezenou kapacitu na 65 536 B (ve výchozím nastavení Linuxu) [19]. Pro danou kapacitu Pipe a hitrate 1 MHit/s bude maximální čas spaní vlákna roven 1,82 ms (obdoba (4.1)). To je poměrně krátký čas. Pro vyšší hitrate bude v budoucnosti nutné, aby byla kapacita Pipe programem Katherine-pipe výrazně navýšena. Je nutné dodat, že vyčítací vlákno může být uspáno také plánovačem operačního systému (nepředvídatelné), což by mohlo vyústit v ucpání pipe.

Plugin vlákno bude při vysokých hitrate nejvytíženějším vláknem Pluginu. Zatížení procesoru lze snížit v případě nízkého hitrate. Toho Plugin vlákno dosahuje uspáváním, pokud je pixelový buffer prázdný (nedochází k tzv. busy waiting). Implementace je následující:

- Pokud vlákno detekuje 10x v radě prázdný pixelový buffer, je uspáno na 1 ms;
- Pokud jsou detekovány pixely v bufferu, je čítač vynulován.

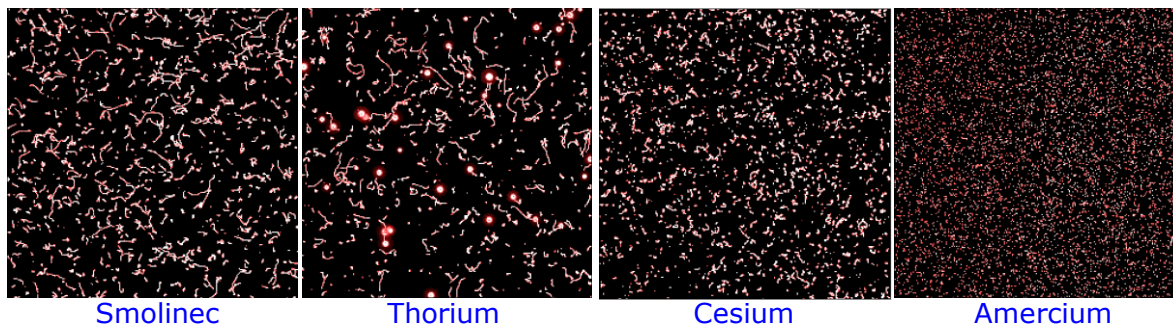
4.7 Výsledný výkon klastrování

Implementovaný systém byl prakticky otestován na Katherine Gen2 v Laboratoři fyzikální instrumentace na FEL ZČU. Zařízení Katherine Gen2 bylo umístěno do olověné komory a test klastrování byl proveden na několika zdrojích IZ.

Tabulka 13: Výsledky klastrování prvního praktického měření

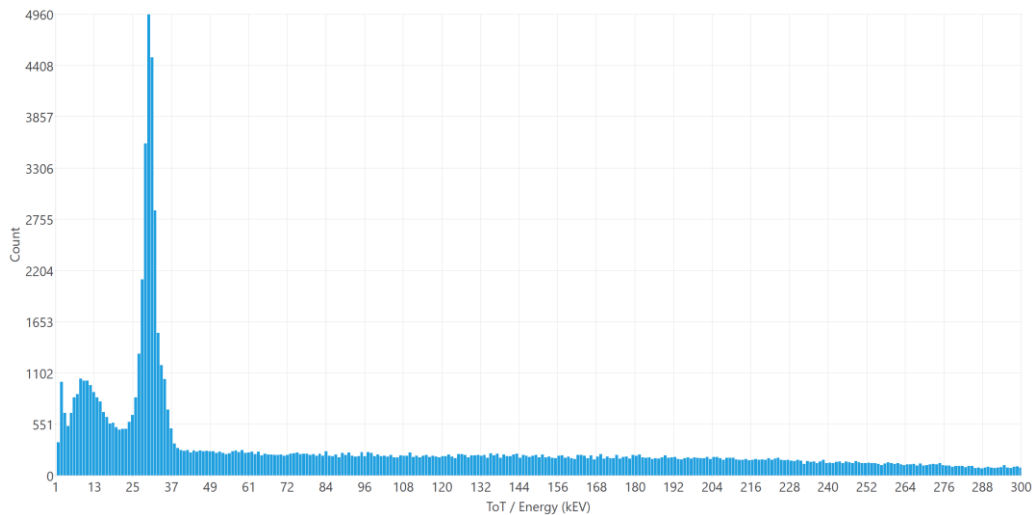
Zdroj IZ	Prům. hitrate (kHit/s)	t (s)	Počet hitů (kHit)	Počet klastrů (kCl)
Smolinec	2,4	456	1 110	100
Thorium	1,8	80	142	10
Cesium 137	2,2	240	511	101
Americium 241	0,63	241	150	103
Rentgenka1	96	4	379	324
Rentgenka2	120	X	X	X

Na Obr. 31 lze vidět snímky obsahující jednotlivé klastry z měření viz Tabulka 13. Téměř všechna měření proběhla podle očekávání, klastrování bylo provedeno v Katherine beze ztráty hitů. Zdroj Americium 241 měl intenzitu IZ 300 kBq a byl ve 2 cm vzdálenosti od detektoru. Rentgenka1 byla nastavena na 18 kV a 50 μ A. Rentgenka2 byla nastavena na 18 kV a 70 μ A. U měření „Rentgenka2“ se při vysokém hitrate projevila nestabilita (viz další kapitola), která způsobila ztrátu zbylých dat.



Obr. 31: Snímky detekovaných částic z měření

Kalibrací energií lze následně zkoumat energetické spektrum jednotlivých zdrojů IZ. U prvku Cesium 137 přes rozpad na Baryum byl charakteristický vrchol histogramu na energii 31 KeV (Obr. 32). Energie vrcholu se shoduje s předpoklady [20][21] což naznačuje, že kalibrace energií a klastrování funguje korektně.



Obr. 32: Histogram energií při měření prvku Cesium 137

4.7.1 Nestabilita při vysokých intenzitách IZ

Problém nastal při vysokém hitrate. Vysoká intenzita IZ při druhém měření na rentgence způsobila tak vysoký hitrate (přes 100 kHit/s), že se mezi programem Katherine-pipe a Plugin zaplnila Pipe, po které jsou předávány hity. Důvodem byla malá velikost Pipe (65 536 B). Pomocí vzorce (4.1) lze vypočítat, že Pipe je při 100 kHit/s zaplněna za 18,2 ms. Plugin nedokázal během této doby přijmout příchozí hity z Pipe, což bylo způsobeno následujícím:

- Katherine-pipe nekontrolovala zaplnění Pipe a zapisovala další hity nehledě. Protože hit umístěný na Pipe má velikost 6 B, lze hitů do Pipe o velikosti 65 536 B umístit přesně 10922,66. To znamená, že po zaplnění je do Pipe umístěn necelý hit tzn. 4 B a vznikne chyba rámce;
- Vyčítací vlákno Pluginu se z důvodu nutnosti synchronizace `std::mutex::lock()` pro zápis pixelu do vstupního bufferu Pixelového bufferu zastavilo na příliš dlouhou dobu a nestihlo další hity z Pipe přijmout.

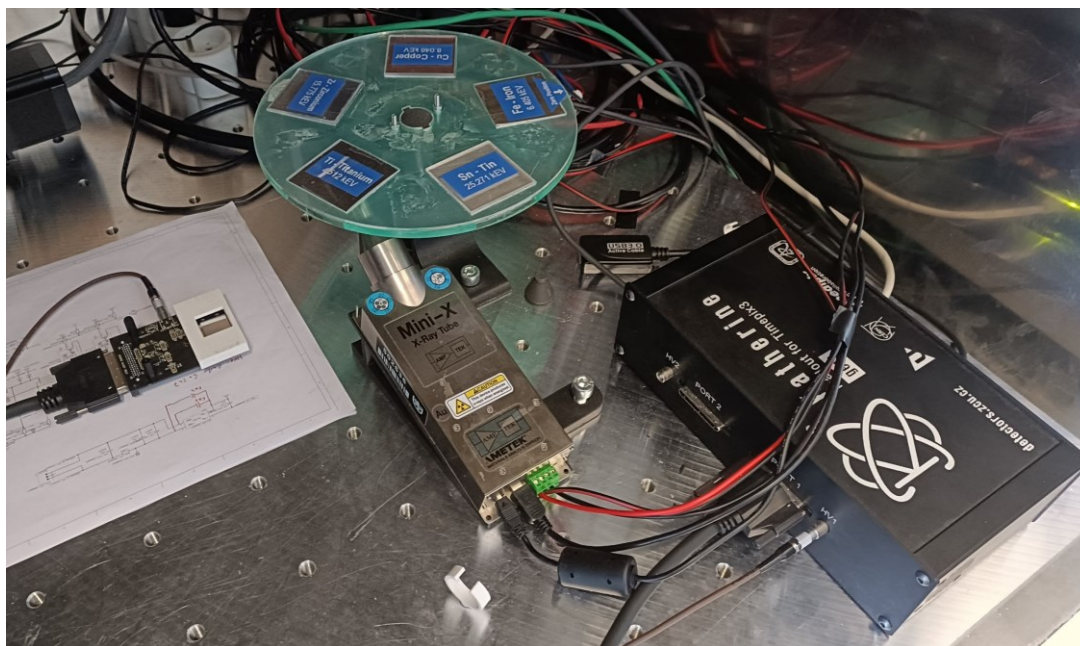
Řízení výkonu viz Kapitola 4.6 tuto chybu nepodchytilo. Jednotlivé problémy byly vyřešeny následovně (ve shodném pořadí):

- Katherine-pipe zapisuje hity do Pipe pouze, je-li v Pipe alespoň 6 B volné paměti. Zároveň je zvětšena kapacita Pipe;
- Vyčítací vlákno eliminuje čekání na synchronizaci tím, že Pixelový buffer nepoužívá `std::mutex::lock()` pro vstupní vnitřní buffer. Správa Pixelového bufferu je tedy plně předána Vyčítacímu vláknu. Tím se výkonová taxa locku snižuje.

4.7.2 Finální výkon klastrování

Po vyřešení problému nestability programu při vysokých hitrate byl změřen dosažený výkon klastrování. Intenzita IZ (hitrate) při měření byla řízena nastavením rentgenky, která generovala charakteristické rentgenové záření odrazem od daného prvku.

Maximální dosažený výkon klastrování **201 kHit/s** (176k klastrů/s) byl dosažen při měření charakteristického rentgenového záření mědi. Tento výkon byl dosažen za nepřímo mířící rentgenky (Obr. 33) s nastavením napětí na 16 kV a proudu 60 μ A.

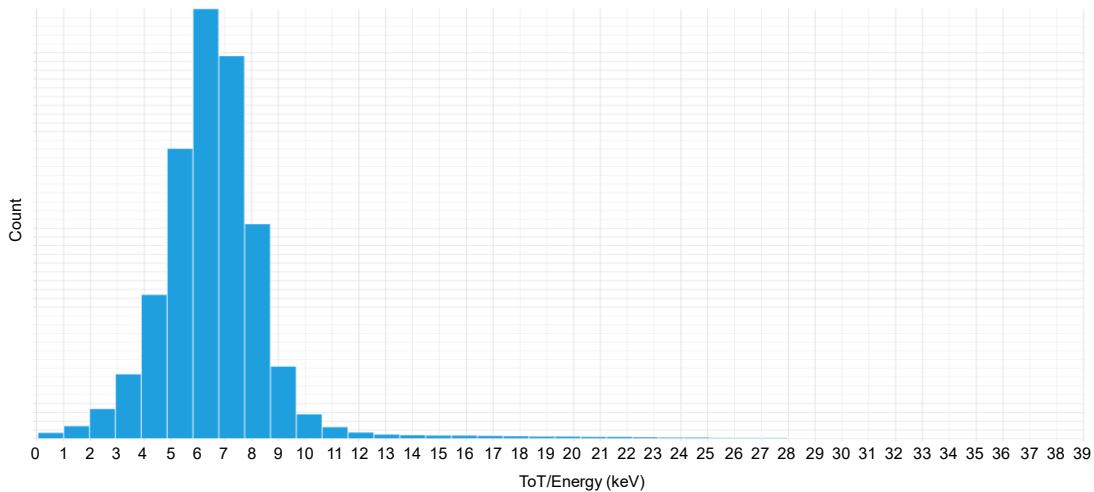


Obr. 33: Měření dosaženého výkonu klastrování. Popis: Katherine Gen2 v pravém dolním rohu; detektor Timepix3 vlevo uprostřed; rentgenka mířící na karusel vybraných prvků uprostřed

Při takto vysokém hitrate byl výkon poněkud kolísající. Hlavním limitujícím faktorem byl počet jader procesoru, neboť 4 běžící vlákna (3 vlákna Pluginu a 1 vlákno Katherine-pipe) na dvoujádrovém ARM procesoru závodila o přidělený výpočetní čas, načež krátkodobě klesal výkon. Kolísání výkonu bylo též způsobené posíláním dat přes TCP protokol, kde Plugin závodil s Katherine-pipe o přístup k Ethernetovému řadiči přes Socket API.

Uvedené problémy lze v budoucnu jednoduše vyřešit použitím vícejádrového procesoru v Katherine další generace, a zakázáním odesílání dat z Katherine-pipe na Ethernet, pokud běží Plugin. Momentálně je stabilní výkon cca **150 kHit/s**.

Obr. 28 ukazuje výsledné GUI Obslužného programu a dosažený výkon 180 kHit/s z měření charakteristického rentgenového záření železa, které je zobrazeno na histogramu viz Obr. 34. Histogram ukazuje charakteristický vrchol železa na energii 6,4 keV shodující se s teoretickými předpoklady [21].



Obr. 34: Histogram energií železa zobrazující charakteristické rentgenové záření o energii 6,4 keV

Dosažený výkon klastrování na Katherine Gen2 při měření tedy dosahuje **30 %** dosažitelného maximálního výkonu (Kapitola 4.5.1). Zvýšením počtu jader na 4 by teoreticky mělo být dosaženo téměř 100 % výkonu. Další zvýšení počtu jader umožní paralelizaci klastrování, která už při dvou výpočetních vláknech klastrování může dosahovat téměř dvojnásobného výkonu (Kapitola 3.9).

4.8 Začlenění implementovaného algoritmu do stávajících měřících řetězců

Při dosaženém výkonu klastrování 201 kHit/s na Katherine Gen2 lze očekávat, že výkon bude na starší verzi Gen1 podobný, jelikož je rozdílná pouze velikost RAM a verze Linuxu. Dosažený výkon dává široké možnosti využití.

Před touto diplomovou prací bylo možné provádět klastrování pouze na PC. Detekované hity byly přenášeny po Ethernetu do PC, kde bylo možné následně provádět klastrování. To představovalo nároky na výpočetní výkon a paměť PC. Implementace klastrování pomocí Pluginu a Obslužného programu přesouvá tyto výpočetní nároky z PC na Katherine. To má následující efekty.

- Klastrování v zařízení šetří výpočetní výkon a paměť PC;
- Pixel Counting a Akvizice histogramů šetří propustnost Ethernetu a paměť PC;
- Možnost filtrace klastrů podle velikosti přímo v zařízení;
- Uživateli je umožněno analyzovat výsledky klastrování online při měření;
- Možnost logování výsledků minimalistickými platformami a tvoření IoT sítí.

Využití může být zajímavé pro minimalistické platformy typu Raspberry Pi. Do Katherine stačí odeslat pakety pro začátek/konec měření, a i málo výkonné zařízení může logovat

výsledky klastrování (klastry, histogramy atd.). To otevírá dveře vytvoření rozsáhlých IoT sítí, které mohou monitorovat IZ na mnoha místech a rozeznávat případné anomálie a zajímavé události. Zajímavé události by mohly být odesílány do velína, kde by bylo možné je sledovat téměř v reálném čase.

Implementovaný algoritmus nalezne své využití v každém případě, kdy není maximální výkon klastrování zásadní pro prováděné měření. Pokud se tedy bude jednat o měření slabších polí s nižší hitrate (než je dosažený výkon), bude možné získávat výsledky měření beze ztráty dat. Využití algoritmus nalezne i pro vyšší hitrate, protože algoritmus implementuje dynamické řízení výkonu (Kapitola 4.5). Dojde sice ke ztrátě dat úměrné překročení maximálního výkonu algoritmu, ale to nemusí při určitých měřeních vadit.

Tabulka 14: Rozhodovací tabulka pro výběr způsobu akcelerace klastrování

Akcelerace klastrování	Měření slabého pole	Měření pole do hitrate 201 kHit/s	Měření vyšších hitrate (povolena ztráta dat)	Měření vyšších hitrate (beze ztráty dat)
Plugin	ANO	ANO (preference)	ANO (preference)	NE
Burdaman	ANO	ANO	ANO	ANO

Tabulka 14 ukazuje, kdy je výhodné využít klastrování v hardware. Je to zejména při vysokém hitrate, kdy se ušetří velká část prostředků PC. Ještě výhodnější je situace, kdy uživatele zajímají pouze klastry nad/pod určitou velikost. Pokud uživatele například zajímají protony (částice o velikosti několika desítek až stovek pixelů), je velice výhodné odfiltrovat klastry menší než 10 pixelů, čímž jsou efektivně odfiltrována nezajímavá data. Úplně nejvýhodnější situací je, pokud uživatele zajímají pouze energetické histogramy částic. V takovém případě se zatížení Ethernetu a využití paměti PC sníží na minimum, protože jsou odesílány pouze 2 bajtové proměnné na jeden klastr, který může jinak obsahovat stovky pixelů. Při dlouhém měření to může znamenat ušetření gigabajtů paměti.

4.9 Možnosti budoucího zvýšení výkonu klastrování

V této kapitole zmíníme budoucí možnosti zvýšení výkonu klastrování v hardware, které jsou sumarizovány viz Tabulka 15. Největšího zvýšení výkonu by se dosáhlo paralelizací klastrování. Paralelizaci lze provést v FPGA (Kapitola 3.10) nebo CPU (Kapitola 3.9), obě možnosti mají vysoký potenciál zvýšení výkonu (viz Obr. 22), ale také vysoké nároky na hardware. Stávající 2 jádrový ARM procesor paralelizací klastrování nezrychlí, bylo by nutné použít alespoň 4 jádrového procesoru, lépe 8 jádrového pro dosažení signifikantních

výsledků. Paralelizace v FPGA by zase představovala spolu s vysokými HW nároky (velký počet LE) také velkou náročnost implementace.

Další možností je volba samotných algoritmů klastrování. První možností je implementovat společně s „baseline“ také algoritmus „quadtree“, který za určitých podmínek může dosahovat lepších výsledků. Plugin by tedy mohl dynamicky přepínat mezi algoritmy podle potřeby.

Další možností je implementace klastrování pomocí algoritmů strojového učení jako DBSCAN (Kapitola 2.2.3). Tyto algoritmy fungují na principu aproximace (neprodukují deterministické výsledky), ale vhodně navržený model může dosahovat zajímavých výkonů díky nižší složitosti hledání sousedů $O(n * \log(n))$ [7]. Výzvou implementace bylo vhodné nastavení hyperparametrů modelu pro co nejpřesnější klastrování. Volba hyperparametrů ovšem znamená kompromis mezi výkonem a přesností klastrování.

Zvýšení výkonu by představovala také kalibrace energií pomocí LUT. Kalibrace pomocí LUT je dvakrát rychlejší než výpočet (Kapitola 3.7), zato má vysoké nároky na paměť RAM. Implementace je prakticky dokončená, ale pro využití by bylo vhodné zvýšení RAM Katherine. V Katherine Gen1 (1 GB RAM) by použití mohlo omezovat ostatní programy.

Tabulka 15: Možnosti zvýšení výkonu s jejich náročností implementace, nároky na HW a odhadovaný potenciál vylepšení

Způsob vylepšení	Náročnost implementace	Nároky na HW	Potenciál vylepšení
Paralelizace v FPGA	Velmi vysoká	Vysoké	Vysoký
Paralelizace v CPU	Střední	Vysoké	Vysoký
DBSCAN do Pluginu	Střední	Střední	Střední
Quadtree do Pluginu	Nízká	Střední	Nízký
Kalibrace s LUT	Nízká	Středně/Vysoké	Nízký

Zlepšením uživatelské zkušenosti by byla plná integrace programu Plugin do programu Katherine-pipe (plná integrace do měřícího řetězce). Tím odpadne nutnost využívání Pipe, čímž se zvýší rychlost a efektivita předávání hitů. Také se zvýší robustnost řešení a zjednoduší se dynamické řízení výkonu. Plná integrace bude vyžadovat spojení projektů Plugin a Katherine-pipe. Znamená to migraci Katherine-pipe z jazyka C do C++, pro referenci poslouží Kapitola 4.2. Výsledný program by byl spuštěn z programu Burdaman, do kterého by se implementovala architektura Obslužného programu.

Zhodnocení a závěr

Hlavním cílem práce bylo implementovat akceleraci klastrování ve vyčítacím zařízení Katherine. Důraz byl kladen na výkon klastrování a dosažení nejvyššího možného hitrate.

Nejprve byla představena potřebná fyzikální instrumentace a dostupný hardware Katherine. Následně byly představeny principy klastrování a byl definován algoritmus „baseline“ vycházející přímo z definice. Dále byl představený algoritmus „quadtree“ vycházející z diplomové práce [4]. Další dva algoritmy strojového učení nevyhovovaly z hlediska přesnosti klastrování. Byly tedy vybrány algoritmy „baseline“ a „quadtree“ a pro akceleraci byl vybrán dvoujádrový procesor ARM.

Následoval návrh testovacího programu na PC pro porovnání vybraných algoritmů na datasetech z reálných měření. Algoritmus „baseline“ byl optimalizován důmyslným vyjmutím časové a sousedské podmínky z výpočetních smyček, čímž jeho výkon vzrostl o 124,7 % (geometrický průměr). Optimalizovaný „baseline“ byl v průměru o 40 % rychlejší než „quadtree“ za cenu o 5 % větších nároků na RAM. Bylo zjištěno, že „baseline“ je rychlejší při klastrování menších částic, zatímco „quadtree“ je rychlejší při klastrování velmi velkých částic. Pro finální implementaci do Katherine byl vybrán algoritmus „baseline“.

Také byla diskutována akcelerace kalibračního procesu a bylo dosaženo více jak 2.5x urychlení kalibrace za použití LUT. Nevýhodou LUT bylo velké využití paměti RAM.

Dále byly diskutovány možnosti paralelizace klastrování. Nejvhodnějším způsobem byla časová paralelizace, což také vyplynulo z diplomové práce [16]. Paralelní akcelerace byla diskutována pro procesor a FPGA. Na PC byl algoritmus „baseline“ paralelizován do 16 vláken, což zvýšilo výkon klastrování 7x. Naneštěstí Katherine Gen2 s dvoujádrovým procesorem nemá pro paralelizaci klastrování potenciál.

Následně byla detailně popsána implementace algoritmu „baseline“ v Katherine Gen2. Algoritmus byl implementován v C++14 programem Plugin, který provádí klastrování na jednom výpočetním vlákne procesoru. Další dvě vlákna jsou využita na vyčítání hitů a odesílání dat do Obslužného programu přes TCP transportní protokol.

Závěrem byl zjištěn maximální dosažitelný výkon klastrování 670 kHit/s pro daný procesor, kterému jsme se dále pokusili přiblížit. Klastrováním rentgenového záření bylo dosaženo finálního výkonu klastrování 201 kHit/s a byla ověřena korektnost výsledků kontrolou energetických histogramů vybraných prvků.

Dosažený výkon otevírá dveře širokému využití. Dříve bylo klastrování prováděno pouze v PC, což kladlo velké nároky na výkon. Za použití navrženého klastrování lze výkonové nároky distribuovat do jednotlivých zařízení Katherine. Tím se také otevírají možnosti využití minimalistických platforem, které by mohly ukládat velké množství dat rozsáhlých měřících řetězců a tvořit inteligentní IoT sítě. V budoucnu bude možné dále zvýšit výkon klastrování pomocí navržené časové paralelizace, jež má potenciál pro mnohonásobné zvýšení výkonu ve vhodném procesoru.

Literatura

- [1] Helebrant, J. *Pixelové polovodičové detektory Medipix/Timepix*. Online. Státní ústav radiační ochrany, 2019. Dostupné z: https://www.suro.cz/aplikace/ramesis-wiki/index.php/Pixelové_polovodičové_detektory_Medipix/Timepix [citováno 2023-12-15]
- [2] Burian, P. et al. Katherine: *Ethernet Embedded Readout Interface for Timepix3*. Journal of Instrumentation, 2017, 12.11: C11001
- [3] Buriann P. et al. *Multidetector Embedded Readout Interface for Timepix3*. Online. Faculty of Electrical Engineering, University of West Bohemia, 2021. Dostupné z: <https://indico.cern.ch/event/820476/contributions/4373155/> [citováno 2023-12-15]
- [4] Meduna, L. *Detecting elementary particles with Timepix3 detector*. Diplomová práce. RNDr. František Mráz, CSc (Vedoucí práce). Praha: Univerzita Karlova, Matematicko-Fyzikální Fakulta, 2019.
- [5] *Cloud Chamber* – Wikipedia. Online. Dostupné z: https://en.wikipedia.org/wiki/Cloud_chamber [citováno 2023-12-17]
- [6] Nabil El Malki; Franck Ravat; Olivier Teste. *KD-means: clustering method for massive data based on kd-tree*. 22nd International Workshop On Design, Optimization, Languages and Analytical Processing of Big Data - DOLAP 2020 -, 2020, Kodaň, Dánsko. fhal-03080514
- [7] Ester, M. et al. *A density-based algorithm for discovering clusters in large spatial databases with noise*. Proceedings of the 2nd international conference on knowledge Discovery and data mining, 1996, s. 226-231
- [8] *DBSCAN* – Wikipedia. Online. Dostupné z: <https://en.wikipedia.org/wiki/DBSCAN> [citováno 2024-01-05]
- [9] Abbasifard, M., Ghahremani, B. Naderi, H. *A Survey on Nearest Neighbor Search Methods*. International Journal of Computer Applications. 95, 2014. s. 39-52. doi:10.5120/16754-7073
- [10] Borgna, L. et al. *Accelerating the DBSCAN clustering algorithm for low-latency primary vertex reconstruction*. Online. Imperial College London, 2022. Dostupné z: <https://indico.cern.ch/event/1106990/contributions/4998133/> [citováno 2024-01-06]

- [11] *Arm Compiler for Embedded Arm C and C++ Libraries and Floating-Point Support User Guide*. Online. ARM developer, 2024. Dostupné z: <https://developer.arm.com/documentation/100073/0621/The-Arm-C-and-C---Libraries/ISO-C-library-implementation-definition/Standard-C---library-implementation-definition> [citováno 2024-01-26]
- [12] *Qt Features, Framework Essentials, Modules, Tools Add-Ons*. Online. Qt Group, 2024. Dostupné z: <https://www.qt.io/product/features> [citováno 2024-01-26]
- [13] P. Burian et al. *Burdaman*.
- [14] *Move – cppreference*. Online. Dostupné z: <https://en.cppreference.com/w/cpp/utility/move> [citováno 2024-02-09]
- [15] Li, B; Mukundan, R. *A Comparative Analysis of Spatial Partitioning Methods for Large-scale, Real-time Crowd Simulation*. WSCG 2013: Communication Papers Proceedings: 21st International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision in co-operation with EUROGRAPHICS Association, s. 104-111. ISBN 978-80-86943-75-6
- [16] Čelko, T. *Clustering hits and predictions in data from TimePixe3 detectors*. Diplomová práce. RNDr. František Mráz, CSc (Vedoucí práce). Praha: Univerzita Karlova, Matematicko-Fyzikální Fakulta, 2023.
- [17] *Options for Linking*. Online. GNU ORG. Dostupné z: <https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html> [citováno 2024-04-26]
- [18] Hall, B. *Beej's Guide to Network Programming: Using Internet Sockets*. Online. Nezávislá publikace, 2019. 180s. ISBN: 1705309909. Dostupné z: <https://beej.us/guide/bgnet/> [citováno 2024-04-26]
- [19] *Pipe*. Online. Linux manual page, 2023. Dostupné z: <https://man7.org/linux/man-pages/man7/pipe.7.html> [citováno 2024-04-26]
- [20] Fomitchev, M. *The Rich Physics of Cs-137 Gamma Spectrum*. Online. Maximus energy, 2020. Dostupné z: <https://maximus.energy/index.php/2020/10/24/the-rich-physics-of-cs-137-gamma-spectrum/> [citováno 2024-05-06]
- [21] *Interactive Periodic Table*. AMETEK Inc, 2024. Dostupné z: <https://www.edax.com/resources/interactive-periodic-table> [citováno 2024-05-13]

Seznam obrázků

Obr. 1: Jedna alfa částice a elektrony zachycené v mlžné komoře [5].....	- 2 -
Obr. 2: Struktura hybridního pixelového detektoru Medipix2, ze stejné rodiny zařízení jako Timepix3 [1].....	- 2 -
Obr. 3: Průběh měření detektoru [4].....	- 4 -
Obr. 4: Vyčítací zařízení Katherine Gen1 s připojeným Timepix3 detektorem nalevo ...	- 5 -
Obr. 5: Závislost ToT na energii (kalibrační křivka) pro jeden pixel detektoru [2]	- 6 -
Obr. 6: Zachycené částice na celé ploše detektoru (nalevo) a jedna vyznačená alfa částice (klast) po dokončení klastrování dat (napravo).....	- 8 -
Obr. 7: Typy sousedních pixelů. Hlavní pixel je červený, zatímco sousední pixely jsou žluté [4].....	- 9 -
Obr. 8: Znázornění rekurzivního vytváření stromu s velikostí listu 32x32 pix [4].....	- 12 -
Obr. 9: Závislost výkonu/paměti na velikosti listu [4]	- 13 -
Obr. 10: Znázornění definice sousedství. Každý bod má definovaný radius ϵ . Body A jsou tzv. <i>bezprostředními pixely</i> a tvoří jádro klastru. Body B a C jsou tzv. <i>dosažitelnými pixely</i> a stále jsou součástí klastru s body A . Bod N je <i>šum</i> , protože se nenachází v blízkosti žádného jiného bodu [8]	- 15 -
Obr. 11: Příklad výsledku algoritmu DBSCAN na smyšlených datech se dvěma klastry a šumem [8].....	- 16 -
Obr. 12: Příklad architektury algoritmu DBSCAN v FPGA [10]	- 18 -
Obr. 13: Pipeline testovacího prostředí pro měření výkonu klastrování	- 20 -
Obr. 14: Snímky datasetů pro testování. Nahoře typické klastry, dole ukázkové snímky dat- 22 -	
Obr. 15: Znázornění pixelů na časové ose pro kontrolu časové podmínky klastru. Modré pixely jsou součástí otevřeného klastru. Δt je časová hranice klastru. Pixely $P1$ a $P2$ časovou podmínku splní, zatímco pixel $P3$ nikoliv.....	- 24 -
Obr. 16: Znázornění pixelů v 2D prostoru pro kontrolu podmínky sousedství. Modré pixely jsou součástí otevřeného klastru. Zelené pixely podmínku plní a mohou se stát součástí klastru. Žluté pixely podmínku plní, ale nemohou se stát součástí klastru. Červené pixely podmínku neplní.....	- 25 -
Obr. 17: Procentuální redukce času klastrování pro jednotlivé datasety oproti předchozí verzi algoritmu.....	- 26 -

Obr. 18: Porovnání výkonů klastrování algoritmů baseline a quadtree v závislost na datasetu	- 31 -
Obr. 19: Hranice dvou časových oken A a B . Klastr C1 je uvnitř hranice, ale spojení nebude potřeba. Klastr C2 byl rozdělen mezi dvě vlákna, a bude muset být následně spojen	- 34 -
Obr. 20: Navazující výpočetní cykly s vyznačením nutnosti spojování. T_n jsou vlákna se svými indexy	- 36 -
Obr. 21: Architektura časové paralelizace klastrování na PC. Výpočetní cyklus (nahore); spojování (dole)	- 37 -
Obr. 22: Čas klastrování v závislosti na počtu vláken pro dataset B	- 37 -
Obr. 23: Architektura paralelního klastrování v SoC	- 39 -
Obr. 24: Model vláken klastrování v FPGA. Kompletní klastry (OK); přední hraniční klastry (F); zadní hraniční klastry (B)	- 39 -
Obr. 25: Vyčítací zařízení Katherine Gen2	- 40 -
Obr. 26: Závislost Pluginu a Obslužného programu na stávajícím měřicím řetězci	- 42 -
Obr. 27: Architektura systému. Plugin běžící v Katherine komunikuje s Obslužným programem přes TCP spojení	- 43 -
Obr. 28: GUI Obslužného programu s výsledky klastrování charakteristického rentgenového záření železa. Dosažený hitrate 0.18 MHit/s je vidět ve spodní sekci „Online statistics“	- 46 -
Obr. 29: Formát datové zprávy	- 47 -
Obr. 30: Postup akcelerace klastrování v Katherine. Program Katherine-pipe je spuštěn programem Burdaman, Plugin provede klastrování a odešle data Obslužnému programu.... -	49 -
Obr. 31: Snímky detekovaných částic z měření	- 53 -
Obr. 32: Histogram energií při měření prvku Cesium 137	- 54 -
Obr. 33: Měření dosaženého výkonu klastrování. Popis: Katherine Gen2 v pravém dolním rohu; detektor Timepix3 vlevo uprostřed; rentgenka mířící na karusel vybraných prvků uprostřed	- 55 -
Obr. 34: Histogram energií železa zobrazující charakteristické rentgenové záření o energii 6,4 keV	- 56 -

Seznam tabulek

Tabulka 1: Specifikace testovací sestavy	- 19 -
Tabulka 2: Rozdíly v klastrování na Katherine vs PC.....	- 20 -
Tabulka 3: Datasets pro testování algoritmů. Typy částic: A: Mix; B: Ionty olova s úhlem dopadu 60°; C: Protony; D: Mix; E: Elektrony s úhlem dopadu 30°. Velikost klastrů a <i>ToT</i> jsou uváděny ve formátu maximální hodnota / aritmetický průměr.....	- 21 -
Tabulka 4: Výsledky klastrování datasetů pomocí programu Burdaman.....	- 22 -
Tabulka 5: Výkon základního algoritmu „baseline“ (aritmetický průměr z 5 měření)..	- 23 -
Tabulka 6: Výkon po 1. optimalizaci algoritmu (aritmetický průměr z 5 měření)	- 24 -
Tabulka 7: Výkon po 2. optimalizaci algoritmu (aritmetický průměr z 5 měření)	- 25 -
Tabulka 8: Finální optimalizace algoritmu pomocí <i>std::move</i> (aritmetický průměr z 5 měření).....	- 26 -
Tabulka 9: Finální výkon algoritmu „quadtree“ (aritmetický průměr z 5 měření)	- 30 -
Tabulka 10: Funkce Pluginu implementované v zařízení Katherine.....	- 40 -
Tabulka 11: Funkce Obslužného programu	- 41 -
Tabulka 12: Naměřený jedno vláknový výkon klastrování na Katherine Gen2 (aritmetický průměr 5 měření)	- 51 -
Tabulka 13: Výsledky klastrování prvního praktického měření	- 53 -
Tabulka 14: Rozhodovací tabulka pro výběr způsobu akcelerace klastrování.....	- 57 -
Tabulka 15: Možnosti zvýšení výkonu s jejich náročností implementace, nároky na HW a odhadovaný potenciál vylepšení.....	- 58 -

Příloha A: Zdrojové kódy

Zdrojové kódy jsou umístěny v repozitáři na stránce GitHub.

Repozitář obsahuje dva projekty:

- Obslužný program pro PC: *pc_gui*;
- Plugin pro Katherine: *plugin*.

Dále jsou v repozitáři spustitelné soubory obou programů ve složce *releases*. Pro úspěšné TCP připojení obou programů je nutné, aby měl PC nastaven IP adresu ethernet adaptéru na "192.168.1.10". Poté je nutné spustit obslužný program a kliknout na tlačítko „connect“. Následně spustit program na zařízení Katherine. Poté by se měly zařízení spojit což je potvrzeno zeleným nápisem „Online Clustering“ v GUI (viz příloha B). Měření lze poté spustit programem Burdaman. Dále budou popsány nástroje, jejichž pomocí lze upravovat a znovu sestavit program.

Program *pc_gui* lze sestavit za pomoci:

- Visual Studio 2022 (kompilátor MSVC);
- Qt 6.4.0 s nainstalovaným pluginem QtCharts.

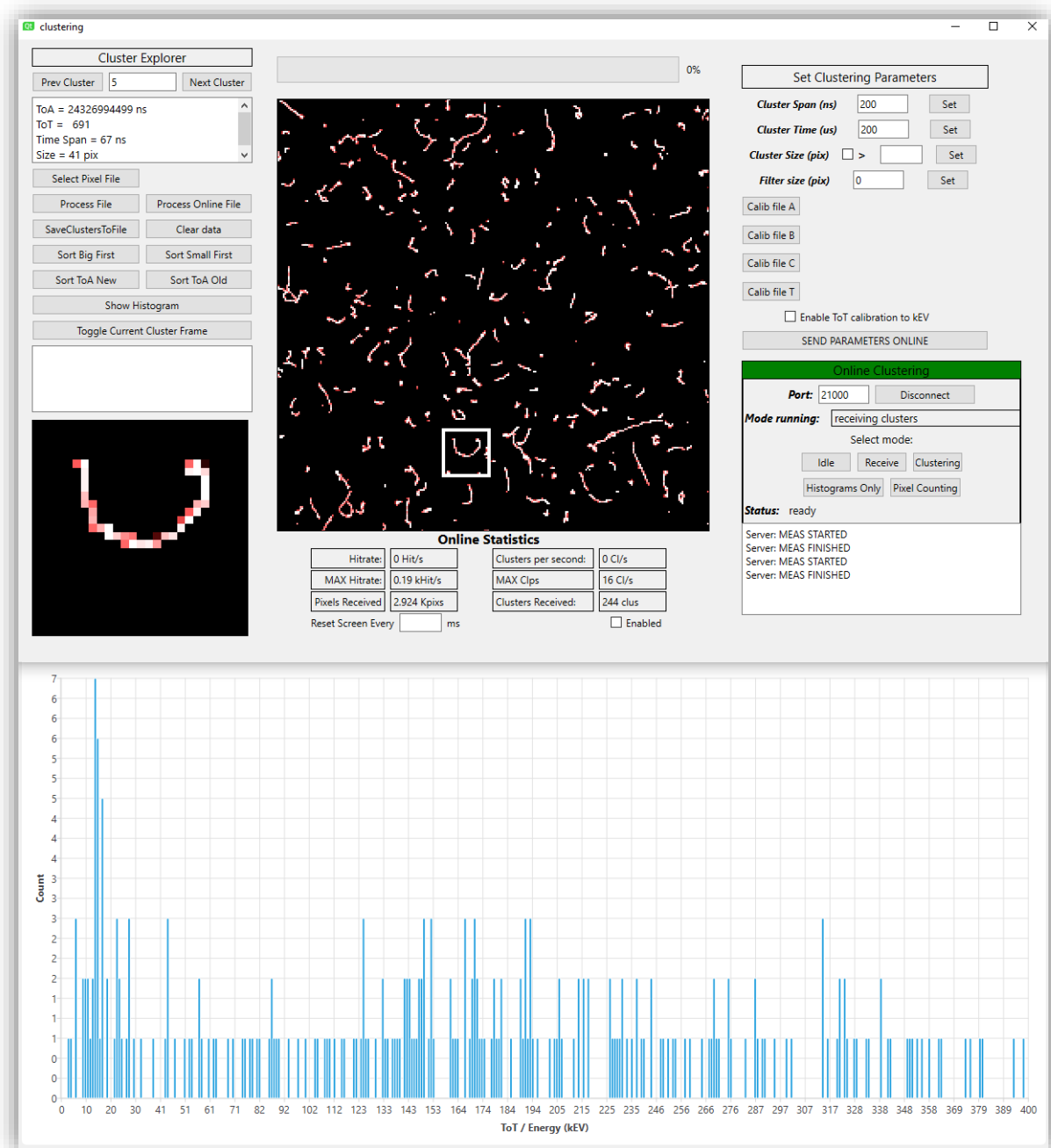
Program *plugin* lze sestavit za pomoci:

- Eclipse for DS-5 v5.29.1;
- Kompilátor GCC 4.x [arm-linux-gnueabi] (DS-5 built-in).

Repozitář je k dispozici na adrese: https://github.com/siverick/katherine_clustering

V případě jakýchkoliv dotazů mě neváhejte kontaktovat na emailu: richsivera@gmail.com

Příloha B: GUI Obslužného programu



Příloha C: Snímek elektrické svorkovnice pořízený režimem Pixel Counting

