Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

PORTABILITA' DELLE PRESTAZIONI DEGLI ALGORITMI DI RICOSTRUZIONE FISICA SU DISPOSITIVI ETEROGENEI UTILIZZANDO IL MODELLO DI ASTRAZIONE SYCL

Relatore:
Prof. RENZO DAVOLI

Presentata da:
LAURA CAPPELLI

Correlatori:
Dr. FRANCESCO GIACOMINI
Dr. ANDREA BOCCI

II Sessione
Anno Accademico 2019 - 2020

Indice

Introduzione

Il CERN è il centro di ricerca europeo che si occupa di fisica delle particelle con lo scopo di approfondire la conoscenza della materia e di studiare il funzionamento dell'universo. Le macchine utilizzate per la ricerca sono chiamate acceleratori di particelle: esse accelerano fasci di particelle, ad esempio protoni, a velocità prossime a quelle della luce per poi farli urtare in corrispondenza di complessi dispositivi chiamati rivelatori. L'energia che si sprigiona in seguito a una collisione si trasforma in nuove particelle, le quali si disperdono in tutte le direzioni attraversando i diversi strati del rivelatore.

Le collisioni, in particolare quelle più rare, sono di grande interesse per i fisici perché permettono di aumentare la conoscenza riguardo al mondo dell'infinitamente piccolo. In questo momento l'acceleratore di punta del CERN è il Large Hadron Collider (LHC), grazie al quale nel 2012 è stata annunciata al mondo la scoperta del bosone di Higgs. Per aumentare il potenziale di ricerca del CERN, è in atto un imponente aggiornamento di LHC e dei suoi rivelatori. Questo permetterà di produrre più collisioni per unità di tempo, che, a loro volta, permetteranno lo studio di eventi fisici ancora più rari.

Non solo l'hardware è sottoposto ad aggiornamento; anche il software deve essere modificato di conseguenza. Si consideri, infatti, che la quantità di dati collezionati ogni secondo dai rivelatori di LHC aumenterà considerevolmente a seguito del suo aggiornamento. Il software è responsabile sia di elaborare i dati al fine di ricostruire quello che avviene a seguito di ogni collisione, sia di filtrarli in tempo reale in modo da selezionare solo quelli contenenti possibile nuova fisica; tali algoritmi sono detti rispettivamente di

ricostruzione e di trigger.

Negli anni che ci separano dall'avvio del nuovo acceleratore, previsto per il 2027, si dovrà compiere un importante lavoro di ricerca e sviluppo che dovrà permettere di scrivere del software in grado di concretizzare il suo potenziale di nuove scoperte in fisica. È stato determinato che parte dell'aumento della potenza di calcolo necessaria sia da ricercare in risorse computazionali di tipo eterogeneo. Il software, quindi, deve essere eseguibile su un insieme di dispositivi di diversa natura, quali CPU, GPU e FPGA. Non è, però, sostenibile un approccio che preveda una diversa implementazione per ogni dispositivo che potenzialmente potrebbe essere utilizzato; occorre, invece, individuare una tecnologia che permetta di implementarne un'unica versione compatibile con quante più architetture possibili, mantenendo comunque delle prestazioni soddisfacenti su ognuna.

Questa tesi si pone l'obiettivo di approfondire SYCL [1, 2], un modello di programmazione astratto basato sullo standard ISO C++ [3] che consente di scrivere in un unico sorgente il codice da compilare ed eseguire su un insieme di dispositivi diversi. Una sua implementazione, Data Parallel C++, è oggetto di un progetto di ricerca dell'esperimento CMS, con l'obiettivo di valutare una sua eventuale adozione. I criteri di valutazione riguardano sia il soddisfacimento di adeguati livelli di prestazioni, sia l'effettiva portabilità su architetture diverse senza modifiche al codice sorgente.

Il capitolo 1 di questa tesi introduce brevemente il CERN e gli acceleratori di particelle. Il capitolo 2 approfondisce il programma di aggiornamento di CMS, uno dei quattro esperimenti del Large Hadron Collider; in questo capitolo si trova una descrizione del software utilizzato per la raccolta e l'analisi dei dati e una descrizione più dettagliata delle sfide che si dovranno affrontare alla luce del programma di aggiornamento di LHC. Segue lo studio di SYCL e delle sue implementazioni nel capitolo 3, mentre nel capitolo 4 è riportato il caso di studio che ha visto l'applicazione di questo standard agli algoritmi di CMS. Questo lavoro è inserito all'interno del progetto di ricerca di Patatrack [4], un'attività che si sta occupando di valutare le diverse tecnologie per il calcolo eterogeneo

per l'esperimento CMS. In questo capitolo si trova un'introduzione a tale progetto, il procedimento che ha portato all'implementazione in SYCL del software del gruppo di ricerca e un'analisi dei risultati ottenuti.

La tesi si conclude con alcune considerazioni sullo standard SYCL e sulla sua implementazione utilizzata per questo progetto, alla luce dell'esperienza maturata durante questo lavoro; vengono, inoltre, suggerite alcune indicazioni per gli sviluppi futuri del progetto Patatrack.

Capitolo 1

Il CERN e gli acceleratori di particelle

1.1 Il CERN

Il CERN è sicuramente uno dei centri più importanti a livello mondiale per quanto riguarda la ricerca scientifica. Fin dalla sua nascita nei primi anni 50 del '900 riunisce ricercatori di tutto il mondo con l'obiettivo di studiare com'è fatto l'universo e quali sono le leggi che ne descrivono il funzionamento.

Al termine della Seconda Guerra Mondiale, la ricerca scientifica europea aveva bisogno di aggiornarsi. Fu così che nacque l'idea di un laboratorio europeo di fisica atomica. Questa proposta, avanzata da un gruppo di rinomati scienziati tra i quali Pierr Auger, Edoardo Amaldi e Niels Bohr, non aveva solo l'obiettivo di condividere i costi sempre crescenti della strumentazione necessaria alla ricerca, ma anche quello di unire e condividere le ricerche sulla fisica nucleare di tutta Europa. Fu così che nel Dicembre 1951, presso la conferenza internazionale dell'UNESCO di Parigi, nacque il *Conseil Européen pour la Recherche Nucléaire*, o CERN.

Con il passare del tempo, la conoscenza sulla struttura della materia è andata oltre il concetto di nucleo atomico; oggi l'area della fisica di cui si occupa il CERN è chiamata *fisica delle particelle* e spesso ci si riferisce a questo centro di ricerca come al Laboratorio

Europeo di Fisica delle Particelle [5].

Nonostante il principale campo di interesse del CERN sia la fisica, è importante evidenziare che la ricerca svolta in questi laboratori ha portato e porta tutt'oggi alla nascita di numerose tecnologie utilizzate anche in altri ambiti, primi fra tutti l'informatica e l'industria tecnologica. L'acquisizione e l'analisi dei dati prodotti dagli esperimenti richiedono infatti tecnologie all'avanguardia per ottenere il massimo delle prestazioni possibili.

Per citare alcune delle invenzioni più importanti nell'ambito informatico si considerino il Word Wide Web, inventato da Tim Berners-Lee nel 1989, e il Grid-computing, modello che descrive un'infrastruttura per il calcolo distribuito.

1.2 Gli acceleratori di particelle

I ricercatori del CERN hanno a disposizione un insieme di potenti macchine sulle quali effettuare esperimenti: gli *acceleratori di particelle*, capaci di accelerare fasci di protoni o elettroni a velocità prossime a quelle della luce. A queste velocità, l'energia cinetica delle particelle in collisione si trasforma in nuove particelle. Queste hanno una vita estremamente breve in quanto instabili e decadono molto velocemente in particelle più leggere che a loro volta si trasformano in altre particelle ancora più leggere. Questo fenomeno è regolato dalla nota equazione di Einstein $E = mc^2$ che descrive la relazione tra energia e massa. La legge afferma che la materia altro non è che una forma concentrata di energia e che, di conseguenza, le due grandezze sono interscambiabili.

Tramite lo studio delle collisioni e delle nuove particelle che si originano da esse, i fisici sono in grado di studiare il mondo dell'infinitamente piccolo; si ottengono importanti risultati che migliorano la nostra comprensione della materia e del funzionamento dell'universo. Infatti, le particelle più massive create dagli acceleratori di particelle esistevano soltanto nell'Universo primordiale.

1.2.1 Come funzionano gli acceleratori

In questa sezione si vuole descrivere, senza scendere troppo nei dettagli, il percorso che un fascio di protoni compie dalla sua immissione in un acceleratore sino al momento in cui viene fatto collidere.

Inizialmente si iniettano degli atomi di un gas, come ad esempio l'idrogeno, all'interno di una camera sorgente. Si applica poi alla camera un campo elettrico; in questo modo si riescono a strappare gli elettroni dagli atomi e ad isolare i protoni del gas. Successivamente, tramite campi elettromagnetici, si guidano le particelle rimaste nella camera sorgente all'interno dell'acceleratore.

Lungo i condotti si trovano cavità di radiofrequenza, camere metalliche che emettono onde radio ad una frequenza tale da accelerare le particelle in moto al loro interno. È importante sottolineare che nelle cavità nelle quali viaggiano le particelle non devono essere presenti altre molecole di gas, queste infatti potrebbero ostacolare il moto delle particelle. A questo scopo, i fasci di protoni si muovono all'interno di tubi metallici nei quali si mantiene il vuoto spinto.

Gli acceleratori si possono suddividere in due categorie in base alla loro forma. Gli acceleratori lineari sono formati esclusivamente da strutture acceleranti perché sono attraversati dalle particelle solo una volta; queste, pertanto, non devono essere deviate e possono mantenere la loro traiettoria lineare. Di conseguenza, per aumentare l'energia delle particelle è necessario aumentare la lunghezza della macchina. Negli acceleratori circolari, invece, i fasci di particelle percorrono più volte lo stesso circuito e ad ogni giro ricevono un'accelerazione. Per direzionare la traiettoria delle particelle si utilizzano potenti magneti; maggiore è l'energia dei protoni, maggiore dovrà essere l'intensità del campo magnetico che li direziona e li mantiene nell'orbita circolare.

Infine, una volta che le particelle hanno raggiunto una energia significativa, vengono fatte collidere o contro bersagli fissi o contro un altro fascio che giunge in direzione opposta. In questo secondo caso la macchina è chiamata collisore. In un acceleratore di

questo tipo, l'energia delle collisioni è maggiore perché è data dalla somma delle energie delle particelle che si scontrano.

In corrispondenza dei punti nei quali avvengono questi urti sono collocati dei rivelatori, potenti dispositivi in grado di individuare le particelle che si originano a seguito delle collisioni fra protoni[1]. Queste particelle divergono in tutte le direzioni e attraversano i diversi strati di sotto-rivelatori che compongono il dispositivo. Essi sono realizzati o per individuare il tipo specifico di particelle, o per misurare alcune loro caratteristiche fisiche quali velocità, massa, energia e quantità di moto[2].

I sotto-rivelatori possono essere dispositivi di tracciamento, calorimetri e dispositivi di identificazione [5].

- i dispositivi di tracciamento sono composti da un numero molto elevato di sensori che hanno lo scopo di rilevare le traiettorie delle particelle sfruttando gli effetti dell'interazione di una carica elettrica con alcuni materiali. Un sensore si attiva quando è attraversato da una particella carica elettricamente; questo comporta l'invio di un segnale ad un dispositivo che memorizza tale informazione. In un secondo momento un software analizza tutte le attivazioni rilevate e ricostruisce le traiettorie effettivamente percorse dalle particelle.

  Si consideri che all'interno dei rivelatori sono collocati dei potenti magneti. Questi influiscono sul moto delle cariche elettriche prodotte da una collisione deviando la loro traiettoria. Per questo motivo, tali cariche non hanno un moto lineare bensì curvilineo, in alcuni casi anche a spirale. La deviazione di traiettoria è molto im-

[1] I rivelatori non sono in grado di rilevare direttamente le particelle massive prodotte da una collisione: esse decadono troppo velocemente. Tali dispositivi, però, sono in grado di rilevare le particelle più leggere che si originano a seguito dei decadimenti.

[2] La quantità di moto di una particella è un vettore p definito come: $\vec{p} = m \cdot \vec{v}$ dove m è la massa della particella e $\vec{v}$ la sua velocità. Il vettore $\vec{p}$ ha la stessa direzione e lo stesso verso di $\vec{v}$ e si misura in $kg \cdot (m/s)$ [6].

portante per i fisici perché permette di risalire alla quantità di moto della particella, grandezza fisica che aiuta ad identificarla.

- i calorimetri hanno due obiettivi: arrestare il moto della maggior parte delle particelle che li attraversano e misurare la loro energia. In particolare, i calorimetri elettromagnetici catturano l'energia di elettroni e fotoni mentre i calorimetri adronici interagiscono con gli adroni, ovvero le particelle quali protoni e neutroni caratterizzate dalla presenza di quark. Entrambe queste tipologie di calorimetri, tuttavia, non sono in grado di bloccare il moto di muoni e neutrini.

- i dispositivi di identificazione utilizzano due diverse tecniche per risalire all'identità delle particelle; entrambe sfruttano l'emissione di radiazioni che si verifica quando le particelle attraversano un dato mezzo o quando attraversano il confine tra due isolanti elettrici con differente resistenza.

Grazie a tutte queste misure, i fisici riescono ad analizzare gli effetti di una collisione; studiando i dati ottenuti è possibile individuare la presenza di particelle insolite o osservare risultati inattesi che non aderiscono ai modelli attuali.

1.2.2 Le proprietà degli acceleratori

Le principali caratteristiche che identificano e descrivono un acceleratore di particelle sono:

- Il tipo di particella iniettata nell'acceleratore. Questa caratteristica dipende strettamente dallo scopo dell'esperimento. È fondamentale che la particella utilizzata abbia una carica elettrica; solo in questo modo, infatti, i campi elettromagnetici degli acceleratori riusciranno ad accelerarla. Le macchine del CERN accelerano protoni e nuclei di atomi ionizzati quali nuclei di piombo, argon e xeno.

- La luminosità istantanea descrive il numero di collisioni che avvengono nell'acceleratore per unità di superficie nell'unità di tempo, ed è espressa in $cm^{-2}s^{-1}$ o $fb^{-1}s^{-1}$. La luminosità integrata, invece, è misurata in fb^{-1} e descrive il numero di collisioni che si verificano in un dato intervallo di tempo.

  La luminosità è una grandezza fondamentale per la descrizione delle performance di un acceleratore: se si riescono ad osservare un numero molto elevato di collisioni, sarà più probabile individuare fra queste quelle di interesse per l'esperimento in questione.

- L'energia delle particelle è una grandezza fisica che si misura in *elettronvolt*; un elettronvolt (eV) è definito come l'energia guadagnata (o persa) da un elettrone (o da un protone) che si muove tra due punti posti nel vuoto tra i quali vi è una differenza di potenziale di 1 V [7].

  All'interno degli acceleratori, le cariche elementari acquisiscono un'energia nell'ordine di grandezza di 10 TeV. Questa quantità di energia, applicata al mondo dell'infinitamente piccolo, si avvicina a quella che esisteva subito dopo il Big Bang. Si consideri, inoltre, che la massa di un protone corrisponde a circa 1 GeV; pertanto, un protone accelerato ad un'energia di 6,5 TeV ha un'energia 6500 volte più grande della sua massa a riposo.

1.3 Il Large Hadron Collider

I fisici sono interessati ad analizzare collisioni che avvengono ad energie sempre più elevate; di conseguenza essi hanno bisogno di acceleratori via a via più grandi e potenti.

Ad oggi, il CERN gestisce un insieme di acceleratori circolari e lineari di diversa grandezza e potenza. Solo alcune di queste macchine comprendono dei rivelatori in grado di fornire dati agli esperimenti, le altre sono usate come iniettori, ovvero come acceleratori che sollecitano le particelle prima di passarle a macchine più grandi. Gli

acceleratori, infatti, sono collegati in sequenza con lo scopo di ottenere energie via via crescenti: una volta che una macchina ha accelerato alla velocità massima possibile le particelle che fluiscono al suo interno, esse vengono iniettate nella componente successiva.

Fra gli acceleratori del CERN si trova anche il *Large Hadron Collider*, o LHC, l'acceleratore più grande e potente al mondo avviato per la prima volta il 10 settembre 2008. LHC è un collisore circolare di 27 km di circonferenza che permette alle particelle di acquisire un'energia pari a 6,5 TeV cosicché le collisioni abbiano energia pari a 13 TeV.

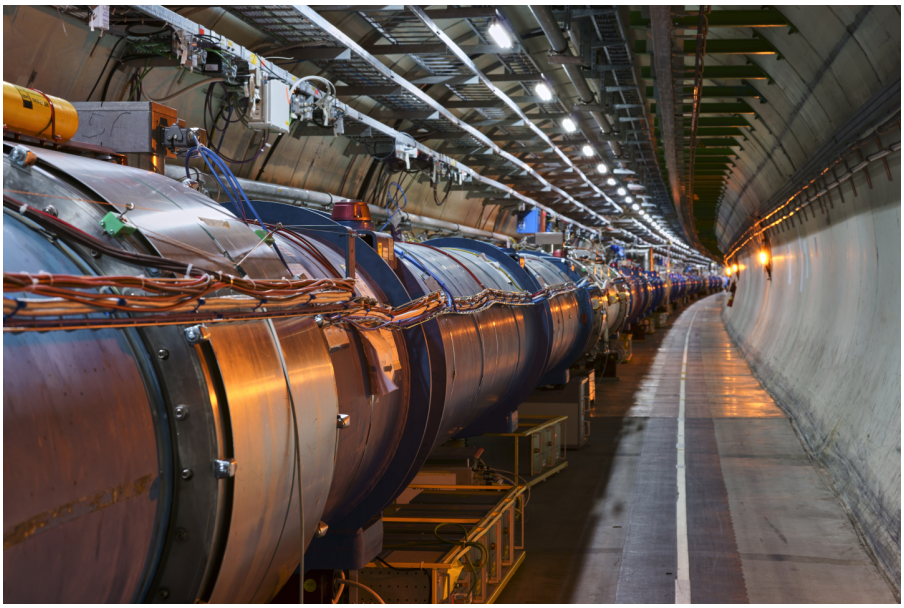


Figura 1.1: *Il Large Hadron Collider*

Fonte: `https://cds.cern.ch/images/CERN-PHOTO-201802-030-2`

1.3.1 Come funziona il Large Hadron Collider

Le particelle che fluiscono all'interno di LHC sono protoni estratti da atomi di idrogeno o ioni di piombo. Questi, prima di entrare nell'acceleratore principale, attraversano altri acceleratori come mostra la figura 1.2 [5].

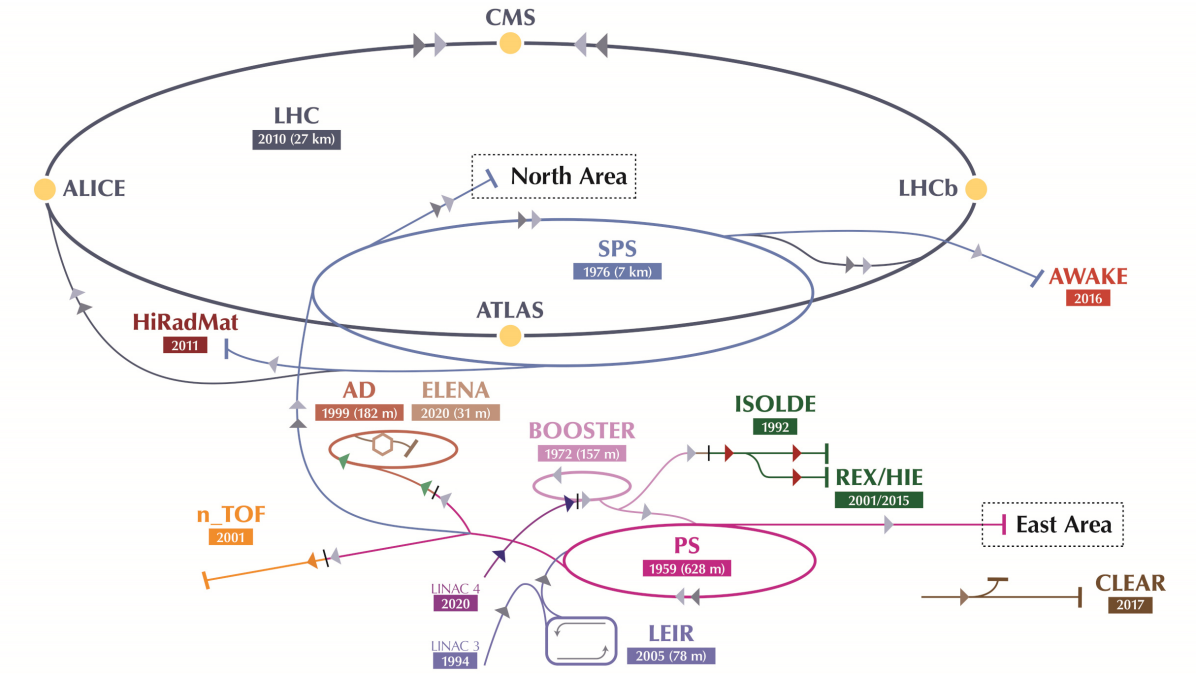


Figura 1.2: *La figura schematizza il complesso insieme di acceleratori e deceleratori del CERN. L'anello più grande rappresenta LHC; si noti che in giallo sono evidenziati i quattro rivelatori dell'acceleratore: ATLAS, CMS, ALICE e LHCb*

Fonte: *https://home.cern/science/accelerators/accelerator-complex/panoramas*

Le particelle entrano nel LHC suddivise in due fasci che circolano all'interno di due tubi distinti in direzione opposta. I protoni di ogni fascio sono raggruppati in pacchetti da 100 miliardi di particelle ciascuno. Occorrono all'incirca 20 minuti prima del raggiungimento dell'energia massima a cui corrisponde una velocità prossima a quella della luce. In ciascuna cavità è mantenuto il vuoto spinto per ridurre al minimo le collisioni tra i fasci e le particelle dell'aria.

Lungo tutto il percorso si trovano potenti elettromagneti che direzionano la traiettoria dei protoni. Questi elettromagneti sono composti da bobine di materiale superconduttore, mantenuti a una temperatura di pochi gradi Kelvin.

Numerosi altri potenti magneti sono utilizzati per direzionare e focalizzare i fasci di

protoni. Tra questi ci sono sia magneti dipolari lunghi 15 metri, sia magneti quadrupolari che raggiungono anche i 7 metri di lunghezza [5].

Intorno ai punti di collisione si trova un altro tipo di magnete che ha l'obiettivo di focalizzare il più possibile i fasci di particelle. In questo modo la probabilità di collisione aumenta. Si consideri che i protoni sono così piccoli che il compito di farli collidere in un punto preciso dello spazio è simile a sparare aghi a 10 chilometri di distanza l'uno dall'altro con una precisione tale da farli scontrare a metà strada.

Dal centro di controllo del CERN è possibile specificare dove far collidere i due fasci di protoni. Le collisioni possono avvenire in quattro punti posti in corrispondenza di quattro rivelatori: ATLAS, CMS, ALICE e LHCb [5].

ATLAS è uno dei due esperimenti general-purpose di LHC insieme a CMS. È composto da sei strati di sotto-rivelatori che registrano traiettoria, energia e quantità di moto delle particelle in modo da poterle identificare. Le sue dimensioni si attestano a 46 m di lunghezza e a 25 m di diametro per un peso di 7000 tonnellate (figura 1.3a).

CMS abbreviazione di *Compact Muon Solenoid*, è il secondo rivelatore general-purpose di LHC. CMS lavora in coppia con ATLAS al fine di indagare sugli stessi fenomeni con due approcci differenti. Questo rivelatore è costruito attorno ad un enorme magnete a solenoide, il più potente mai costruito, formato da una bobina cilindrica di materiale superconduttore capace di generare un campo magnetico di 3,8 tesla (6 ordini di grandezza più grande di quello terrestre); CMS è lungo 22 m, ha un diametro di 15 m e pesa 14000 tonnellate (figura 1.3b). Nel 2019 si attestava come una delle più grandi collaborazioni scientifiche mai realizzate: più di 5000 tra fisici, ingegneri, ricercatori e studenti provenienti da circa 50 Stati sono coinvolti in questo esperimento.

ALICE o *A Large Ion Collider Experiment*, è un rivelatore dedicato allo studio delle collisioni tra ioni pesanti e delle proprietà del*Quark Gluon Plasma*, uno stato della

materia con alta densità energetica che è esistito pochi momenti dopo il Big Bang[3]. ALICE pesa 10000 tonnellate, è lungo 26 m e largo 16 m (figura 1.3c).

LHCb ha l'obiettivo di investigare le differenze fra materia e antimateria tramite lo studio di una particella, il *beauty quark* (o b quark). Questo esperimento utilizza una serie di sotto-rivelatori che a differenza degli altri esperimenti non sono concentrici, bensì sono disposti a distanze crescenti dal punto di collisione fino a 20 m di distanza. LHCb è lungo 21 m per 10 m di altezza e 23 di larghezza e pesa circa 5600 tonnellate (figura 1.3d).

1.3.2 Triggering e acquisizione dei dati

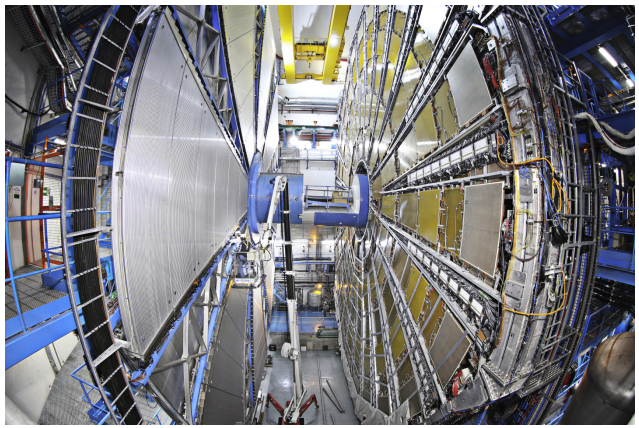
I due fasci di particelle di LHC vengono fatti scontrare all'interno dei rivelatori potenzialmente ogni 25 ns; ciascuno di questi *eventi* comporta che due pacchetti di particelle che circolano in verso opposto si intersechino, facendo sì che decine di particelle interagiscano, trasformando la loro energia cinetica e dando vita a fenomeni differenti. Il numero medio di collisioni che si verificano in un evento è noto come *pile up*.

Quando LHC viene utilizzato per far collidere protoni, questi eventi avvengono circa 30 milioni di volte al secondo, con un pile up medio di 50 collisioni; la figura 1.4 mostra uno di questi eventi ricostruiti dal rivelatore CMS.

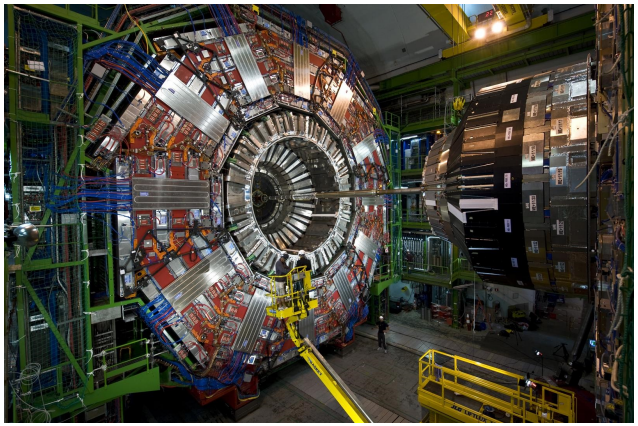
Si consideri, tuttavia, che la maggior parte delle collisioni non sono interessanti per i fisici[4]: essi infatti sono alla ricerca di interazioni che abbiano un'alta probabilità di generare fenomeni non ancora esplorati o nuovi. Generalmente per ogni evento si verifica una collisione interessante, questa però è sovrapposta ad una cinquantina di altre interazioni. È necessario pertanto un sistema detto *trigger* che filtri i dati in tempo

[3]Per maggiori approfondimenti si rimanda a [5] nella sezione "experiments".

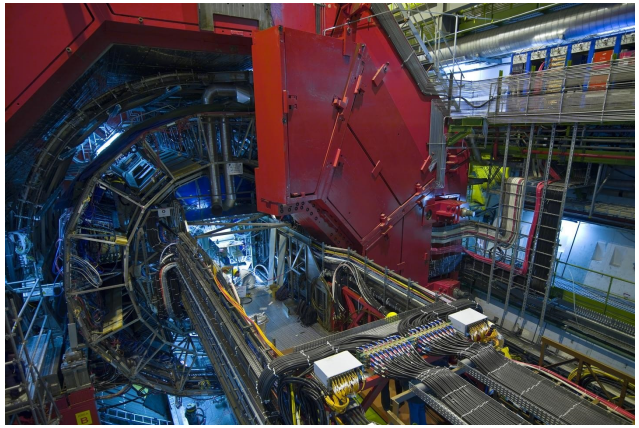
[4]Ad esempio, il bosone di Higgs, che è stato effettivamente osservato pochi anni fa da LHC, è prodotto in media una volta ogni 10 secondi.



(a) ATLAS



(b) CMS

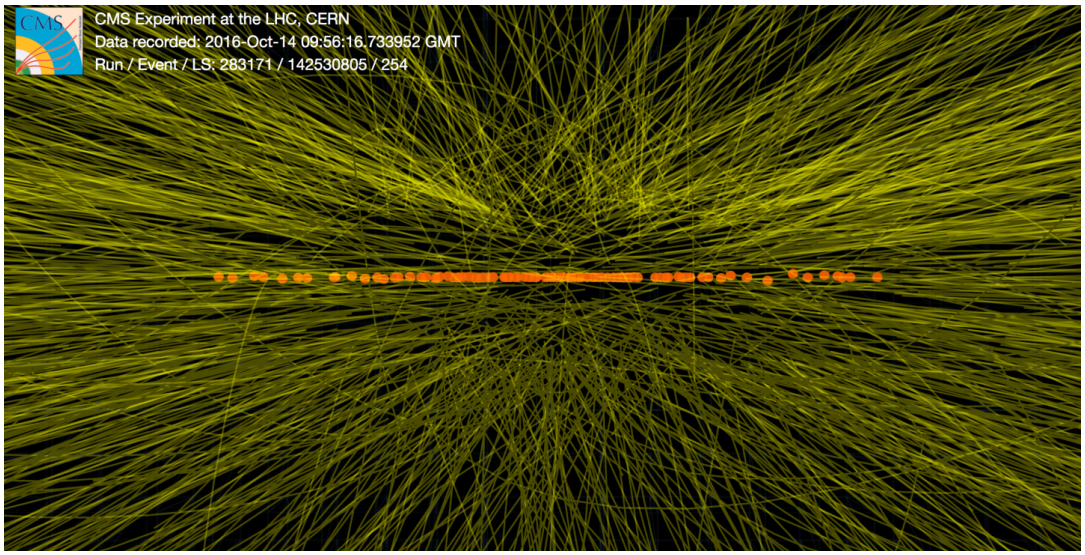


(c) ALICE

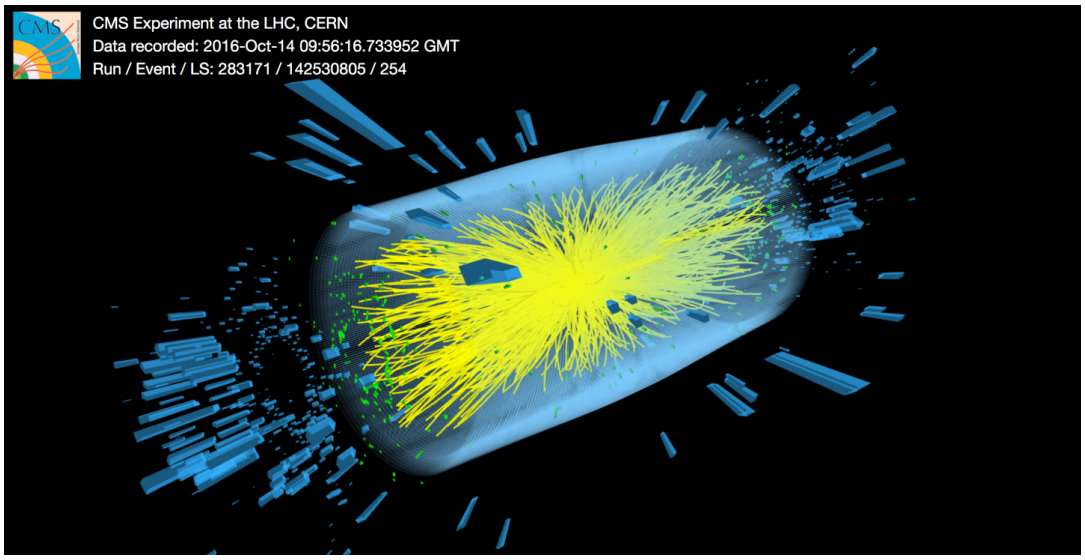
(d) LHCb

Figura 1.3: *I quattro rivelatori di LHC (fotografie e modelli): ATLAS* (a)*, CMS* (b)*, ALICE* (c) *e LHCb* (d)

Fonte: `https://home.cern/resources/image/experiments` *rispettivamente nelle sezioni* atlas-images-gallery, cms-images-gallery, alice-images-gallery *e* lhcb-images-gallery



(a)



(b)

Figura 1.4: *Un evento rilevato da CMS secondo due prospettive diverse. La figura 1.4a evidenzia i punti di collisione, la figura 1.4b sottolinea la moltitudine di particelle che si originano*

Fonte: `https://cds.cern.ch/record/2231915`

reale memorizzando esclusivamente quelli potenzialmente interessanti per la ricerca [8, 9]; visivamente lo scopo è quello di passare dal caos della figura 1.4 alla più ordinata e comprensibile figura 1.5.

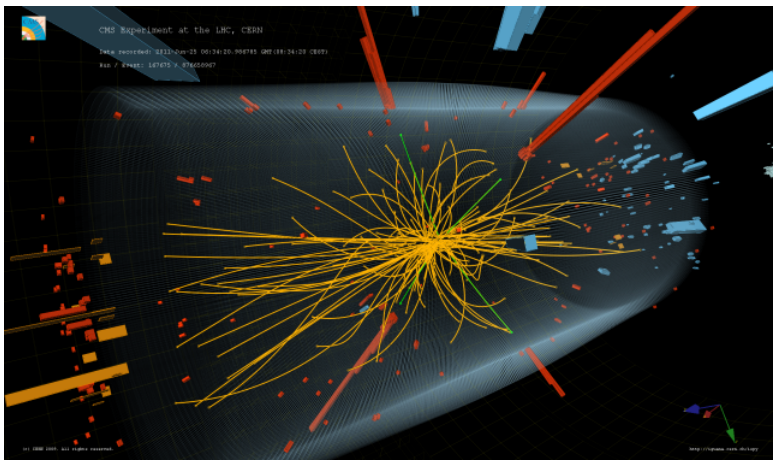


Figura 1.5: *Ricostruzione di una singola collisione tra protoni rilevata da CMS*

Fonte: `https://cds.cern.ch/record/1378102`

I sistemi di trigger di ATLAS e CMS ricevono eventi ad una frequenza di circa 30 MHz, e devono ricostruirne in tempo reale le proprietà fondamentali per poter selezionare in media 1 - 2 kHz di eventi. Tali sistemi sono organizzati in una pipeline a due livelli:

level-1 trigger il primo livello di trigger è implementato in hardware, utilizzando elettronica dedicata e algoritmi di ricostruzione programmati in FPGA; il suo compito è di sfruttare una lettura veloce, parziale, a bassa granularità per ridurre il numero di eventi dalla frequenza di interazione dell'acceleratore di circa 30 MHz alla frequenza massima di lettura del rivelatore di circa 100 kHz.

high-level trigger il secondo livello di trigger, o trigger di alto livello, è implementato in software e gira su delle farm di computer commerciali che comprendono circa 30000-40000 CPU cores; il suo compito è di sfruttare tutte le informazioni lette dal rivelatore per ricostruire in maniera più accurata e precisa le caratteristiche fisiche

degli eventi, in modo da poter individuare i più rari ed interessanti, e ridurre ad una media di 1-2 kHz il numero di eventi salvati offline per le successive ricostruzioni ed analisi.

1.3.3 Elaborazione offline e la Worldwide LHC Computing Grid

Anche dopo le selezioni applicate dai sistemi di trigger, LHC produce un'enorme quantità di dati: al massimo delle sue prestazioni genera più di cinque petabyte all'anno di informazioni. Elaborare questa mole di dati è particolarmente difficile, soprattutto è fattibile solo se numerosi computer collaborano tra loro. A questo scopo LHC utilizza un'infrastruttura distribuita di dispositivi sia per quanto riguarda la memorizzazione dei dati, sia per la loro rielaborazione; questa struttura è chiamata *Worldwide LHC Computing Grid* o WLCG ed ha una gerarchia a tre livelli chiamati *Tiers* come mostrato dalla figura 1.6.

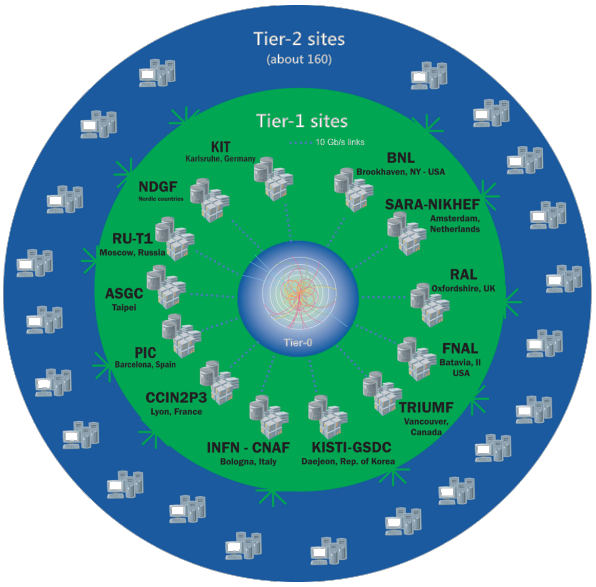


Figura 1.6: *Organizzazione a tre livelli, o "Tiers", della WLCG*

Fonte: `https://wlcg-public.web.cern.ch/tier-centres`

Al centro si trova il Tier-0, il data center situato al CERN. Questo livello mantiene una copia dei dati grezzi (o raw data), effettua una prima ricostruzione degli eventi e rielabora in modo più approfondito i dati nel periodo di inattività di LHC. Nonostante sia il livello più importante, il Tier-0 fornisce solo il 20% della capacità computazionale totale. I dati, infatti, sono distribuiti a tredici Tier-1, grandi centri di calcolo situati in diverse parti del mondo[5]. Ciascuno di questi centri è responsabile di mantenere una copia parziale delle informazioni; in questo modo è garantita una corretta ridondanza dei risultati ottenuti. Il secondo livello della gerarchia si occupa, inoltre, di elaborare i dati in modo più raffinato e di fornire tali informazioni ai Tier-2.

Il terzo livello della gerarchia è composto principalmente da università e da altri istituti scientifici. Essi di solito hanno una capacità computazionale sufficiente per effettuare ulteriori analisi specifiche sui dati. Esistono quasi 160 Tier-2 sparsi su tutto il globo; in questo modo fisici e studenti di tutto il mondo possono dare il loro contributo alla ricerca scientifica del CERN [5].

1.4 L'High-Luminosity LHC

Il Large Hadron Collider ha contribuito alla ricerca scientifica fino al 2018, anno in cui è stato spento per manutenzione e aggiornamenti. Per aumentare ulteriormente il suo potenziale di scoperte, infatti, l'acceleratore necessita di essere aggiornato; solo in questo modo si riusciranno a rilevare eventi rari che non erano stati osservati in precedenza. Con questo progetto i fisici avranno la possibilità di studiare più approfonditamente il *Modello Standard* della fisica delle particelle e avranno modo di approcciarsi ai problemi aperti riguardanti la materia oscura e il modello della supersimmetria [10].

[5]Uno di questi è ospitato a Bologna presso il centro INFN-CNAF, a cento metri dal Dipartimento di Informatica - Scienza e Ingegneria.

In linea teorica ci sarebbero diversi modi per ottenere un numero maggiore di collisioni interessanti. Ad esempio, si potrebbe aumentare l'energia delle particelle; questo però è difficile da realizzare nella pratica perché tale grandezza fisica è limitata dalla struttura dell'acceleratore. Quello che si è deciso di fare è di aumentare la luminosità di LHC di un fattore 10 rispetto a quello attuale. Poiché le modifiche che dovranno essere apportate all'acceleratore sono considerevoli, si è deciso di cambiare il suo nome in *High-Luminosity Large Hadron Collider*, in breve HL-LHC.

La prima fase del progetto è iniziata nel 2011 e ha visto la collaborazione non solo degli Stati membri del CERN, ma anche di Russia, Giappone e Stati Uniti. Dopo i primi cinque anni di progettazione e ricerca, i lavori di sviluppo, test e implementazione sono iniziati nel 2018 e richiederanno una decina di anni. Il primo periodo di attività di HL-LHC è chiamato *Run 4* ed è previsto a metà del 2027 come mostrato dal grafico in figura 1.7. In questa occasione il pile up passerà dalle attuali 50 collisioni per evento a 140, fino ad arrivare a 200 collisioni per evento durante il Run 5.

Diverse componenti dell'acceleratore sono in fase di aggiornamento o dovranno essere sostituite con altre componenti ancora più performanti. L'aumento della luminosità impone, infatti, l'utilizzo di magneti superconduttori all'avanguardia e nuove accortezze in termini di vuoto, raffreddamento e protezione delle macchine. Di conseguenza, il successo di High-Luminosity LHC dipende strettamente da una serie di tecnologie innovative che stanno richiedendo uno sforzo globale.

Anche i rivelatori sono coinvolti nel progetto: è previsto un aumento della granularità, della risoluzione e della frequenza di lettura dei rivelatori che produrrà un miglioramento della precisione delle misure. Se paragonassimo i rivelatori a delle macchine fotografiche che scattano fotografie tridimensionali alle collisioni fra particelle, esse dovranno scattare fotografie più rapidamente, ad una risoluzione più elevata e attribuendo una profondità maggiore ai colori di ciascun pixel.

Non solo le componenti meccaniche stanno richiedendo un imponente lavoro di ag-

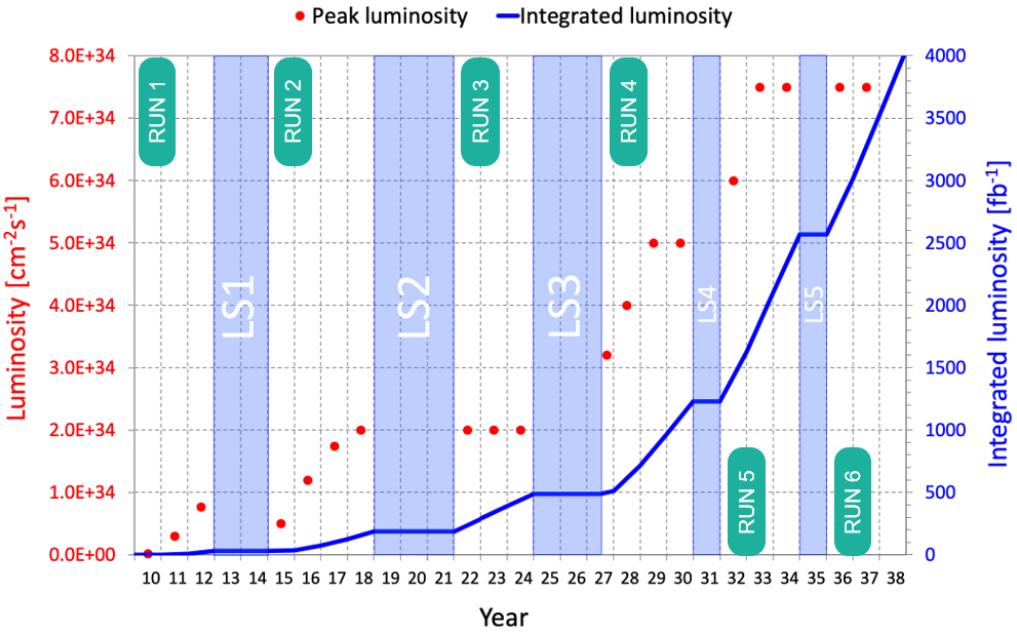


Figura 1.7: *Il programma di aggiornamento dell'High-Luminosity LHC. Si può notare che nel 2027, al termine del "Long Shutdown 3" (LS3), è prevista una fase di attività dell'acceleratore chiamata Run 4 nella quale la luminosità subirà un aumento*

Fonte: `https://lhc-commissioning.web.cern.ch/schedule/HL-LHC-plots.htm`

giornamento, anche il software necessità di essere rivisto alla luce di questo progetto. Per descrivere meglio quali sono le esigenze computazionali necessarie per il Run 4 di HL-LHC, si riportano in tabella 1.1 i parametri fisici più significativi che sono stati programmati[6].

Si consideri che la complessità computazionale degli algoritmi di ricostruzione di CMS cresce più che linearmente con l'aumentare del pile up. Questo comporta che il tempo e le risorse richieste per la ricostruzione degli eventi siano i parametri più importanti per determinare le necessità computazionali di HL-LHC.

Nel prossimo capitolo si tratterà più specificatamente di come l'esperimento CMS

[6]Per maggiori dettagli si rimanda a [11]

Tabella 1.1: *Parametri fisici più significativi programmati per HL-LHC*

	Run-3	**Run-4**
Energia di collisione (TeV)	14	14
Luminosità integrata (fb^{-1}/anno)	100	275
Pile up medio	55	140-200
Eventi memorizzati all'anno ($\times 10^9$)	9	56
Dimensioni di un evento in formato raw	1 MB	6,5 MB

stia cercando di adeguare il proprio software di ricostruzione e trigger per raccogliere ed analizzare i dati che saranno prodotti nei successivi Run di High-Luminosity LHC.

Capitolo 2

Il programma di aggiornamento dell'esperimento CMS

2.1 Il Compact Muon Solenoid e il suo software

2.1.1 Il rivelatore

Il Compact Muon Solenoid, o CMS, è uno dei due rivelatori *general-purpose* di LHC. In figura 2.1 sono evidenziati i diversi sotto-rivelatori concentrici che lo compongono.

Più internamente troviamo il *Silicon Tracker*, esso ha la funzione di identificare le traiettorie delle particelle che si originano a seguito di una collisione. Il Tracker è composto da circa 75 milioni di sensori di silicio disposti in quattordici strati concentrici. Quando una particella attraversa un sensore, essa interagisce elettromagneticamente con il silicio e produce un segnale, detto *hit*. L'elaborazione di tutti gli hit permette di ricostruire la traiettoria della particella.

Si consideri che la densità delle particelle nello spazio diminuisce via via che ci si allontana dal punto di interazione; per questo motivo, i primi layer del Silicon Tracker devono avere una elevata risoluzione per poter distinguere le particelle con precisione.

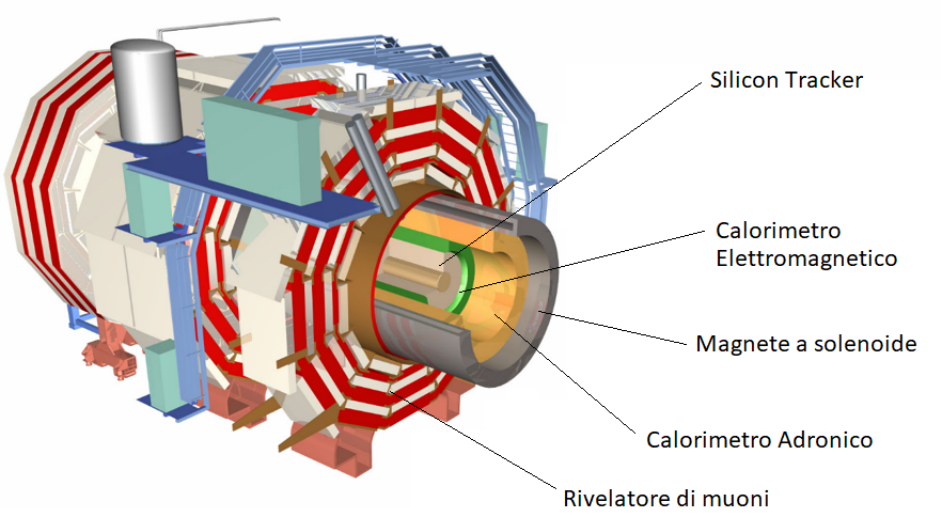


Figura 2.1: *La figura mostra i diversi rivelatori che nel loro insieme formano il Compact Muon Solenoid*

Fonte: `https://cds.cern.ch/record/1407706/plots`

I quattro strati più interni sono pertanto costituiti da sensori di silicio, chiamati *pixel*, che effettuano rivelazioni accurate in due dimensioni; questa precisione è supportata da un numero elevato di canali i quali producono una notevole quantità di dati per ogni evento. I successivi dieci strati del sotto-rivelatore sono attraversati da un numero minore di particelle e possono quindi avere una precisione inferiore. Essi si basano su sensori chiamati *strip* che raccolgono informazioni in una sola dimensione e che necessitano di meno canali.

Procedendo verso l'esterno troviamo il *Calorimetro Elettromagnetico*, o ECAL, e il *calorimetro Adronico*, o HCAL. Questi dispositivi hanno l'obiettivo di rilevare l'energia delle particelle che li attraversano; in particolare, il primo riesce a misurare l'energia di elettroni e fotoni, il secondo si occupa di misurare l'energia degli adroni, ad esempio neutroni e protoni. Per effettuare queste misure, i calorimetri convertono l'energia delle particelle che li attraversano in segnali elettrici mantenendo la proporzionalità tra energia rilasciata e segnale raccolto. Questo comporta il completo assorbimento delle particelle

che, di conseguenza, arrestano il loro moto.

Attorno ai calorimetri è posto il magnete solenoidale che dà il nome all'esperimento. Esso è composto da spire di una bobina superconduttrice che produce un campo magnetico uniforme di 3,8 tesla quando viene percorsa da corrente elettrica. Il suo compito è quello di deviare le particelle cariche prodotte dalle collisioni al fine di riuscire a misurarne la quantità di moto.

Lo strato più esterno è costituito dal rivelatore di muoni, particelle cariche che appartengono alla stessa famiglia degli elettroni, ma di massa 200 volte maggiore. I muoni sono particelle che riescono ad uscire dagli strati più interni del rivelatore senza essere arrestate; necessitano pertanto di un ulteriore sotto-rivelatore specifico che ne rilevi il passaggio [9].

2.1.2 Il software di CMS

CMS utilizza un unico software modulare, chiamato CMSSW, costituito da più di 4000 algoritmi scritti negli ultimi 15 anni grazie al contribuito di centinaia di studiosi e ricercatori provenienti da diversi ambiti. Visto nella sua interezza, è soprattutto un insieme di algoritmi di ricostruzione fisica in quanto si occupa di ricostruire le proprietà fisiche delle particelle che attraversano il rivelatore CMS.

Ad oggi CMSSW comprende più di 8 milioni di linee di codice ed è scritto principalmente in C++ (48%), ma contiene anche parti in python (29%), XML (15%) e Fortran(5%); l'intero sorgente è contenuto in un'unica repository Git [12] con licenza Apache 2.0.

Il software è costituito da diversi moduli e ciascuno di essi è composto a sua volta da alcuni algoritmi. Ogni modulo realizza una specifica funzionalità di CMSSW: può analizzare i dati prodotti da un singolo sotto-rivelatore o elaborare i risultati ottenuti dai moduli precedenti. Questa caratteristica garantisce al software tutti i vantaggi che derivano dalla programmazione modulare; tra questi si sottolinea l'indipendenza dei moduli,

che permette di sviluppare e testare ciascuno di essi singolarmente, e il riutilizzo degli stessi. I moduli possono, inoltre, essere eseguiti in parallelo permettendo la ricostruzione di più eventi contemporaneamente. Questa funzionalità è gestita da *Threading Building Blocks* (TBB) [13], una libreria C++ sviluppata da Intel; essa fornisce un insieme di algoritmi, uno scheduler e primitive di sincronizzazione di basso livello che consentono di gestire la memoria condivisa in un contesto parallelo.

CMSSW viene utilizzato per tutte le attività di simulazione e ricostruzione di CMS; due delle attività principali sono la ricostruzione online e quella offline. L'*online* software definisce le procedure di High-Level Trigger (o HLT) e di acquisizione dati; il suo scopo è quello di ricostruire gli eventi online, ovvero durante i periodi di attività dell'acceleratore, al fine di selezionare solo quelli interessanti. Il software *offline* si occupa, invece, dell'analisi dei dati una volta che questi sono stati selezionati e memorizzati nel file system distribuito del CERN o sulla grid. Se il software online è eseguito esclusivamente durante i Run di LHC, quello offline può essere eseguito ogni volta che è necessario, indipendentemente dall'attività dell'acceleratore.

Nonostante le diverse applicazioni, le ricostruzioni online ed offline sono molto simili. Ad esempio, l'algoritmo di ricostruzione delle tracce delle particelle è lo stesso, ma il software online si limita alle particelle con alta energia, quello offline è invece alla ricerca di fenomeni particolari. Anche se in alcuni casi gli algoritmi sono gli stessi, è importante sottolineare che il tempo a loro disposizione è diverso: come descritto nella sezione 1.3.2, HLT effettua un'elaborazione dei dati in modo quasi sincrono con la loro ricezione (il tempo di ricostruzione medio non può superare 300 ms), mentre il software offline non ha vincoli simili da rispettare e può effettuare una loro analisi in modo più accurato e preciso impiegando anche diversi secondi. Utilizzare un unico codice con configurazioni differenti a seconda del contesto comporta un importante vantaggio: il processo di ricerca di bug e di ottimizzazione che si fa su una delle due parti porta beneficio anche all'altra.

D'ora in avanti mi focalizzerò sul software online in quanto è quello su cui ho lavorato

in questo progetto di tesi. La figura 2.2 schematizza in un grafo a torta il tempo richiesto dai diversi moduli che compongono tale algoritmo:

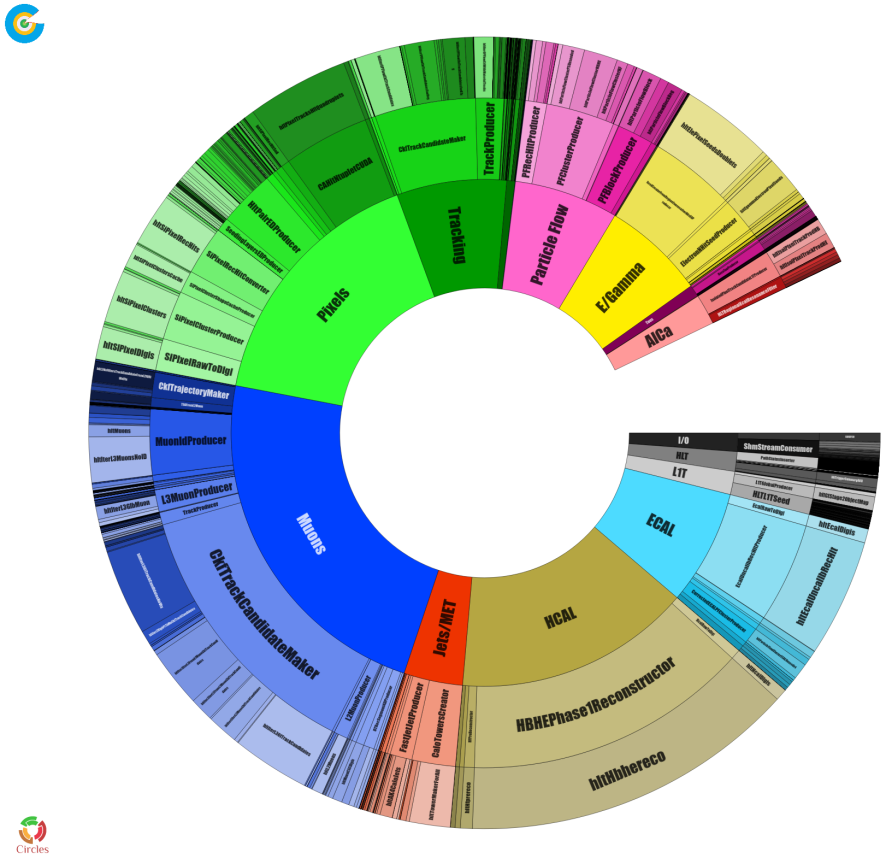


Figura 2.2: *Il grafico a torta mostra il tempo richiesto dai diversi moduli che compongono CMSSW online*

Fonte: *[14, slide 7]*

- I moduli in nero rappresentano la logica di input-output, ovvero si occupano della lettura dei raw data e dello storage in file oggetto compressi dei risultati ottenuti.
- Le sezioni ECAL e HCAL contengono i moduli che elaborano le informazioni rilevate dai calorimetri elettromagnetico e adronico. Più in particolare, essi si occupano di leggere i raw data prodotti dai sotto-rivelatori e di codificare queste informazio-

ni in un formato facilmente accessibile chiamato *digi*. A questo punto è possibile eseguire gli algoritmi di ricostruzione fisica locale che calibrano i dati e ricavano da essi l'energia rilasciata dalle particelle assorbite dai calorimetri.

- Pixel è la sezione che si occupa di elaborare le informazioni provenienti dagli omonimi sensori del Silicon Tracker. Anche in questo caso, i raw data sono inizialmente convertiti nel formato digi in modo tale da essere manipolati con più facilità. Successivamente, gli hit sono raggruppati in cluster, calibrati e passati agli algoritmi di ricostruzione fisica che determinano le traiettorie delle particelle fino a trovare il punto nel quale è avvenuta la collisione che le ha generate.

- I moduli di Tracking contengono gli algoritmi che elaborano i dati di tutti i layer del Silicon Tracker. Questa sezione è simile alla precedente perché anch'essa si occupa di ricostruire le traiettorie della particelle; a differenza dei Pixel, però, il "full tracking" è più lento, e quindi viene girato su una frazione più bassa degli eventi.

- La sezione in blu contiene i moduli per la ricostruzione dei muoni. Essi elaborano le informazioni prodotte dai rivelatori più esterni di CMS al fine di studiare i muoni prodotti dalle collisioni, e le combinano con quelle del Silicon Tracker per ricostruire le loro traiettorie.

- La sezione Particle Flow comprende i moduli per effettuare una ricostruzione ad alto livello. Essa combina i depositi di energia elaborati dai calorimetri con le tracce ricostruite da Pixel e Tracking. In questo modo è possibile individuare con più accuratezza le particelle prodotte in un evento.

- Jets/MET comprende algoritmi che effettuano una ricostruzione fisica ad alto livello. Per dare un'intuizione dello scopo di questa sezione, si consideri che essa elabora le informazioni in output dai moduli ECAL, HCAL e Particle Flow al fine

di determinare le energie delle particelle instabili prodotte da una collisione. Tali particelle decadono pochi istanti dopo la loro formazione producendo un "getto" di altre particelle più leggere, le uniche che sono effettivamente misurate dai rivelatori.

- Anche E/Gamma descrive algoritmi di ricostruzione di alto livello, ma utilizza i dati degli altri moduli al fine di analizzare esclusivamente il comportamento di elettroni e fotoni.

- La sezione più sottile adiacente a E/Gamma include i moduli per la ricostruzione di leptoni Tau, un tipo di particella elementare, utilizzando tecniche simili a quelle del Particle Flow.

- La sezione in arancione chiamata *AlCa* si occupa della calibrazione dei dati letti dai diversi sotto-rivelatori di CMS.

Si noti, infine, che dal grafo a torta della figura 2.2 manca una sezione. Per capire a cosa corrisponde, si consideri che i tempi dei diversi moduli sono stati calcolati avviando il cronometro all'inizio e alla fine degli algoritmi corrispondenti. Se si sommano tutti questi valori, però, si ottiene un tempo inferiore a quello misurato dall'intero programma. La sezione mancante corrisponde in larga parte al tempo richiesto per lo scheduling dei thread dei diversi algoritmi.

2.2 Il problema delle risorse di calcolo

Come già introdotto nella sezione 1.4, è evidente che il programma di ricerca del nuovo acceleratore deve essere supportato da un aggiornamento delle risorse di calcolo e dei software utilizzati per la raccolta e l'elaborazione dei dati. Solo con un adeguato modello computazionale, infatti, sarà possibile sfruttare appieno il potenziale di HL-LHC.

2.2.1 Le architetture

Il primo aspetto da sottolineare riguarda le architetture dei calcolatori disponibili nei centri di calcolo. Attualmente, la potenza computazionale dell'esperimento CMS è fornita prevalentemente da CPU; tuttavia, negli ultimi anni, il mercato si sta muovendo verso un utilizzo dei cosiddetti acceleratori, quali GPU e FPGA, grazie al fatto che, a parità di costo di acquisto e mantenimento, essi forniscono una potenza computazionale maggiore. L'aspettativa dei ricercatori è che in pochi anni la maggioranza dei supercomputer sarà caratterizzata dalla prevalenza di queste architetture. CMSSW, pertanto, per sfruttare al meglio l'opportunità di utilizzare i supercomputer, dovrà adeguarsi a questo cambiamento, individuando fin da subito gli acceleratori e le tecnologie più promettenti e adeguate ai suoi scopi [15].

È importante sottolineare che gli acceleratori non solo potranno migliorare le performance del software, ma consentiranno anche di avere più potenza computazionale a parità di costo e volume. È stato valutato, infatti, che per molti algoritmi sia più conveniente affidarsi alle GPU e aggiornare il codice di conseguenza piuttosto che aumentare la quantità di CPU; queste, infatti, richiederebbero molto più spazio dove poter essere collocate e molta più elettricità.

Si assume che, a parità di prezzo, la potenza computazionale delle sole CPU a cui CMS potrà accedere aumenterà del 20% ogni anno. Il grafico in nero della figura 2.3 mostra la proiezione di questa stima tra il 2018 e il 2030, anno nel quale si concluderanno le acquisizioni del Run 4. In blu, invece sono rappresentate le risorse necessarie per raggiungere gli obiettivi di HL-LHC. È evidente che un modello di calcolo basato esclusivamente sulle CPU non sia sufficiente per soddisfare la potenza computazionale richiesta dal nuovo acceleratore.

Attualmente in commercio si trovano GPU provenienti prevalentemente da quattro produttori; ciascuno di essi rilascia questi acceleratori con differenti librerie e strumenti

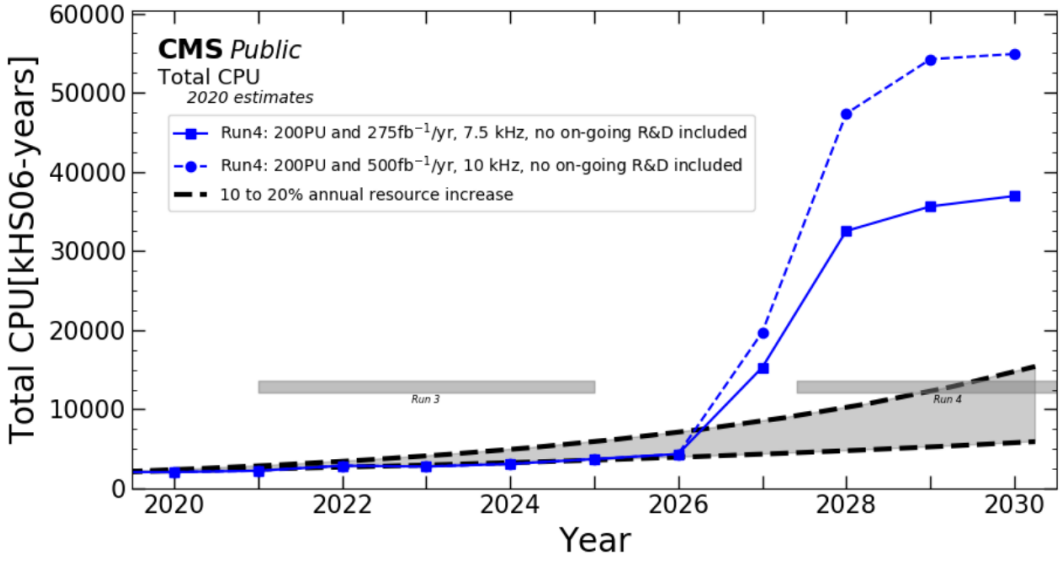


Figura 2.3: *Grafici delle stime della potenza computazionale fornita dalle CPU a disposizione di CMS (in nero) e della potenza computazionale necessaria per HL-LHC (in blu). La potenza computazionale è espressa in HEPSPEC, un benchmark che utilizza una serie di applicazioni comuni in ambito fisico per stressare il processore*

Fonte: *[16, slide 10]*

di programmazione: le GPU NVIDIA utilizzano l'ecosistema CUDA[1] [17], le GPU AMD si programmano con ROCm[2] [18], le GPU Intel XE utilizzano oneAPI[3] [19] e le GPU ARM Mali sfruttano OpenCL[4] [20]. Nonostante tutti questi strumenti siano diversi, essi

[1]CUDA (Compute Unified Device Architecture) è una piattaforma per il calcolo parallelo ed un modello di programmazione per le GPU. Il toolkit CUDA include librerie per le GPU NVIDIA, un runtime, un compilatore e diversi tool.

[2]ROCm è il software sviluppato da AMD per l'High Performance Computing e per la programmazione su GPU. L'ecosistema di ROCm utilizza e supporta diverse tecnologie open tra cui framework, librerie matematiche e modelli di programmazione.

[3]oneAPI è un modello di programmazione sviluppato da Intel per la programmazione parallela in ambiente eterogeneo. Verrà approfondito nei capitoli successivi.

[4]OpenCL è uno standard libero sviluppato da Khronos che fornisce delle API di basso livello basate su C e C++ per la programmazione di applicazioni parallele che possono essere eseguite su piattaforme diverse. I codici da girare sugli acceleratori devono essere contenuti in appositi sorgenti i quali, dopo un

condividono lo stesso modello di programmazione[5].

Per quanto riguarda le FPGA, i produttori più importanti sono: Altera (acquisita qualche tempo fa da Intel) e Xilinx (acquisita recentemente da AMD). I modelli di programmazione per questi acceleratori utilizzano Verilog[6] [21], OpenCL o oneAPI.

2.2.2 Il codice

Sorge quindi la necessità di un adattamento del codice: esso deve essere rivisto in modo da poter essere eseguito su un insieme di architetture eterogenee, come CPU, GPU e FPGA, provenienti potenzialmente da produttori diversi.

La sfida che ci si pone in questo contesto è piuttosto complessa: gli algoritmi implementati per CPU dal punto di vista del software design sono molto diversi da quelli che girano su GPU, e questi ultimi sono a loro volta diversi da quelli che girano su FPGA. Si pensi, ad esempio, che le applicazioni scritte per GPU generalmente sfruttano massivamente il parallelismo sui dati; questo è invece difficile da ottenere sulle CPU con buoni risultati di performance. Inoltre, lo spostamento di dati dalla CPU alla GPU è un'operazione molto costosa dal punto di vista delle prestazioni e può facilmente diventare il "collo di bottiglia" dell'intero software. Questo problema è molto meno evidente in applicazioni che girano esclusivamente su CPU. Quello che si sta cercando è quindi una tecnologia che oltre a permettere di scrivere un unico programma che giri su diverse architetture, consenta anche di ottenere delle buone performance su ciascuna di esse.

---

opportuno processo di compilazione, sono integrati alla restante applicazione.

[5]Le GPU, o *graphics processing unit*, sono particolari tipi di coprocessori specializzati nella grafica computerizzata; negli ultimi anni stanno avendo un grande successo anche nella programmazione parallela generica, nota come GPGPU. In questo secondo contesto, il modello di programmazione più diffuso prevede la presenza di un host il quale demanda alla GPU l'esecuzione di specifiche routine, chiamate kernel. Questi sono eseguiti in parallelo sulle unità computazionali della GPU seguendo un approccio SIMD (single instruction, multiple data).

[6]Verilog è un linguaggio standard di descrizione hardware usato per programmare circuiti elettronici.

Alcuni primi passi in questa direzione sono già stati fatti per il Run 3 di LHC: una parte del codice dell'High-Level Triggering è stata portata in CUDA per essere eseguita su GPU NVIDIA. I risultati ottenuti sono promettenti ma il test è molto specifico: il codice implementato gira su un unico tipo di acceleratore, le GPU, e per di più di un unico produttore, la NVIDIA. Si rendono necessari la ricerca e lo sviluppo di possibili soluzioni che superino questa limitazione. Tali tecnologie saranno valutate in termini di portabilità di prestazioni su dispositivi eterogenei al fine di individuare quale tra queste sia la più adatta alle esigenze di CMSSW. Si sono individuate di conseguenza le quattro più promettenti:

Alpaka è una libreria C++ che fornisce un layer di astrazione per lo sviluppo di codice da eseguire su acceleratori eterogenei. Nasce nel 2015 presso l'Università di Dresda come libreria per esperimenti di fisica a bassa energia; negli anni si è evoluta come prodotto a disposizione della comunità scientifica ed è attualmente mantenuta dall'istituto CASUS del centro di ricerca Helmholtz-Zentrum Dresden-Rossendorf [22]. Il codice del progetto è open-source ed è disponibile su GitHub [23].

Alpaka è una libreria indipendente da qualsiasi piattaforma hardware e supporta un vasto numero di back-end tra i quali CUDA [17], HIP[7] [24], TBB [13], OpenMP[8] [25]. Il suo modello di esecuzione prevede l'utilizzo della CPU come host, mentre i kernel sono eseguiti in parallelo sugli acceleratori o su CPU. Il codice che gira su questi dispositivi può essere scritto indipendentemente dall'hardware sottostante; in questo modo è possibile rimandare anche a runtime la scelta dell'architettura sul quale eseguirlo.

[7]HIP, o Heterogeneous-Computing Interface for Portability, è un dialetto di C++ dell'ecosistema ROCm progettato per facilitare la conversione in C++ delle applicazioni CUDA al fine di renderle portabili su GPU NVIDIA e AMD.

[8]OpenMP è una API multi-piattaforma per la creazione di applicazioni parallele su sistemi a memoria condivisa.

Per facilitare gli sviluppatori che provengono da un ambiente CUDA, è stata realizzata un'interfaccia utente in C++ chiamata CUPLA [26]. Essa offre ai programmatori un ambiente Alpaka che si basa su griglie, blocchi e thread, del tutto simile a quello fornito dalla NVIDIA.

Alpaka è una tecnologia molto promettente, sia per le prestazioni che permette di ottenere, sia perché l'istituto di ricerca che sviluppa questa nuova tecnologia ha dimostrato interesse nel sostenere i progetti di CMS.

Kokkos è un modello di programmazione sviluppato dai Sandia National Laboratories[9] [27] che, come Alpaka, definisce astrazioni per la scrittura di software parallelo su acceleratori eterogenei in contesti di High Performance Computing (o HPC) [28]. L'ecosistema di Kokkos comprende, in aggiunta, primitive per la gestione dei dati, kernel matematici e strumenti di debug ed è disponibile con licenza libera su GitHub [29].

Attualmente Kokkos supporta NVIDIA CUDA, AMD ROCm e OpenMP, tuttavia gli sviluppatori sono impegnati nella realizzazione di estensioni per altri back-end. Anche questo modello di programmazione utilizza una sintassi simile a quella di CUDA e supporta TBB.

SYCL è uno standard proposto da Khronos Group[10] [30] per lo sviluppo di codice parallelo in contesti eterogenei [2]. È un modello di programmazione astratto e di alto livello che nasce sopra al back-end OpenCL con l'obiettivo di permettere agli sviluppatori di scrivere in un unico sorgente il codice da eseguire su un insieme eterogeneo di dispositivi.

[9] I Sandia National Laboratories sono due importanti laboratori del Dipartimento dell'Energia statunitense (DOE). Essi si occupano dello sviluppo di componenti per le armi nucleari e di questioni di sicurezza nazionale per la National Nuclear Security Administration (NNSA).

[10] Khronos group è un consorzio di oltre 150 aziende leader nel settore ICT che si occupa di sviluppare e mantenere standard libere da royalty per un'ampia varietà di piattaforme ed ecosistemi.

SYCL è conforme allo standard C++ in quanto ne mantiene sintassi e semantica. Uno degli obiettivi di SYCL è quello di indirizzare codice su un numero sempre crescente di dispositivi diversi. Questo impone ai programmatori alcuni limiti all'utilizzo delle funzionalità di C++ nelle routine da eseguire sugli acceleratori, tra queste: puntatori a funzione, eccezioni, funzioni ricorsive e chiamate a funzioni virtuali.

Nello sviluppo di SYCL, Khronos group collabora con la comunità di C++: nelle linee guida per le future implementazioni di questo standard, infatti, è riportata la necessità di estendere C++ affinché supporti la programmazione su dispositivi eterogenei [31].

Attualmente esistono quattro implementazioni di SYCL: ComputeCPP, hipSYCL, triSYCL e DPC++; ciascuna di queste sarà approfondita nella sezione 3.2. Dal momento che questo standard è oggetto di tesi, in questo contesto non sarà approfondito ulteriormente; una sua descrizione più accurata si trova nel capitolo 3.

oneAPI è un nuovo modello di programmazione proposto da Intel che si pone l'obiettivo di unificare lo sviluppo di codice indirizzato ad architetture eterogenee [19]. Questa tecnologia implementa lo standard SYCL e lo estende con nuove caratteristiche e funzionalità. L'ecosistema oneAPI comprende il linguaggio di programmazione Data Parallel C++ (DPC++), diversi strumenti, librerie, ed un'interfaccia di basso livello per l'esecuzione di codice sui dispositivi Intel. Nel progetto che ha portato alla stesura di questo elaborato è stato approfondito e utilizzato oneAPI; una sua descrizione più dettagliata è riportata nei capitoli successivi.

2.2.3 Il caso di studio del gruppo Patatrack

Patatrack è un'attività del CERN alla quale partecipano persone di diversa provenienza che dal 2016 si occupano di ricerca e sviluppo per CMS; il suo principale obiettivo

è quello di dimostrare che la ricostruzione online su ambiente eterogeneo sia possibile e vantaggiosa, sia per quanto riguarda il software, sia dal punto di vista delle infrastrutture.

Per prima cosa, il gruppo si è occupato di progettare e sviluppare algoritmi paralleli ed eterogenei per la ricostruzione dei calorimetri e del Pixel Detector; i risultati ottenuti sono apparsi fin da subito promettenti (si veda il capitolo 4 per maggiori dettagli). Successivamente, è stata creata una versione semplificata del software online con l'obiettivo di avere un codice apposito per il testing delle quattro tecnologie riportate nella sezione precedente, prima di tutto per verificare la loro effettiva applicabilità a CMSSW, ma anche per analizzare e confrontare le performance dei software riscritti con il loro utilizzo.

È proprio in questo contesto che si è collocato il progetto di questa tesi. In particolare, il lavoro che mi ha coinvolto ha previsto, inizialmente, lo studio di SYCL, una delle quattro tecnologie in fase di sperimentazione; una sintesi delle specifiche di questo modello di programmazione si trova nel capitolo 3. In un secondo momento, sono stata inserita in un gruppo di lavoro con l'obiettivo di convertire la versione semplificata di CMSSW da CUDA a una delle quattro implementazioni di SYCL, DPC++. Questo secondo lavoro è descritto dettagliatamente nel capitolo 4.

Capitolo 3

Il modello di programmazione SYCL

3.1 Lo standard

SYCL è uno standard per la programmazione di sistemi eterogenei libero da royalty, proposto e mantenuto da Khronos Group, un consorzio no-profit di oltre 150 aziende del settore ICT che si occupa di realizzare diversi standard per un'ampia varietà di piattaforme ed ecosistemi [30].

SYCL è un modello di programmazione astratto che aggiunge semplicità e flessibilità d'uso ai sottostanti concetti di portabilità ed efficienza di OpenCL: definisce una libreria di alto livello che astrae le interfacce di back-end[1] dei singoli dispositivi permettendo ai programmatori di scrivere in un unico sorgente conforme a C++ sia il codice da eseguire sull'host, la macchina su cui gira l'applicazione, sia il codice da eseguire su dispositivi eterogenei; questo è chiamato *SYCL kernel function*, o solamente kernel, ed è definito all'interno di un *function object*[2]. Rispetto al modello di compilazione usuale,

[1] Le interfacce di back-end si occupano di implementare la comunicazione tra il runtime e i dispositivi fisici per l'effettiva esecuzione delle istruzioni sulle specifiche architetture hardware.

[2] In C++, un function object è un oggetto di una classe che definisce l'overload dell'operatore "chiamata di funzione" `()`. Un oggetto di quella classe può essere invocato come se fosse una funzione.

applicazioni di questo tipo richiedono un processo di compilazione più sofisticato, il quale sarà approfondito nelle sezioni successive.

SYCL è stato pensato per essere conforme all'ISO C++. In particolare, finché non viene richiesta integrazione con OpenCL, un compilatore per C++ è in grado di compilare un'applicazione SYCL producendo un eseguibile che giri correttamente sulla CPU. Per garantire la portabilità su un numero ampio di dispositivi, è necessario introdurre alcune limitazioni alle funzionalità dell'ISO; ad esempio nei kernel non si possono utilizzare puntatori a funzione, eccezioni e chiamate a funzioni virtuali. A tal proposito, Khronos collabora con il comitato di standardizzazione di C++ con l'obiettivo di estendere il linguaggio affinché le sue future specifiche integrino la programmazione su dispositivi eterogenei [31].

Il codice 3.1 riportato di seguito è un esempio che mostra le principali sezioni di un'applicazione SYCL[3]. Per prima cosa si noti che nella prima riga è incluso l'header file `CL/sycl.hpp`; questo consente l'utilizzo delle funzionalità definite dallo standard. Tutti i nomi introdotti dalla libreria sono inclusi nel namespace `cl::sycl` (riga 4). L'applicazione è poi strutturata in tre scope annidati:

kernel scope è il blocco più interno (righe 23-25) e contiene il kernel che deve essere compilato ed eseguito dall'acceleratore. Questo codice richiede un function object che generalmente viene creato con una *lambda expression*[4]. È importante sottolineare che i kernel devono essere definiti esclusivamente all'interno di un *command group* (si veda il prossimo punto dell'elenco) e che ciascun command group può contenere al più un kernel.

command group scope è il blocco intermedio (righe 17-26); esso definisce un'unità di lavoro che comprende la definizione delle strutture dati da copiare sugli acceleratori

[3]Il codice 3.1 si trova in [32, pagina 19].

[4]Una lambda expression è un'espressione C++ che produce un function object anonimo.

e il codice da eseguire in parallelo. Sintatticamente, un command group è un function object.

application scope è il blocco più esterno che include gli altri due (righe 10-27). Specifica tutto il codice che non è incluso in un command group; in particolare, in questa sezione sono definite le *queue*, strutture che permettono di indicare su quale dispositivo eseguire i kernel, e i buffer, le strutture dati che contengono le informazioni che dall'host dovranno essere trasferite al dispositivo per essere manipolate. L'esecuzione dei distruttori degli oggetti creati all'interno di questo scope permette il completamento di tutte le operazioni sul dispositivo.

```
1  #include <CL/sycl.hpp>
2
3  int main() {
4    using namespace cl::sycl;
5    int data[1024]; // Allocate data to be worked on
6
7    // By sticking all the SYCL work in a {} block, we ensure
8    // all SYCL tasks must complete before exiting the block,
9    // because the destructor of resultBuf will wait.
10   {
11     queue myQueue; // Create a default queue to enqueue work to
12
13     // Wrap our data variable in a buffer
14     buffer<int, 1> resultBuf { data, range<1> { 1024 } };
15
16     // Create a command_group to issue commands to the queue
17     myQueue.submit([&](handler& cgh) {
18       // request access to the buffer
19       auto writeResult =
20         resultBuf.get_access<access::mode::discard_write>(cgh);
```

```
21        // Enqueue a parallel_for task
22        cgh.parallel_for<class simple_test>(range<1> { 1024 },
23            [=](id<1> idx) {
24          writeResult[idx] = idx[0];
25        }); // End of the kernel function
26      }); // End of our commands for this queue
27    } // End of scope, we wait for work producing resultBuf to complete
28  }
```

Codice 3.1: *Codice di esempio di un programma SYCL*

Nelle prossime sezioni si trova una spiegazione più dettagliata delle caratteristiche sulle quali si fonda il nuovo standard del gruppo Khronos. In particolare, nella sezione 3.1.1 si approfondisce il modello di compilazione delle applicazioni SYCL; la loro topologia è descritta nella sezione 3.1.2 insieme al modo per selezionare i dispositivi sui quali eseguire i kernel. In seguito si definiscono il flusso di esecuzione delle applicazioni (sezione 3.1.3) e la gestione della memoria (sezione 3.1.4). La trattazione prosegue nella sezione 3.1.5 con l'analisi di alcune strutture fondamentali per la programmazione. Infine, nella sezione 3.2 è riportata una descrizione per ognuna delle quattro implementazioni di SYCL più note. È importante sottolineare, infatti, che queste specifiche altro non sono che la descrizione di uno standard; di conseguenza, la scelta di come realizzarle concretamente è lasciata libera agli sviluppatori delle sue implementazioni.

3.1.1 Il modello di compilazione

Per quanto riguarda la compilazione, SYCL utilizza l'approccio *single-source multiple compiler-passes*, o SMCP: l'applicazione passa attraverso diversi compilatori che producono o una rappresentazione intermedia (IR) in codice binario, o un insieme di istruzioni (ISA) specifiche per il dispositivo. I risultati di questi processi sono successivamente uniti in un unico eseguibile finale.

Ciascuna implementazione di SYCL traduce questo modello di compilazione in un work-flow che dipende dai back-end supportati. Generalmente, i kernel sono compilati dal compilatore OpenCL (o di un altro back-end) dell'acceleratore che li dovrà eseguire, mentre il resto dell'applicazione è compilato da un compilatore standard di C++.

Per spiegare più dettagliatamente i passaggi del processo di compilazione, si consideri l'esempio di ComputeCpp [33] (maggiori dettagli sul questa implementazione si trovano nella sezione 3.2). Il back-end utilizzato da questa implementazione è OpenCL, il quale supporta differenti tipi di compilazione in modo da poter essere portabile su dispositivi diversi: la principale è SPIR/SPIR-V[5] utilizzata dai dispositivi Intel e dalle GPU AMD, ma si stanno studiando anche altre compilazioni che utilizzino GCN[6] e PTX[7].

Generalmente è possibile scegliere fra i due diversi processi di compilazione mostrati in figura 3.1:

just-in-time compilation Compilazione non specifica per un'architettura hardware a carico del SYCL device compiler: esso compila separatamente sia i kernel SYCL, sia l'applicazione C++ nella quale sono definiti; successivamente unisce i risultati ottenuti in un file leggero e portabile che, però, dovrà essere ricompilato a runtime dal compilatore del dispositivo (figura 3.1b). Se da un lato questo processo permette di rimandare a runtime la scelta dell'effettivo acceleratore sul quale eseguire i kernel, dall'altro occorre rimpiazzare il compilatore dell'intera applicazione con il

[5]SPIR, o Standard Portable Intermediate Representation, è un linguaggio open-source sviluppato da Khronos Group per la computazione parallela su dispositivi OpenCL. SPIR in particolare è un linguaggio intermedio utilizzato in fase di compilazione per creare file binari indipendenti dai dispositivi sui quali saranno eseguiti.

[6]GCN, o Graphics Core Next, è il nome di una microarchitettura e di un set di istruzioni per essa sviluppata da AMD per programmare le sue GPU.

[7]PTX, o Parallel Thread Execution, è un linguaggio utilizzato da NVIDIA per la programmazione in CUDA. I compilatori nvcc responsabili della compilazione di codice CUDA producono un file intermedio scritto in PTX; questo passa poi ai compilatori dei driver delle schede grafiche che traducono il codice PTX in codice binario.

SYCL device compiler e questo può essere complesso se comporta la modifica di una toolchain preesistente.

ahead-of-time compilation I kernel sono compilati manualmente indicando esplicitamente il dispositivo hardware incaricato della loro esecuzione. In questo modo, il compilatore del dispositivo può effettuare una compilazione più spinta riducendo notevolmente il tempo impiegato dal runtime. La restante applicazione è compilata con il compilatore dell'host (che può essere un qualunque compilatore per C++) il quale integra nel suo processo l'header file ottenuto precedentemente (figura 3.1a). L'utilizzo di due compilatori separati favorisce l'integrazione di SYCL in toolchain già esistenti.

3.1.2 La topologia di un'applicazione SYCL

Il modello per la topologia definito dallo standard prevede un host e uno o più acceleratori. L'applicazione SYCL è eseguita inizialmente sull'host poi, tramite i command group, si inviano i kernel o ai dispositivi OpenCL, o ai dispositivi con altri back-end, o alla CPU in modalità SYCL host device.

Gli acceleratori a cui inviare i kernel sono determinati a run-time attraverso una *topology discovery*. Questo procedimento restituisce l'insieme dei back-end disponibili (ad esempio Intel OpenCL, Intel Level Zero o Nvidia OpenCL) e per ognuno di questi sono indicati tutti i dispositivi a cui si può accedere.

In SYCL esistono due modi per conoscere la topologia di un sistema [34][35]. Con la modalità manuale il programmatore ha a disposizione delle API per interrogare esplicitamente le piattaforme e i dispositivi al fine di trovare quali tra questi sono disponibili. A questo scopo SYCL definisce le classi `platform` e `device` con un insieme di metodi e campi per il loro utilizzo. In particolare, nel codice 3.2 sono riportati gli esempi delle funzioni `get_platforms()` e `get_devices()`.

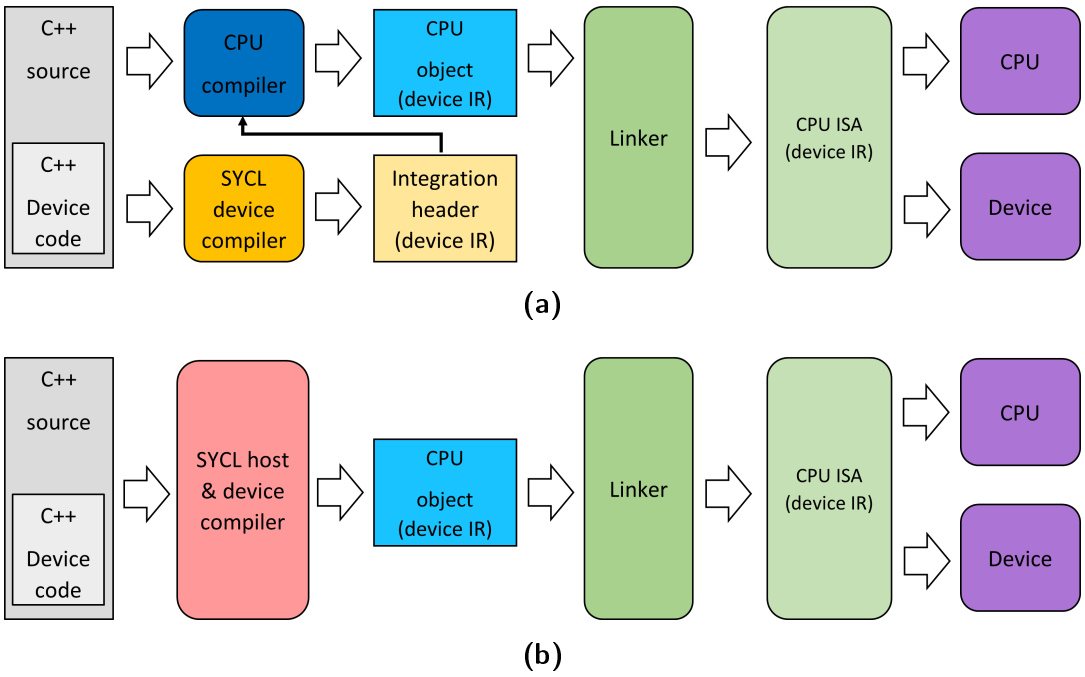


Figura 3.1: *La prima immagine rappresenta il work-flow del processo di compilazione di un'applicazione ComputeCpp se si usano sia il device-compiler che un generico compilatore C++; la seconda immagine rappresenta lo stesso work-flow se si utilizza il solo device-compiler*

Fonte: *IWOCL/SYCLcon SYCL Tutorial* `https://www.youtube.com/watch?v=tbHa7bxN5qo&t=5585s`, *min 6-7*

```
1 auto platforms = sycl::platform::get_platforms();
2 auto devices = sycl::device::get_devices();
```

Codice 3.2: *Esempio di utilizzo delle funzioni per interrogare manualmente la topologia del sistema*

Lo standard prevede anche una modalità automatica di topology discovery. Questa utilizza un function object chiamato *device selector* che restituisce i dispositivi associandoli ad un valore che li classifica per efficienza e disponibilità (i valori negativi indicano che il dispositivo non è disponibile). Si possono utilizzare sia device selector specifici per

un tipo di piattaforma (codice 3.3, righe 1-2), sia function object che analizzano tutte le piattaforme possibili (codice 3.3, righe 4-5).

```
1 gpu_selector gpuSelector{};
2 auto gpuDevice = gpuSelector.select_device();
3
4 default_selector defSelector{};
5 auto chosenDevice = defSelector.select_device();
```

Codice 3.3: *Esempi di device-selector automatici; il primo è relativo alle sole GPU, il secondo considera tutti i tipi di piattaforme*

Nonostante SYCL implementi diversi device selector, lo standard fornisce ai programmatori la possibilità di crearne di personalizzati. Questi devono implementare la funzione `operator()` che associa ad ogni possibile dispositivo un valore di preferenza attribuito dal programmatore in base alle proprie esigenze. Per ottenere informazioni su dispositivi e piattaforme si può utilizzare la funzione `get_info<T>` il cui parametro di template specifica il tipo dell'informazione che si vuole ottenere.

Per inviare lavoro ad un dispositivo, SYCL utilizza un oggetto di tipo `queue`. Al momento della creazione, tale oggetto è associato ad un acceleratore specifico tramite l'utilizzo di un opportuno device selector. Le code hanno lo scopo di gestire l'ordine di esecuzione dei command group e dei kernel definiti al loro interno. Questi ultimi, infatti, possono essere visti come task asincroni rispetto al codice che gira sull'host, e possono essere eseguiti anche in un ordine diverso rispetto a quello nel quale sono definiti, l'importante è che siano disponibili tutte le informazioni necessarie alla loro esecuzione (si veda la sezione successiva).

3.1.3 Il modello di esecuzione

L'esecuzione di un programma SYCL può essere suddivisa in due parti: da un lato i kernel SYCL, dall'altro l'applicazione eseguita dall'host.

Modello di esecuzione dell'applicazione

L'applicazione generalmente segue il modello di esecuzione dei codici C++ che girano su CPU. Il suo scopo è quello di gestire l'avvio dei kernel verificando che tutti i requisiti necessari alla loro esecuzione, definiti nei command group, siano soddisfatti.

I command group si presentano come lambda expression e sono presi in input dal metodo `submit()` delle code. Al momento della loro creazione, pertanto, ciascuno di essi è associato al dispositivo sul quale il kernel definito al suo interno dovrà essere eseguito. L'esecuzione di un command group comporta l'avvio di un task che si occupa di catturare i requisiti necessari al kernel: solo quando essi saranno interamente soddisfatti, l'applicazione avvierà effettivamente la sua computazione.

Per la gestione dell'ordine di esecuzione dei kernel SYCL definisce due tipi diversi di code:

in-order queue I task sono eseguiti rispettando l'ordine della loro immissione nella coda. Il vantaggio è la semplicità di utilizzo: il programmatore ha il controllo sull'ordine della loro esecuzione. D'altro canto, però, le code in-order non sono ottimizzate dal punto di vista delle performance: tra un kernel e il successivo avviene sempre una sincronizzazione con la CPU e non è possibile sovrapporre l'esecuzione di due task indipendenti fra loro.

out-of-order queue I task non sono eseguiti necessariamente nell'ordine con il quale sono immessi nella coda. I kernel, infatti, possono essere eseguiti in un ordine qualsiasi compatibilmente con le eventuali dipendenze che sussistono fra loro. La

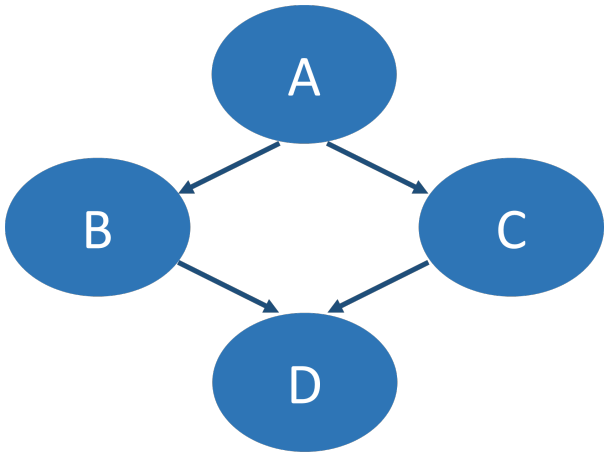


Figura 3.2: *Il grafico rappresenta le dipendenze che sussistono fra i quattro kernel A, B, C e D (le frecce esprimono la relazione "dipende da"). Si deduce che B e C devono essere eseguiti prima di A ma dopo D*

forma più comune che permette di stabilire un loro ordine corretto di esecuzione si basa sui dati.

Ad esempio, si osservi la figura 3.2 che rappresenta i quattro kernel A, B, C e D e le loro dipendenze: si ipotizzi che il kernel A necessiti di accedere in lettura alle strutture dati `b_data` e `c_data` riempite rispettivamente dai kernel B e C; a loro volta essi necessitano di accedere in lettura alla struttura `d_data` del task D per effettuare le loro computazioni. Da questo schema è evidente che il kernel D deve essere eseguito per primo mentre il kernel A per ultimo. I task B e C, invece, possono essere eseguiti in un ordine qualsiasi: o prima B e poi C, o il viceversa, o in parallelo.

Il vantaggio di questo approccio è che l'applicazione può ottimizzare l'esecuzione dei kernel; lo svantaggio, però, è che il programmatore perde il controllo sull'ordine effettivo con il quale i task sono eseguiti.

Da questa descrizione è evidente che serve un meccanismo per catturare le dipendenze fra task diversi. SYCL fornisce non solo delle istruzioni esplicite di sincronizzazione, come

le autoesplicative `wait()` e `depends_on()`, ma realizza anche un'astrazione di alto livello che cattura tali dipendenze in modo automatico. A questo scopo, il programmatore specifica all'interno dei command group (inviati alla stessa coda out-of-order) le risorse necessarie per i kernel definiti al loro interno. A questo punto, l'applicazione è in grado di ricavare da queste indicazioni un grafo diretto aciclico (DAG) delle loro dipendenze e, di conseguenza, può definire un ordine di esecuzione dei kernel che sia il più possibile ottimizzato.

Modello di esecuzione dei kernel

Un dispositivo è suddiviso in una o più *Compute Unit* (CU), ciascuna delle quali è a sua volta divisa in *Processing Elements* (PE). Un'implementazione di SYCL che vuole eseguire le istruzioni di un kernel deve effettuare le sue computazioni sui PE del dispositivo in questione. Questi possono eseguire le istruzioni o con un approccio *Single Instruction - Multiple Data* (SIMD), o con il modello *Single Program - Multiple Data* (SPMD), o con una combinazione dei due.

Quando un kernel SYCL viene lanciato, si crea uno spazio di indici chiamato *ND-range*, ossia spazio a N dimensioni, dove N può valere 1, 2 o 3 ed è detto *rango*[8]. Ciascun punto di questo spazio è chiamato work-item ed è eseguito da un PE. I work-item sono raggruppati in work-group la cui dimensione è definita dal programmatore in base al dispositivo sul quale vengono eseguiti: mantenere corrispondenza tra la dimensione delle CU e la dimensione dei work-group garantisce prestazioni ottimali. La figura (figura 3.3) raffigura la gerarchia di questo modello di esecuzione.

L'intero spazio degli indici è definito da un oggetto `nd_range<rank>` e può essere descritto tramite due componenti: un *global range*, ovvero il numero totale di work-item per ogni dimensione, e un *local range*, il numero di work-item per ogni work-group in ogni dimensione.

[8] *rank* in inglese

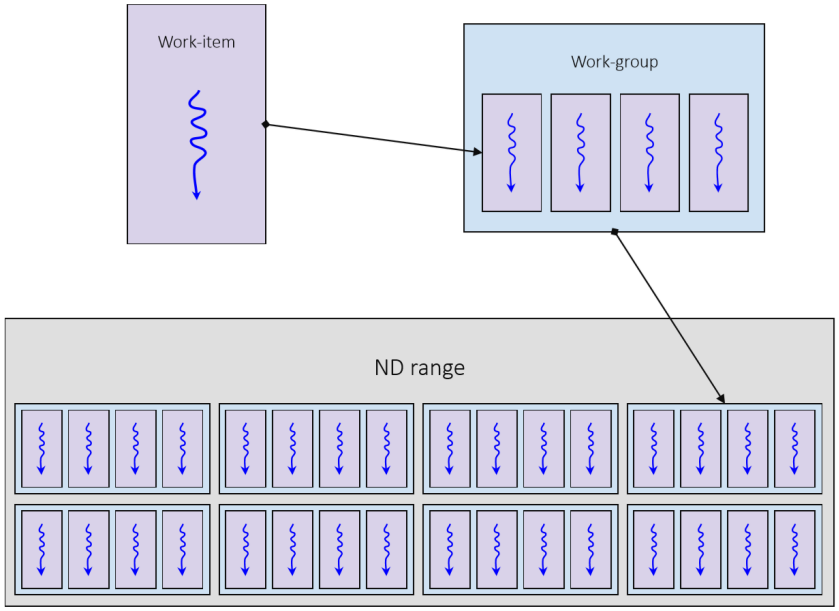


Figura 3.3: *Gerarchia del modello di esecuzione: un ND-range è formato da diversi work-group a loro volta formati da un insieme di work-item*

Fonte: *SYCL Academy [34, Lesson 3 - slide 6]*

I work-item possono essere identificati univocamente sia dal punto di vista globale dell'ND-range, sia dal punto di vista locale dei singoli work-group. Ad esempio, i seguenti indici si riferiscono al work-item evidenziato nella figura 3.4. Si noti che l'ND-range dell'immagine ha global range {12, 12} e local range {3, 3}, da cui si ricava un group range di {4, 4}.

- Global id: (6, 5)
- Group id: (1, 1)
- Local id: (2, 1)

Generalmente, il numero di work-item all'interno di un ND-range può anche essere di alcune migliaia. L'ordine di esecuzione dei work-item è lasciato libero alla specifi-

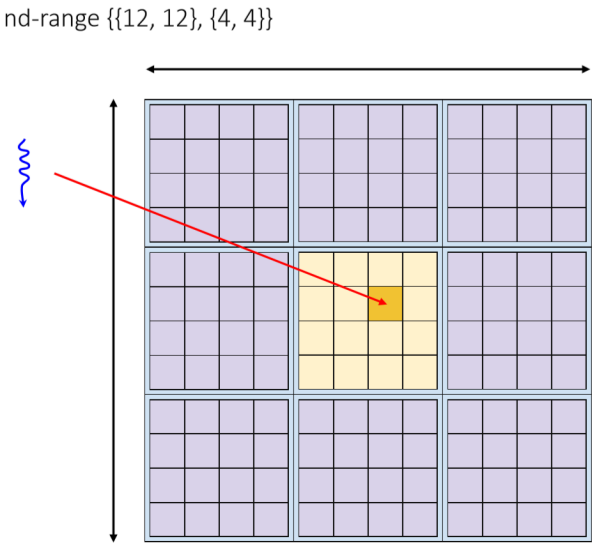


Figura 3.4: *Esempio di work-item collocato in un work-group di un ND-range*

Fonte: *SYCL Academy [34, Lesson 3 - slide 8]*

ca implementazione di SYCL; tuttavia, i work-item di uno stesso work-group possono utilizzare delle istruzioni per sincronizzare la loro esecuzione.

3.1.4 Il modello di gestione della memoria

SYCL definisce tre aree di memoria con diverse velocità di accesso e dimensione (figura 3.5). Per prima cosa, ogni work-item ha a disposizione uno spazio privato chiamato *private memory*; le variabili memorizzate al suo interno non sono visibili agli altri work-item. Ad ogni work-group è poi associato un ulteriore spazio di memoria, chiamato *local memory*, al quale hanno accesso i suoi work-item sia in lettura che in scrittura; questa memoria permette sia la condivisione dei risultati parziali delle computazioni, sia di condividere variabili fra i work-item di uno stesso work-group. Infine, tutti i work-item che fanno parte di uno stesso ND-range hanno a disposizione una terza area di memoria che può essere utilizzata sia in lettura che in scrittura. Questo spazio, chiamato *global memory*, comprende anche un'area riservata per costanti e variabili read-only. La memoria

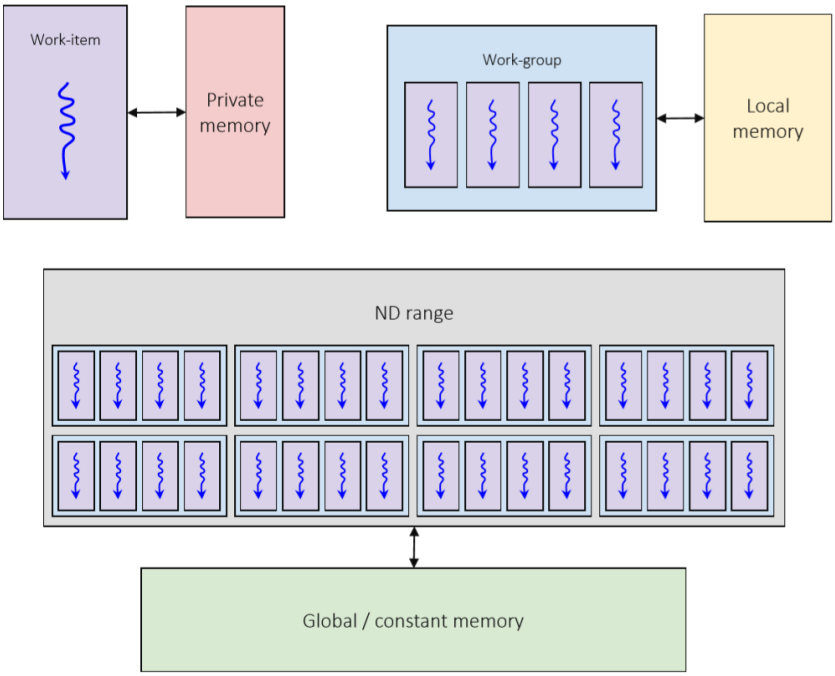


Figura 3.5: *Aree di memoria privata, locale e globale definite dallo standard SYCL*

Fonte: *SYCL Academy [34, Lesson 4 - slide 6]*

globale ha dimensione maggiore delle altre ma è anche quella il cui accesso è più lento; in contrapposizione, la memoria privata è la più piccola ma quella più veloce a cui accedere.

È importante sottolineare che SYCL non garantisce coerenza fra i dati salvati nei tre diversi spazi di memoria. Solo per la memoria privata e quella locale sono previste delle istruzioni di sincronizzazione, ma durante l'esecuzione non ci sono garanzie di consistenza fra work-item appartenenti a work-group diversi.

SYCL prevede due meccanismi distinti per la memorizzazione dei dati e per il loro accesso: i `buffer` gestiscono lo storage dei dati, mentre gli `accessor` sono gli oggetti che permettono ai kernel di accedere alle informazioni. Gli accessor sono interpretati diversamente dai compilatori di host e dispositivi: il primo li utilizza per determinare le dipendenze fra i dati dei kernel, il secondo li interpreta come puntatori alla memoria dei dispositivi ai quali si può accedere in lettura e scrittura.

Esistono diversi modi per utilizzare buffer e accessor; in questo paragrafo presenterò il più comune per dare un'intuizione del loro funzionamento. Per prima cosa si crea un buffer: esso è un puntatore alla memoria dell'host che contiene i dati da elaborare tramite gli acceleratori. Nel momento in cui si definisce, gli indirizzi a cui punta diventano di proprietà del SYCL runtime (figura 3.6a). I buffer hanno un comportamento lazy: la loro creazione non comporta lo spostamento dei dati verso il dispositivo che li elaborerà; questa operazione richiede l'utilizzo degli accessor. Nel momento in cui lo scheduler SYCL avvia l'esecuzione di un command group, si cercano di soddisfare tutti i requisiti necessari per il kernel definito al suo interno. In questa fase si istanziano gli accessor specificando per ciascuno di essi il buffer contenente i dati per il quale deve fare da tramite (figura 3.6b). In questo modo gli accessor possono copiare sull'acceleratore i dati puntati dai buffer solo quando essi saranno considerati pronti dall'applicazione (figura 3.6c).

Lo standard prevede che i dati rimangano nella memoria dei dispositivi anche al termine dell'esecuzione del kernel che li ha elaborati. Può succedere, infatti, che debba essere avviato un ulteriore kernel sullo stesso dispositivo. In questo modo, i dati si troveranno già nella sua memoria e non dovrà avvenire un ulteriore trasferimento. Se, però, il kernel successivo che ne fa richiesta è eseguito su un dispositivo differente, i dati devono essere spostati nella sua memoria. Questo, nei casi peggiori, può richiedere un duplice spostamento che coinvolge l'host come dispositivo intermedio.

La distruzione di un `buffer` è un'operazione bloccante: si attende la terminazione di tutti i work-item che accedono in lettura o scrittura ai suoi dati; poi, se necessario, si effettua una copia del suo contenuto sull'host.

Questo paragrafo si conclude con un esempio: il codice 3.4 è il frammento di un'applicazione SYCL che effettua la somma di due vettori, `dA` e `dB`, salvando il risultato in `dO`. La creazione dei buffer si trova alle righe 3-5; essi sono oggetti di un class template che richiede di indicare esplicitamente il tipo dei dati e il loro rango. Si noti che la costruzione dei buffer richiede il puntatore all'inizio della memoria dei tre vettori `dA`, `dB`

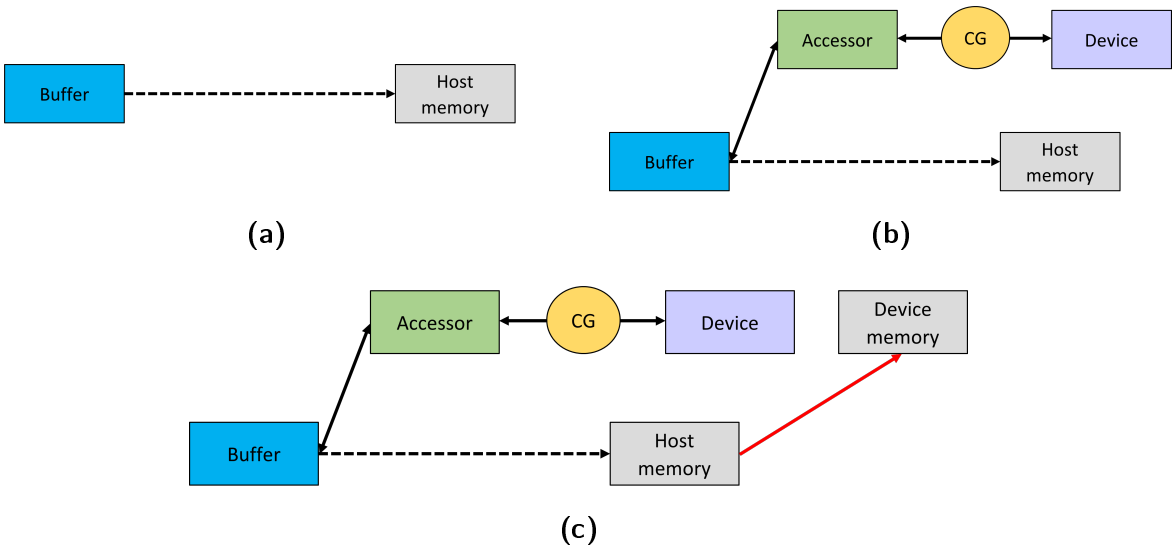


Figura 3.6: *La figura mostra le dipendenze che sussistono fra buffer e accessor. I buffer puntano alla memoria dell'host (figura 3.6a), gli accessor collegano la memoria dell'acceleratore a quella dell'host (figura 3.6b) avviando la copia dei dati solo quando questi sono ritenuti pronti (figura 3.6c)*

Fonte: *SYCL Academy [34, Lesson 4 - slide 11, 12, 14]*

e `dO` e la loro dimensione definita tramite l'oggetto `range<1>`.

La definizione degli accessor si trova, invece, all'interno del command group alle righe 8-10. Anche questi sono oggetti di un class template i cui parametri nell'esempio specificano i privilegi di accesso ai dati, i quali possono essere di sola lettura, di sola scrittura, o entrambi. Altri parametri del template sono, ad esempio, il tipo degli elementi contenuti nei buffer, il rango e la memoria sulla quale i dati saranno allocati.

Si noti, infine, che per scorrere gli indici dei buffer si utilizza un oggetto `id<1>`, dove 1 è il rango dei dati (monodimensionale in questo esempio). Maggiori informazioni sul `parallel_for` si trovano nel paragrafo 3.1.5.

```
1 std::vector<float> dA{...}, dB{...}, dO{...};
2
```

```
3  buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
4  buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
5  buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));
6
7  gpuQueue.submit([&](handler &cgh){
8    auto inA = bufA.get_access<access::mode::read>(cgh);
9    auto inB = bufB.get_access<access::mode::read>(cgh);
10   auto out = bufO.get_access<access::mode::write>(cgh);
11
12   cgh.parallel_for<add>(range<1>(dA.size()),[=](id<1> i){
13     out[i] = inA[i] + inB[i];
14   });
15 });
```

Codice 3.4: *Esempio di utilizzo di buffer e accessor in un'applicazione SYCL che effettua la somma di due array*

Unified Shared Memory

La versione di SYCL rilasciata a Giugno 2020 [1] introduce un nuovo modello di gestione e manipolazione della memoria: la Unified Shared Memory (USM). Essa fornisce uno spazio unificato di indirizzamento tra host e dispositivi basato su puntatori che permette una più facile integrazione con i codici esistenti che utilizzano questo stesso approccio (ad esempio C, C++ e CUDA).

La prima implementazione di SYCL che ha introdotto la USM è DPC++ (si veda la sezione 3.2 per maggiori dettagli). Essa è nata con questa particolare modalità di indirizzamento apparsa sin dal suo rilascio molto promettente. Per questo motivo si è deciso di introdurre la USM nelle successive specifiche dello standard.

Con la USM tutte le architetture utilizzano lo stesso spazio di indirizzamento. Questo non significa che i dati siano accessibili a tutti i dispositivi, ma piuttosto garantisce

coerenza sui valori dei puntatori: indipendentemente dall'hardware che li utilizza, essi si riferiscono sempre alla stessa posizione in memoria.

La USM definisce tre tipi differenti di allocazione [36] riassunti nella tabella 3.1:

device l'host alloca uno spazio di memoria sul dispositivo al quale solo il dispositivo stesso può accedere.

host l'host alloca uno spazio nella sua memoria accessibile da qualsiasi dispositivo. Questo significa che il valore del puntatore è valido sia nel codice dell'host che nei kernel. Tuttavia, quando si accede a dati memorizzati in questo modo, essi provengono sempre dalla memoria dell'host: se un dispositivo ne richiede l'accesso, i dati non sono copiati nella sua memoria, ma sono inviati alla specifica architettura generalmente tramite un bus.

shared l'host alloca uno spazio di memoria accessibile da qualsiasi dispositivo. Questo spazio, però, è puramente virtuale: i dati si troveranno fisicamente sulla memoria del dispositivo se è lui che li sta manipolando, viceversa si troveranno sulla memoria dell'host.

La USM è una funzionalità opzionale e potrebbe non essere supportata da tutti i dispositivi; gli hardware che la implementano, inoltre, potrebbero supportare solo alcuni dei livelli tra quelli proposti.

L'utilizzo della USM comporta che non sia più possibile per il SYCL Runtime costruire automaticamente il DAG delle dipendenze fra i kernel. Sono state individuate alcune soluzioni a questo problema. La prima prevede l'utilizzo della *in-order-queue* al posto della *out-of-order-queue* di default, questo però comporta un drastico peggioramento delle performance. Una seconda alternativa vede l'impiego di istruzioni di sincronizzazione manuale: in questo modo si forza il runtime ad attendere il completamento di una computazione prima di avviare la successiva. Un terzo approccio comporta l'utilizzo del metodo `depends_on()` della classe `handler`: esso accetta uno o più eventi e informa

Tabella 3.1: *Tipi di allocazione della USM*

Tipo di allocazione	Descrizione	Accessibile dall'host?	Accessibile dal device?	Posizione
device	Allocazione della memoria del dispositivo	NO	SI	dispositivo
host	Allocazione nella memoria dell'host	SI	SI	host
shared	Allocazione condivisa tra host e dispositivo	SI	SI	può migrare tra host e device

il runtime che il command group associato all'handler può iniziare solo quando gli eventi in input sono terminati[9].

3.1.5 Il modello di programmazione

Come introdotto nelle sezioni precedenti, i kernl sono codici che possono essere eseguiti su dispositivi eterogenei e sono compilati direttamente dal dispositivo incaricato della loro esecuzione. Tali funzioni devono essere associate ad un command group il quale è responsabile di accodarle ad una queue e, conseguentemente, ad un dispositivo. È importante ricordare che per ogni command group si può definire un'unica kernel function.

A livello implementativo, i kernel sono function object tipicamente creati utilizzando delle lambda expression e sono passati in input ad uno dei metodi della classe `handler`. Un oggetto di questa classe è il gestore delle istruzioni di un command group. Se si osserva il codice 3.1 alla riga 17, si nota che la lambda expression che definisce il command group prende in input un oggetto della classe `handler` chiamato `cgh` nei codici seguenti.

[9]Per maggiori dettagli sulla USM si rimanda a [36], capitoli 3 e 6, e a [1], sezione 4.8.

Questo oggetto è utilizzato all'interno dello scope del command group per riferirsi al command group stesso.

Esistono quattro metodi per la definizione di un kernel, ciascuno dei quali esprime una forma diversa di parallelismo:

- Il metodo `single_task()` permette di eseguire una computazione su un singolo work-item (codice 3.5).

```
1 cgh.single_task([=](){
2   // code
3 });
```

Codice 3.5: *Kernel definito con il metodo* `single_task()`

- Il `parallel_for()` è il modo più comune per inviare un task a tutti i work-item di un ND-range e può essere configurato in diversi modi, come descritto in seguito. Un esempio è riportato nel codice 3.6:

```
1 cgh.parallel_for(range<2>(64, 64), [=](id<2> idx){
2   // code
3 });
```

Codice 3.6: *Kernel definito con il metodo* `parallel_for()`

- I metodi `parallel_for_work_group()` e `parallel_for_work_item()` sono utilizzati per implementare il cosiddetto *parallelismo gerarchico*. Il primo metodo definisce un task che deve essere gestito da un work-group; all'interno di questo blocco si può utilizzare il secondo metodo per esprimere un codice che deve essere eseguito da tutti i work-item del work-group in questione (codice 3.7). In questo modo è possibile differenziare le istruzioni fra work-item di work-group diversi.

```
cgh.parallel_for_work_group(range<2>(64, 64), [=](group<2> gp){
  // SYCL kernel function is executed once per work-group
  parallel_for_work_item(gp, [=](h_item<2> it){
    // SYCL kernel function is executed once per work-item
  });
});
```

Codice 3.7: *Kernel definito con i metodi `parallel_for_work_group()` e `parallel_for_work_item()`*

Gli ultimi tre metodi dell'elenco, oltre alla kernel function, richiedono in input anche altre variabili al fine di definire la configurazione dell'ND-range. Inoltre, la lambda expression del kernel richiede a sua volta un ulteriore input necessario per iterare su ciascun work-item.

Esistono diversi modi per configurare lo spazio degli indici, il codice 3.6 mostra uno dei più comuni: esso utilizza un oggetto `range<rank>(num-item)`, dove `rank` specifica il rango dell'ND-range e `num_item` il numero di work-item dello spazio. Un secondo approccio altrettanto comune è mostrato nel codice 3.8, in questo caso si utilizza un oggetto `nd_range<rank>`. Questo metodo di configurazione è utilizzato in quei casi in cui si vuole specificare la dimensione dei work-group; infatti, la costruzione di un oggetto `nd_range` richiede come argomenti due oggetti `range` che indicano rispettivamente il global range e il local range: il primo corrisponde al numero di work-item dell'intero ND-range, il secondo indica il numero di work-item per work-group.

```
cgh.parallel_for(nd_range<1>(range<1>(1024), range<1>(32)),
    [=](item<1> item){
  //kernel code
});
```

Codice 3.8: *Configurazione di un ND-range tramite l'utilizzo di un oggetto `nd_range<dim>`*

Sono previsti diversi modi anche per iterare sullo spazio degli indici, il più comune dei quali è mostrato nel codice 3.8. Si può notare che il kernel prende in input un oggetto `item<dim>` il quale fornisce diversi metodi che permettono di risalire agli indici che identificano univocamente i work-item. In questo modo all'interno del kernel sarà possibile indicizzare tutte le unità di lavoro dell'ND-range.

La gestione degli errori

Il linguaggio C++ offre ai programmatori diverse opzioni per la segnalazione e gestione di errori che si possono verificare durante l'esecuzione di un programma. Una delle più comunemente usate è rappresentata dalle eccezioni e dal corrispondente uso del blocco *try-catch*. Tale uso in un contesto eterogeneo richiede, però, particolare attenzione. Si consideri, infatti, che gli errori che si verificano in applicazioni di questo tipo possono essere suddivisi in:

errori sincroni Si verificano durante l'esecuzione del codice che gira sull'host; generalmente sono causati da situazioni impreviste che riguardano API, oggetti, particolari istruzioni, ecc. Questo tipo di errori può essere gestito secondo le modalità tipiche delle eccezioni. In particolare, per catturare il fallimento delle istruzioni SYCL eseguite dall'applicazione, lo standard definisce la classe `sycl::exception`, il cui utilizzo è conforme alla classe `std::exception`.

errori asincroni Si verificano durante l'esecuzione di un kernel, pertanto riguardano il codice che gira su un dispositivo. Gli errori di questo tipo sono asincroni con il programma eseguito sull'host e, di conseguenza, non possono essere gestiti in modo sincrono. Tuttavia, come tutti gli eventi imprevisti, anche gli errori asincroni producono l'arresto anomalo del programma se non vengono correttamente gestiti.

È evidente che serve un nuovo meccanismo che permetta di gestire all'interno dell'applicazione gli errori asincroni generati da un kernel. Lo standard di Khronos definisce che

tali errori debbano essere intercettati a basso livello dal runtime SYCL e memorizzati all'interno di una lista. L'applicazione potrà poi accedere a questa lista quando lo riterrà opportuno. Questo è possibile grazie all'utilizzo di un *asynchronous error handler*, una funzione associata ad una coda che permette di definire le istruzioni da eseguire per ogni errore generato[10].

In questa sezione si sono trattate le principali caratteristiche di SYCL, quelle più importanti al fine di questa tesi di laurea. Per maggiori dettagli sullo standard si rimanda alla bibliografia [1, 2, 36].

3.2 Le implementazioni di SYCL

Un'implementazione di SYCL tipicamente è costituita da vari componenti disposti in uno stack a più livelli (figura 3.7):

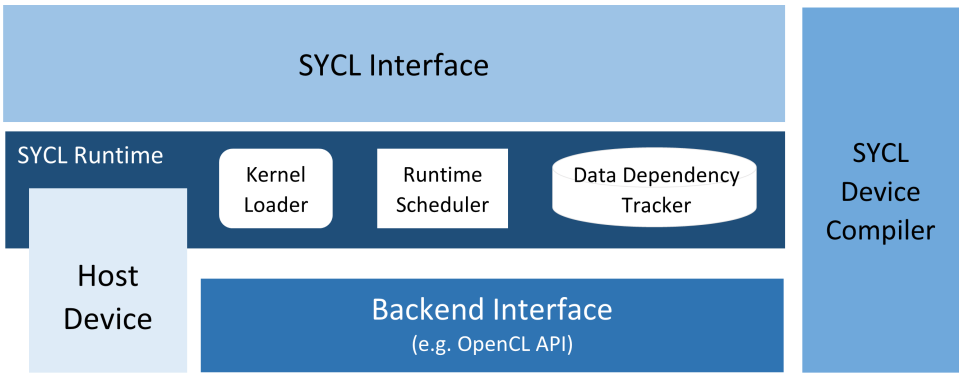


Figura 3.7: *Le varie componenti di una tipica implementazione di SYCL*

Fonte: *SYCL Academy [34, Lesson 1 - slide 11]*

- La *SYCL Interface* è una libreria C++ conforme allo standard SYCL che astrae l'interazione con host e dispositivi.

[10]Per maggiori informazioni sulla gestione degli errori in SYCL si veda [36], capitolo 5.

- Il *SYCL Runtime* si occupa dello scheduling dei comandi, del caricamento dei kernel e della gestione delle dipendenze dei dati fra host e dispositivi.

- L'*Host Device* comprende il back-end per l'esecuzione del codice C++ nativo; è il responsabile dell'emulazione dei modelli di esecuzione, gestione della memoria e programmazione definiti dallo standard. Può essere utilizzato come acceleratore in assenza di altri dispositivi e per effettuare debugging di codice.

- La *Backend Interface* implementa la comunicazione tra il runtime e i dispositivi fisici per l'effettiva esecuzione delle istruzioni sulle specifiche architetture hardware. Lo standard è OpenCL, ma sono state sviluppate di recente anche altre interfacce quali CUDA per le schede NVIDIA e Level Zero per i dispositivi Intel.

- Il *SYCL Device Compiler* è il compilatore di codice C++ che riconosce e compila i kernel da eseguire sugli acceleratori. Il risultato di questo processo, come descritto nella sezione 3.1.1, può essere o il codice binario specifico per un certo dispositivo, o una rappresentazione intermedia (IR) generica.

Sopra a questo ecosistema si possono trovare anche librerie di alto livello che si poggiano su SYCL per implementare routine per contesti specifici.

Attualmente esistono quattro diverse implementazioni di SYCL: Data Parallel C++, ComputeCpp, triSYCL e hipSYCL. La figura figura 3.8 riassume il panorama attuale, in particolare mostra i dispositivi supportati dalle diverse implementazioni. Di seguito si trova una breve descrizione di ciascuna di esse, seguendo l'ordine cronologico nel quale sono state rilasciate.

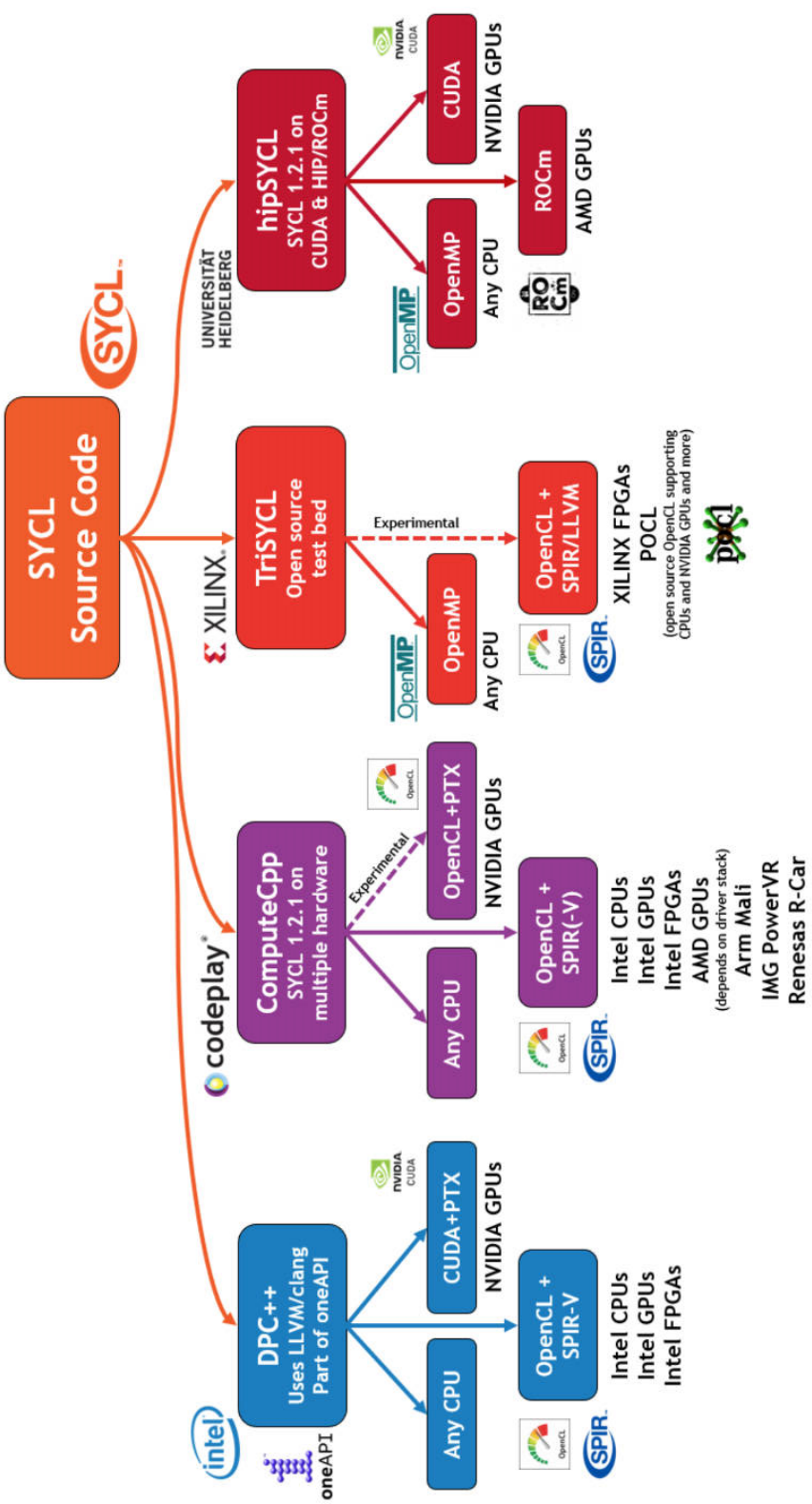


Figura 3.8: *Le implementazioni di SYCL*

Fonte: `https://www.khronos.org/sycl/`

ComputeCpp

ComputeCpp [33] è un'implementazione commerciale di SYCL 1.2.1 realizzata da Codeplay[11] [37] a partire dal 2014. Esiste però una *community edition* gratuita, completa di tutto l'ecosistema SYCL, pensata per una prima valutazione del prodotto.

ComputeCpp è stata la prima implementazione di SYCL ad aver superato il test di conformità di Khronos in quanto soddisfa pienamente tutte le specifiche dello standard. Essa definisce, inoltre, delle estensioni di SYCL che potranno essere incluse nelle sue future specifiche, tra queste una modalità più espressiva per rappresentare il data-flow di C++. ComputeCpp ha recentemente introdotto il supporto alla USM uniformandosi alla versione di SYCL rilasciata a Giugno 2020.

L'ecosistema che circonda ComputeCpp è ampio: oltre ai differenti back-end supportati dall'implementazione, essa contiene alcuni strumenti di supporto alla programmazione e alcune librerie che forniscono routine di più alto livello di intelligenza artificiale e algebra lineare, ad esempio *SYCL-ML*, *SYCL-DNN* e *SYCL-BLAS*.

triSYCL

ComputeCpp è stata la prima implementazione di SYCL ad essere rilasciata, ma il suo codice non era libero e questo ne rallentava la diffusione. L'azienda AMD decise pertanto di avviare un progetto di ricerca con l'obbiettivo di studiare e validare lo standard proposto da Khronos [35]. Nacque così la seconda implementazione di SYCL che prese il nome di triSYCL; il suo codice è pubblico su GitHub [38].

Attualmente il progetto è ancora attivo ed è supportato principalmente da Xilinx, uno tra i più importanti produttori di FPGA. Nonostante l'implementazione non sia completa e ne sia sconsigliato l'uso ad utenti finali, è utilizzata principalmente con lo

[11] Codeplay è un'azienda irlandese che produce software come compilatori, debugger, runtime, sistemi di testing, ecc sia per sistemi eterogenei che per architetture single purpose. Collabora, inoltre, per la definizione e la ricerca di standard quali SYCL e C++.

scopo di sperimentare possibili soluzioni implementative dello standard, in particolare per esplorare dei back-end alternativi. Recentemente, anche RedHat, Intel e Khronos hanno iniziato a contribuire al progetto: la comunità scientifica ha visto in triSYCL un'utile implementazione per effettuare investigazioni sullo standard. Ad oggi, oltre a fornire importanti feedback a Khronos, triSYCL si sta rivelando un progetto in grado di dare suggerimenti anche al comitato ISO C++.

Tra i principali risultati ottenuti da triSYCL, vale la pena menzionare il prototipo del compilatore basato su tecnologia clang/LLVM[12] [39] che permette la programmazione in SYCL per FPGA. Questo compilatore è stato preso da Intel come punto di partenza per lo sviluppo di Data Parallel C++ (si vedano le sezioni successive).

hipSYCL

hipSYCL è un'implementazione di SYCL realizzata dalla Heidelberg University e rilasciata a Settembre 2019 [40]. Il progetto è ancora in fase di sviluppo per cui l'implementazione è incompleta.

Ciò che differenzia hipSYCL dalle altre implementazioni è che non usa il back-end OpenCL, ma ne ha definito uno per l'ambiente AMD HIP, così da abilitare l'uso sia delle GPU AMD sia di quelle NVIDIA. Questa scelta consente di utilizzare nelle applicazioni hipSYCL anche codice scritto in CUDA o in ROCm, permettendo l'interoperabilità fra tutti gli ambienti.

hipSYCL comprende un runtime SYCL che gira sopra i runtime CUDA/HIP e un compilatore realizzato come plugin di clang. Anche questa implementazione di SYCL è pubblica e disponibile su GitHub [41].

[12]clang è un compilatore per i linguaggi della famiglia C (C, C++, Objective C/C++, OpenCL, CUDA, RenderScript) sviluppato sull'infrastruttura LLVM, un insieme di moduli riutilizzabili per la compilazione di linguaggi; è un software libero concorrente di GCC.

Data Parallel C++

Data Parallel C++, o DPC++ in breve, è l'implementazione più recente di SYCL ed è inclusa in oneAPI, il toolkit di Intel per la programmazione parallela su dispositivi eterogenei[13] [42] [19]. Intel propone oneAPI come nuovo modello di programmazione aperto che implementa ed estende gli standard esistenti. Al progetto collaborano anche altre aziende del settore ICT e alcuni istituti di ricerca. Il suo obiettivo è quello di creare un modello di programmazione unificato che permetta di scrivere codice che possa essere eseguito su un'ampia varietà di dispositivi diversi, come CPU, GPU e FPGA.

Il cuore del toolkit è il linguaggio DPC++ [36], la vera è propria implementazione di SYCL. Tale linguaggio si basa su C++ 17 estendendolo con le funzionalità dello standard di Khronos.

Per quanto riguarda il back-end, oneAPI comprende un'interfaccia hardware di basso livello chiamata *Level Zero*. Questa si interfaccia direttamente con il runtime di SYCL e fornisce una serie di funzionalità e servizi per ottenere il massimo delle prestazioni dai dispositivi Intel che la implementano. oneAPI supporta altri due back-end: OpenCL e CUDA[14].

oneAPI comprende alcuni strumenti che supportano i programmatori nello sviluppo di codice DPC++. Tra questi si evidenzia il *Data Parallel Compatibility Tool*, uno strumento che agevola la conversione di applicazioni da CUDA a SYCL, applicando trasformazioni automatiche per gran parte del codice, con evidenti benefici in termini di velocità e di correttezza. L'output del compatibility tool è un codice DPC++ nel quale sono evidenziate tramite dei commenti tutte le istruzioni che questo strumento non è riuscito a convertire o che richiedono l'intervento del programmatore per un loro raffinamento.

[13] La versione Beta10 di oneAPI è stata rilasciata a fine Ottobre 2020 ma è disponibile una versione 1.0 già da fine Settembre 2020.

[14] Il back-end CUDA di oneAPI è stato realizzato da CodePlay.

Il compilatore di DPC++, chiamato `dpcpp`, è stato realizzato con la collaborazione degli sviluppatori di triSYCL ed è liberamente disponibile su GitHub [43].

Un aspetto che distingue DPC++ dalle altre implementazioni di SYCL è il supporto per il modello di memoria basato su USM ed è la motivazione principale per la quale il lavoro di tesi si è svolto utilizzando questa specifica implementazione.

Capitolo 4

Software eterogeneo nel tracciamento del rivelatore a pixel

4.1 Il codice Pixeltrack Standalone

L'attività Patatrack, come ho introdotto nella sezione 2.2.3, si occupa di ricerca e sviluppo per il software dell'esperimento CMS. Il gruppo che la persegue è eterogeneo: non comprende solo ricercatori del CERN, ma anche studenti e studiosi provenienti da altri centri di ricerca, tra i quali il Fermilab[1] [44]. Il gruppo coopera con il CERN openlab [8], un progetto di partnership tra il CERN e alcune organizzazioni di ricerca pubbliche e private, tra le quali importanti aziende ICT come Oracle, Intel e Google.

L'obiettivo per il quale è stato costituito questo gruppo è quello di dimostrare che la ricostruzione online di CMSSW può trarre benefici dall'utilizzo di risorse eterogenee.

Le prime ricerche si sono concentrate nella progettazione e nello sviluppo in CUDA dei moduli dei calorimetri e dei pixel [45]. Le schede grafiche della NVIDIA, infatti, sono facilmente integrabili nella high-level trigger farm dell'esperimento CMS dal momento

[1] Il Fermilab, o Fermi National Accelerator Laboratory, è un laboratorio americano che si occupa di ricerca nell'ambito della fisica delle particelle.

che sono compatte e a basso consumo. In aggiunta a ciò, le conoscenze pregresse dei ricercatori del gruppo sono state un secondo fattore che ha guidato la scelta in questa direzione.

Alcuni risultati significativi ottenuti sono riportati in figura 4.1: i codici di ricostruzione di tripletti e quadrupletti (si veda in seguito per una loro definizione) sono stati eseguiti prima su CPU poi su scheda grafica GPU NVIDIA Tesla T4 [46]; su questa seconda architettura i tripletti hanno evidenziato un miglioramento delle performance del 47% mentre i quadrupletti sono migliorati del 58%. Maggiori dettagli sui test effettuati si trovano in [45].

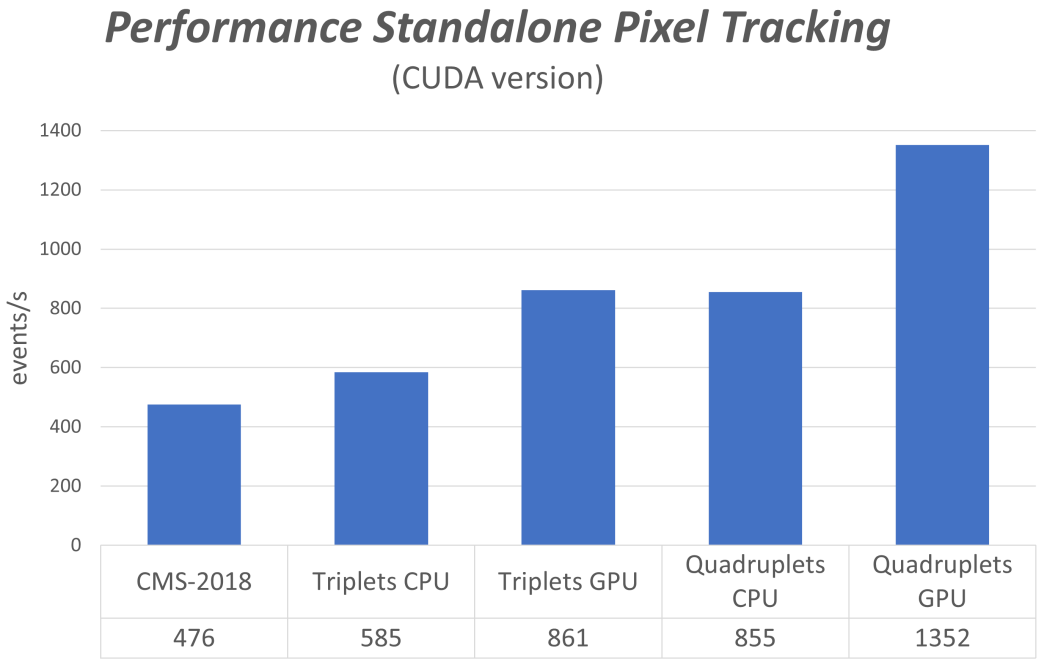


Figura 4.1: *Il grafico mostra i miglioramenti delle prestazioni degli algoritmi CUDA di tripletti e quadrupletti se eseguiti su GPU. Le misure sono espresse in eventi processati al secondo*

Fonte: *[45, pagina 16]*

Tali risultati sono apparsi fin da subito molto promettenti, tant'è che, nonostante queste sperimentazioni siano partite in vista del Run 4, gli algoritmi ottenuti sono già

stati integrati nel software che verrà utilizzato per il Run 3. Tuttavia, le ricerche sull'eterogeneità non si possono ancora dichiarare concluse. Anzi, molte architetture e diverse tecnologie devono ancora essere valutate. Si consideri che su questo fronte il mondo ICT avanza molto velocemente, pertanto è fondamentale individuare fin da subito quelle tecnologie che nel 2027, anno di avvio di HL-LHC, saranno all'avanguardia e non obsolete. Inoltre, visto quanto riportato nella sezione 2.2, è fondamentale incrementare ulteriormente la potenza computazionale disponibile, a parità di costo.

Recentemente è stata creata una versione semplificata di CMSSW online con l'obiettivo di avere un software apposito per valutare le nuove tecnologie per il calcolo eterogeneo parallelo che appaiono sul mercato. In particolare, gli studi si stanno concentrando sugli strumenti software introdotti nella sezione 2.2.3: Alpaka, Kokkos, SYCL/oneAPI. Il codice di questo progetto, chiamato *Standalone Patatrack Pixel Tracking* (nel seguito Pixeltrack Standalone), è pubblico [4]. Esso si basa su una versione più leggera del framework di CMSSW e comprende solo i moduli per la ricostruzione delle tracce acquisite dal rivelatore a pixel del Silicon Tracker; in questo modo è possibile lavorare su un'applicazione che presenti le stesse caratteristiche di base del framework (ad esempio utilizza TBB e gli stessi sistemi per lo scheduling dei moduli), ma che eviti la complessità e le dipendenze dell'intero software.

La prima versione di Pixeltrack Standalone utilizza CUDA per la parte di eterogeneità. La figura 4.2 schematizza il workflow del codice: l'input del sistema sono i raw data (reali o simulati) provenienti dall'elettronica del rivelatore, mentre l'output è costituito dalle traiettorie delle particelle e dai vertici primari, ossia i punti presunti di collisione tra i protoni dei fasci circolanti nell'acceleratore; questi devono essere restituiti con un formato tale per cui possano essere presi in input dagli altri moduli del software per ulteriori processamenti. A questo proposito, si è valutato che l'approccio *Structure of Arrays* (SoA), invece del più comune *Array of Structures* (AoS), consenta di evitare conversioni di formato tra i diversi algoritmi del workflow.

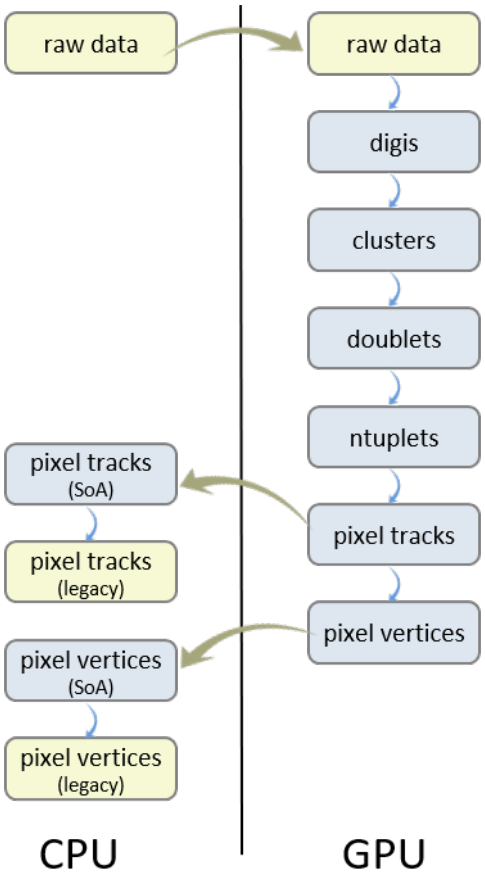


Figura 4.2: *Lo schema rappresenta gli step del software che ricostruisce i vertici (punti di collisione) e le tracce delle particelle a partire dai raw data prodotti dai pixel del Silicon Tracker*

Fonte: *[45, pagina 3]*

Si noti che i risultati della ricostruzione sono restituiti alla CPU solamente al termine dell'intero processo e non avvengono copie intermedie tra host e dispositivo: questo perché lo spostamento dei dati può risultare facilmente il collo di bottiglia dell'intero algoritmo.

Si analizzano ora più in dettaglio gli step del workflow:

- Il modulo *digis* si occupa di convertire i raw data provenienti dall'elettronica del

rivelatore in una forma più facilmente maneggevole chiamata, appunto, *digi*. Gli input si presentano in un formato compresso, pertanto devono essere prima estratti per poter essere interpretati. Un digi rappresenta un singolo pixel sollecitato dal passaggio di una particella e contiene informazioni sulle sue coordinate spaziali e sulla quantità di carica raccolta.

- Il modulo *clusters* si occupa di raggruppare i digi adiacenti: quando una particella attraversa un layer di pixel, infatti, sollecita un gruppo di canali contigui. Durante questa fase, quindi, si uniscono le rilevazioni collegate a un unico hit e si calcola il reale punto di interazione della particella con il sensore.

- I moduli *doublets* e *ntuplets* creano delle associazioni tra i cluster dei diversi strati con l'obiettivo di collegare quelli che sono stati sollecitati dalla stessa particella. Inizialmente si creano coppie di cluster di layer adiacenti; in seguito, tramite un algoritmo di ricerca depth-first, si raggruppano i cluster prima in tripletti, poi in quadrupletti e via via in n-tuple.

- Infine i moduli *pixel tracks* e *pixel vertices* ricostruiscono le traiettorie complete delle particelle associandovi altre informazioni ad esse correlate, come il punto nel quale è avvenuta presumibilmente la collisione.

4.2 La versione SYCL di Pixeltrack Standalone

Il progetto che mi ha coinvolto per la stesura di questa tesi ha l'obiettivo di verificare se il software Pixeltrack Standalone sia riscrivibile in SYCL mantenendo intatta la logica degli algoritmi; il codice deve essere eseguibile su un insieme eterogeneo di acceleratori in linea con l'analisi effettuata nella sezione 2.2. Inoltre, si vogliono ottenere delle prime misure di performance per avere una visione d'insieme dell'integrazione di SYCL con CMSSW.

4.2.1 Da CUDA a oneAPI DPC++

L'implementazione di SYCL selezionata per questo progetto è oneAPI DPC++: essa è il miglior compromesso tra completezza e compatibilità con il software di Patatrack. Da un lato, infatti, hipSYCL e triSYCL non implementano ancora tutte le funzionalità dello standard; in aggiunta, il loro processo di sviluppo non sembra essere veloce quanto servirebbe a CMSSW e le loro caratteristiche tecniche non le rendono conformi alle esigenze del progetto (si veda la sezione 3.2). Per quanto riguarda ComputeCpp, anche se è una versione completa di SYCL, al momento dell'inizio del progetto non implementava soluzioni per una gestione della memoria simili a quelle proposte da CUDA, in particolare non implementava ancora il modello Unified Shared Memory (descritto nella sezione 3.1.4) che invece era già supportato da DPC++. Tale modello garantisce ai ricercatori di non dover stravolgere completamente il codice, ma di mantenere una certa compatibilità con la versione di partenza del progetto.

Nel momento in cui mi sono approcciata al progetto (primavera 2020), oneAPI risultava essere una tecnologia ancora in fase di sviluppo. È in atto, tuttavia, una stretta collaborazione fra Intel e il CERN: da un lato l'azienda statunitense si impegna ad aggiornare i ricercatori sugli sviluppi del suo prodotto e si rende disponibile per chiarimenti ed informazioni; dall'altro lato, i ricercatori utilizzano oneAPI testando le sue componenti in diversi progetti e si impegnano a riportare eventuali bug e consigli implementativi agli sviluppatori di Intel. Questa collaborazione è stata un'ulteriore motivazione che ha portato alla scelta di DPC++ come implementazione di SYCL per Pixeltrack Standalone.

Il software da cui partire per la codifica in DPC++ è la versione CUDA del progetto Standalone. La trasformazione da una forma all'altra ha richiesto diversi passaggi: per prima cosa abbiamo trasformato il framework con lo scopo di implementare uno scheduling efficiente di algoritmi e kernel SYCL. Successivamente abbiamo convertito i moduli di CMSSW che riguardano il rivelatore a pixel dal linguaggio CUDA alla corrispondente versione in SYCL. L'obiettivo di questi passaggi è la possibilità di eseguire il nuovo codice

sia su CPU che su dispositivi eterogenei utilizzando diversi back-end. In questo modo è possibile verificare la reale portabilità del codice e confrontare le prestazioni ottenute sui vari dispositivi.

Il progetto è ancora in fase di sviluppo e ad oggi sono stati convertiti in SYCL il framework, gli algoritmi per decodificare i dati e i moduli *digi* e *clusters* della figura 4.2.

4.2.2 Utilizzo del Data Parallel Compatibility Tool

Abbiamo realizzato una traduzione iniziale del codice grazie al Data Parallel Compatibility Tool, o DPCT. Con questo strumento contenuto nel toolkit oneAPI è stato possibile convertire correttamente in pochi secondi circa l'80-90% del codice CUDA.

Per utilizzare il Compatibility Tool da terminale è necessario configurare l'ambiente OneAPI. A questo punto si può eseguire il comando `dpct` per tradurre un singolo file o un intero progetto. Nel primo caso è sufficiente passare il file CUDA al tool per avviare la traduzione, ad esempio:

```
$ dpct file-da-convertire.cu
```

Per la conversione di un intero progetto è necessario un passaggio preliminare, che consiste nell'intercettare l'output del comando di compilazione del progetto in modo tale da individuare tutte le dipendenze richieste dal codice: esse sono salvate in un file in formato *json*, chiamato di default `compile_commands.json`. Una volta ottenuto questo file, detto *compilation database*, è possibile avviare `dpct` specificando, oltre al compilation database stesso, la directory del progetto, la directory nella quale inserire gli output e i file CUDA da convertire. Un esempio è riportato nel codice 4.1: la prima riga intercetta il comando di compilazione del progetto, mentre la seconda avvia la conversione del file `main.cu`.

Il procedimento che abbiamo utilizzato per il codice Pixeltrack Standalone è il se-

```
$ intercept-build make cuda
$ dpct -p compile_commands.json -in-root=src/cuda/ -out-root=src/
   cuda/dpcpp_out src/cuda/main.cu
```

Codice 4.1: *Utilizzo del DPTC per la traduzione di un intero progetto*

condo: è stato necessario, infatti, intercettare le dipendenze di TBB e di altri software utilizzati dall'applicazione.

Per ogni file CUDA, il DPCT restituisce un file DPC++ con lo stesso nome del sorgente, ma con estensione *.dp.cpp*[2]. Questi file necessitano dell'intervento del programmatore per la conversione delle istruzioni che il Compatibility Tool non è riuscito a trasformare. Il passaggio è reso più semplice grazie ad opportuni commenti posti in corrispondenza dei costrutti problematici. In questo modo il programmatore può trovare un codice di errore e una breve spiegazione del motivo per il quale la trasformazione è fallita o richiede un suo intervento.

Ad esempio, il DPCT non è in grado di convertire la gestione degli errori asincroni di CUDA. La tecnologia NVIDIA, infatti, prevede che le istruzioni asincrone restituiscano un valore che segnali se la sottomissione dell'operazione ha avuto successo; SYCL, invece, utilizza un meccanismo completamente diverso che si basa sugli asynchronous error handler (vedi sezione 3.1.5). In questi casi il Compatibility Tool riporta un messaggio di errore simile a quello del codice 4.2.

Una volta che il programmatore ha terminato la revisione del codice, è possibile avviare la sua compilazione con il compilatore per DPC++ e far partire l'eseguibile ottenuto.

[2]Oltre ai file DPC++, il Compatibility Tool crea un file in formato YAML per ogni file CUDA che ha richiesto una conversione; questi file contengono alcune informazioni sulle trasformazioni effettuate. Queste informazioni, però, risultano di non facile comprensione e non sono state utilizzate per questo progetto.

```
/*
  DPCT1003:12: Migrated API does not return error code.
  (*, 0) is inserted. You may need to rewrite this code.
*/
```

Codice 4.2: *Codice di errore del DPCT*

Esempio di conversione con il DPCT

In questa sezione si riporta un semplice esempio per capire come il Compatibility Tool traduce in DPC++ i programmi scritti in CUDA. Il codice 4.3 implementa un kernel che riempie in parallelo un vettore effettuando una semplice somma per ognuno dei suoi elementi. La traduzione di tale programma è riportata successivamente nel codice 4.4[3].

```
1  #include <cuda.h>
2  #include <stdio.h>
3
4  const int vector_size = 256;
5
6  __global__ void SimpleAddKernel(float *A, int offset) {
7    A[threadIdx.x] = threadIdx.x + offset;
8  }
9
10 int main() {
11   float *d_A;
12   int offset = 10000;
13
14   cudaMalloc( &d_A, vector_size * sizeof( float ) );
15   SimpleAddKernel<<<1, vector_size>>>(d_A, offset);
16
```

[3]Il codice di esempio è stato tratto dal manuale online del Compatibility Tool [47]

```
17   float result[vector_size] = { };
18   cudaMemcpy(result, d_A, vector_size*sizeof(float),
19              cudaMemcpyDeviceToHost);
20   cudaFree( d_A );
21
22   for (int i = 0; i < vector_size; ++i)
23     printf( "%.1f ", result[i] );
24
25   return 0;
26 }
```

Codice 4.3: *Semplice codice CUDA la cui traduzione in DPC++ è riportata nel codice 4.4*

```
1  #include <CL/sycl.hpp>
2  #include <dpct/dpct.hpp>
3  #include <stdio.h>
4
5  const int vector_size = 256;
6
7  void SimpleAddKernel(float *A, int offset, sycl::nd_item<3> item_ct1){
8    A[item_ct1.get_local_id(2)] = item_ct1.get_local_id(2) + offset;
9  }
10
11 int main(){
12   dpct::device_ext &dev_ct1 = dpct::get_current_device();
13   sycl::queue &q_ct1 = dev_ct1.default_queue();
14
15   float *d_A;
16   int offset = 10000;
17
18   d_A = sycl::malloc_device<float>(vector_size, q_ct1);
19
```

```
20   q_ct1.submit([&](sycl::handler &cgh) {
21     cgh.parallel_for(sycl::nd_range(sycl::range(1, 1, vector_size),
22                                     sycl::range(1, 1, vector_size)),
23                      [=](sycl::nd_item<3> item_ct1) {
24                        SimpleAddKernel(d_A, offset, item_ct1);
25                      });
26   });
27
28   float result[vector_size] = { };
29   q_ct1.memcpy(result, d_A, vector_size * sizeof(float)).wait();
30
31   sycl::free(d_A, q_ct1);
32
33   for (int i = 0; i < vector_size; ++i) {
34     printf( "%.1f ", result[i] );
35
36   return 0;
37 }
```

Codice 4.4: *Traduzione in DPC++ del codice 4.3 scritto in CUDA. La conversione è stata realizzata dal compatibility tool*

Si può notare, per prima cosa, che l'header file <cuda.h> è stato sostituito dagli header di SYCL e del DPCT.

Per indicizzare ogni singola unità computazionale all'interno dei kernel, CUDA fornisce ai programmatori l'oggetto `threadIdx` e i suoi campi `x`, `y`, `z`. Nel codice in analisi, dal momento che lo spazio dei thread è monodimensionale, è sufficiente l'istruzione `threadIdx.x` per risalire alle coordinate univoche di ogni singolo thread (codice 4.3, riga 7). DPC++, invece, richiede che sia passato esplicitamente un oggetto in input a ciascun kernel per poter indicizzare i work-item al suo interno. A questo riguardo, si evidenzia che il Compatibility Tool predilige sempre un ND-range a tre dimensioni anche quando, come in questo caso, il codice CUDA definisce uno spazio lineare. Nel

codice 4.4 la funzione `SimpleAddKernel()` che contiene le istruzioni da parallelizzare necessita, quindi, del parametro aggiuntivo `sycl::nd_item<3> item_ct1`; la classe `sycl::nd_item<3>` implementa diversi metodi per risalire agli indici di ciascun work-item, nel codice di esempio è utilizzato il metodo `get_local_id(rank)` che restituisce l'indice locale del thread. Si noti che DPC++ indicizza le dimensioni in ordine inverso: la dimensione x corrisponde all'indice 2, la dimensione y all'indice 1 e la dimensione z all'indice 0.

In tutte le applicazioni SYCL è necessario definire la queue alla quale inviare il kernel da eseguire. Nel codice 4.4 questo è implementato alle righe 12-13: si individua il dispositivo con le caratteristiche migliori e a esso si associa di default la coda in-order, `q_ct1` nell'esempio. Queste sono le impostazioni standard definite dal Compatibility Tool; i programmatori possono eventualmente modificare tali parametri durante la fase di revisione del codice.

Nel `main` del codice CUDA, a riga 14, si alloca un vettore `dA` sulla GPU grazie alla funzione `cudaMalloc()`; questa istruzione è stata convertita con l'equivalente `malloc_device()` definita da SYCL. Questa trasformazione evidenzia l'utilità della USM di SYCL e del suo focus sui puntatori. L'alternativa avrebbe infatti richiesto la definizione di un `buffer` e l'uso di un `accessor` nel command group.

La differenza maggiore fra i due codici è ben evidente nell'invocazione dei kernel. Il primo (riga 15) utilizza la seguente sintassi:

```
SimpleAddKernel<<<1, vector_size>>>(d_A, offset);
```

Questa avvia l'esecuzione della funzione `SimpleAddKernel` sulla GPU definendo uno spazio lineare di `vector_size` thread. La sintassi utilizzata da SYCL si presenta molto più verbosa (righe 20-16 del codice 4.4):

```
q_ct1.submit([&](sycl::handler &cgh) {
  cgh.parallel_for(sycl::nd_range(sycl::range(1, 1, vector_size),
                                  sycl::range(1, 1, vector_size)),
                   [=](sycl::nd_item<3> item_ct1) {
                     SimpleAddKernel(d_A, offset, item_ct1);
                   });
});
```

SYCL introduce un command group e attraverso il suo handler `cgh` è possibile invocare il `parallel_for`. Per prima cosa, esso definisce l'ND-range specificando le dimensioni del global range e del local range: entrambi definiscono uno spazio di $1 \times 1 \times$ `vector_size` elementi, pertanto si deduce che è definito un solo work-group che contiene tutti i work-item dello spazio degli indici. Oltre all'ND-range, il `parallel_for` richiede in input la lambda expression con il codice da eseguire sull'acceleratore. Questa dichiara il parametro `item_ct1` per poter infine avviare la funzione `SimpleAddKernel()`.

Terminata l'esecuzione del kernel, il risultato della computazione è copiato indietro sull'host: la funzione `cudaMemcpy()` è stata tradotta con il corrispondente metodo `memcpy()` definito dalla classe `sycl::queue`. Si noti che nel codice DPC++ è necessaria un'istruzione di sincronizzazione (`wait()`) per garantire l'effettiva terminazione della copia prima di passare all'istruzione successiva.

Il main si conclude con la deallocazione dello spazio di memoria contenente il vettore sul quale si sono effettuate le computazioni e con la stampa del risultato ottenuto. Questa sezione è pressoché identica in entrambi i codici.

Si noti come l'utilizzo della USM obblighi a una gestione esplicita della memoria e della sincronizzazione e non si possa quindi sfruttare il meccanismo dell'application scope.

Risultati del DPCT sul codice di Patatrack

Abbiamo trasformato il codice Pixeltrack Standalone con il Compatibility Tool ottenendo un risultato in linea con l'esempio della sezione precedente. In particolare:

- Le istruzioni di allocazione e spostamento di dati utilizzate nella versione in CUDA sono state convertite utilizzando la Unified Shared Memory di oneAPI. Ad esempio, l'istruzione:

  ```
  cudaCheck(cudaMalloc(&d_in, num_items * sizeof(uint32_t)));
  ```

  è stata tradotta nel corrispondente:

  ```
  d_in = sycl::malloc_device<uint32_t>(num_items, queue.
     get_device());
  ```

- Gli stream CUDA, ovvero le sequenze di operazioni che devono essere eseguite sulla stessa GPU, si sono tradotti in code in-order. Questa conversione ha cambiato significativamente il parallelismo del programma. Infatti, mentre nel codice CUDA l'esecuzione dei kernel è ottimizzata, nel codice prodotto dal DPCT avviene una sincronizzazione al termine di ogni singolo task.

  Nonostante il peggioramento delle performance, SYCL ha apportato anche una semplificazione al codice. Gli stream CUDA non mantengono informazioni inerenti al dispositivo al quale sono associati, per questo è necessario incapsulare tale informazione all'interno di una struttura dati. Le code DPC++, invece, implementano il metodo `get_device()` grazie al quale è sempre possibile risalire al dispositivo sul quale la coda è istanziata. In questo modo ci è stato possibile alleggerire il codice eliminando la struttura del codice CUDA.

Se da un lato il programma è stato semplificato, il codice DPC++ è sicuramente molto meno leggibile della versione in CUDA per quanto riguarda l'invocazione dei kernel: a fronte di una riga di codice, SYCL ne utilizza un numero più elevato.

- Le operazioni di sincronizzazione dei thread del codice di partenza sono state tradotte nelle corrispondenti istruzioni di sincronizzazione in DPC++. Ad esempio, la funzione `__syncthreads()` è stata convertita con `item.barrier()`.

Il Compatibility Tool, tuttavia, non è riuscito a convertire correttamente alcuni aspetti dell'applicazione. Per prima cosa, come è già stato evidenziato precedentemente in questa sezione, siamo dovuti intervenire manualmente sui codici di errore di CUDA.

Un secondo aspetto problematico ha riguardato la gestione degli eventi: le due tecnologie in analisi li implementano in modo sostanzialmente diverso. Un `cudaEvent` è un oggetto che può essere creato al fine di verificare se il flusso di esecuzione ha raggiunto un determinato punto del programma. In CUDA si definisce un evento associandolo ad uno stream, dopodiché lo si può memorizzare all'interno di una struttura dati e verificare in ogni momento del programma se il codice a cui si riferisce è stato eseguito o meno.

In DPC++ un `sycl::event` è un oggetto che può essere utilizzato per sincronizzare trasferimenti di dati, accodamenti di kernel e barriere. La differenza sostanziale con CUDA è che SYCL non permette di creare un evento a priori per monitorare l'esito di una computazione: un SYCL event, infatti, è il risultato della sottomissione di un command group a una coda. Questo comporta che non sia possibile inserire un evento in una struttura dati prima della definizione del command group stesso.

Per ottenere un comportamento simile agli eventi CUDA abbiamo utilizzato oggetti di tipo `std::optional<sycl::event>`: in questo modo è possibile definire un oggetto che può contenere o meno un evento e che è utilizzabile in qualsiasi punto del programma. Di conseguenza, per verificare se la computazione in analisi è stata completata è sufficiente verificare che l'oggetto `std::optional<sycl::event>` contenga

effettivamente un `sycl::event` e poi interrogare quell'evento per avere informazioni sul suo stato.

Per quanto riguarda le istruzioni per la stampa all'interno dei kernel, CUDA implementa la funzione `printf()` tipica del linguaggio C. Nonostante il momento in cui i messaggi sono stampati non sia deterministico, questi comandi possono essere utili per indicare che in un qualche momento un thread ha eseguito il suo codice. Ad esempio:

```
if (threadIdx.x == 0)
  printf("start clusterizer for module %d in block %d\n",
      thisModuleId, blockIdx.x);
```

Il meccanismo utilizzato da DPC++ è più esplicito rispetto a quello offerto da CUDA: per stampare un messaggio all'interno di un kernel occorre istanziare nel suo command group un buffer di tipo `sycl::stream`. Questa definizione richiede di specificare la dimensione massima che il buffer potrà occupare e il numero di caratteri che ciascun work-item vi potrà scrivere. Un esempio è:

```
sycl::stream sycl_stream(64 * 1024, 80, cgh);
```

L'oggetto così definito deve poi essere passato in input alla lambda expression del kernel, che potrà accedervi in scrittura nel seguente modo:

```
if (item.get_local_id(2))
  sycl_stream << "start clusterizer for module " << thisModuleId <<
      " in block " << item.get_local_id(2) << sycl::endl;
```

Le istruzioni di stampa non sono state convertite correttamente dal DPCT, il quale ha segnalato questo problema con un messaggio:

```
/* DPCT1015:10: Output needs adjustment. */
```

Un ultimo aspetto che ha richiesto il nostro intervento riguarda i *warp* di CUDA. Un warp è un gruppo di thread, generalmente 32 nelle schede NVIDIA, il cui lavoro è gestito da un'unica unità di controllo. I thread di un warp eseguono la stessa istruzione nello stesso momento. Questo significa che diversi flussi di controllo eventualmente seguiti da thread diversi all'interno dello stesso warp sono eseguiti sequenzialmente. Per massimizzare le performance bisogna minimizzare queste situazioni.

I warp sono un costrutto di basso livello caratteristico delle schede della NVIDIA; di conseguenza, SYCL deve prevedere una loro astrazione generica che ne mantenga la logica. A questo scopo DPC++ introduce il concetto di *sub-group* come insieme di work-item appartenenti allo stesso work-group che possono comunicare e sincronizzarsi direttamente tra di loro. Nonostante ciò, il Compatibility Tool non è riuscito a convertire correttamente l'utilizzo dei warp nelle corrispondenti istruzioni DPC++, pertanto è stato necessario un nostro intervento manuale.

4.2.3 Il caso concreto dell'algoritmo delle somme prefisse

In questa sezione si riporta un frammento del software Pixeltrack Standalone sia nella versione CUDA, sia nella versione tradotta in DPC++. La traduzione è completa delle correzioni che abbiamo apportato[4].

Il codice da cui è stato estratto il frammento in questione è utilizzato per effettuare dei test di diversa natura, in particolare permette di verificare la correttezza dell'algoritmo delle somme prefisse[5] implementato in CMSSW. Nello specifico, in questa sezione si trovano:

[4] È importante sottolineare che i codici riportati in questa sezione sono stati leggermente riadattati rispetto alle loro versioni originali; lo scopo delle modifiche permette di evidenziare gli elementi importanti per la trattazione di questa tesi mantenendo inalterata la logica del programma.

[5] Il problema delle somme prefisse è una generalizzazione immediata della sommatoria: dati n numeri $a_1, a_2, ..., a_n$, si vogliono calcolare tutte le somme parziali $b_i = \sum_{j=1}^{i} a_j$ per $i = 1, 2, ...n$ [48].

- il codice 4.5, contenente il `main` del programma CUDA;
- il codice 4.7, contenente il kernel da eseguire sulla GPU NVIDIA;
- il codice 4.9, contenente l'algoritmo delle somme prefisse invocato dal kernel CUDA;
- i codici 4.6, 4.8 e 4.10 contengono rispettivamente le traduzioni in DPC++ dei frammenti 4.5, 4.7 e 4.9.

Si parta dall'osservazione del `main`: il codice CUDA risulta essere molto semplice e conciso: si seleziona un dispositivo al quale associare lo stream, si avvia il kernel e si effettua un'operazione di sincronizzazione per garantire la sua effettiva terminazione. Al contrario, il secondo `main` richiede più istruzioni per implementare la stessa logica. Inizialmente si istanzia una coda utilizzando il `default_selector`, dopodiché si crea un command group: al suo interno si trovano le definizioni di uno SYCL stream e di due accessor locali, ovvero le tre strutture dati che devono essere allocate per l'esecuzione del kernel. A questo punto, tramite il `parallel-for`, si può definire l'ND-range e il codice che deve essere parallelizzato. Si noti che lo spazio lineare di CUDA composto da 32 thread è stato convertito ad uno spazio tridimensionale composto da un singolo work-group di 16 thread; inoltre, il `parallel-for` richiede un attributo che imposti la dimensione dei sub-group (maggiori spiegazioni sulla scelta di utilizzare uno spazio e un sub-group di 16 thread invece che da 32 si trovano nella sezione 4.3.2 riguardante le difficoltà riscontrate). Infine si invoca il kernel passando in input non sono l'intero 32 come implementato dal codice CUDA, ma anche l'oggetto `item`, le tre strutture dati che il kernel deve manipolare e la dimensione del sub-group.

```
1 int main() {
2   cms::cudatest::requireDevices();
3
4   testWarpPrefixScan<int><<<1, 32>>>(32);
5   cudaDeviceSynchronize();
```

```
  ...
}
```

Codice 4.5: *Il `main` di un frammento del codice CUDA Pixeltrack Standalone*

```
int main() {
  sycl::default_selector device_selector;
  sycl::queue queue(device_selector);

  queue.submit([&](sycl::handler &cgh) {
    sycl::stream sycl_stream(64 * 1024, 80, cgh);
    sycl::accessor<int, 1, sycl::access::mode::read_write,
      sycl::access::target::local> c_acc(sycl::range(1024),cgh);
    sycl::accessor<int, 1, sycl::access::mode::read_write,
      sycl::access::target::local> co_acc(sycl::range(1024),cgh);

    cgh.parallel_for(sycl::nd_range(sycl::range(1, 1, 16),
                                    sycl::range(1, 1, 16)),
                     [=](sycl::nd_item<3> item)
                         [[cl::intel_reqd_sub_group_size(16)]] {
      testWarpPrefixScan<int>(32, item, sycl_stream,
        c_acc.get_pointer(), co_acc.get_pointer(), 16);
    });
  });
  queue.wait();
  ...
}
```

Codice 4.6: *Traduzione in DPC++ del codice 4.5 scritto in CUDA*

Si passi ora ad osservare l'implementazione dei kernel. Rispetto a quanto trattato nella sezione 4.2.2 si noti che: l'allocazione dei vettori `c` e `co` è stata spostata dal kernel al

command group scope e che tutti gli `assert` sono stati convertiti in blocchi `if-else`. La spiegazione di quest'ultima modifica si trova nella sezione 4.3.2.

```
1  template <typename T>
2  __global__ void testWarpPrefixScan(uint32_t size) {
3    assert(size <= 32);
4    __shared__ T c[1024];
5    __shared__ T co[1024];
6    auto i = threadIdx.x;
7    c[i] = 1;
8    __syncthreads();
9
10   warpPrefixScan(c, co, i, 0xffffffff);
11   __syncthreads();
12
13   assert(1 == c[0]);
14   assert(1 == co[0]);
15   if (i != 0) {
16     if (c[i] != c[i - 1] + 1)
17       printf(format_traits<T>::failed_msg, size, i,
18         blockDim.x, c[i], c[i - 1]);
19     assert(c[i] == c[i - 1] + 1);
20     assert(c[i] == i + 1);
21     assert(c[i] = co[i]);
22   }
23 }
```

Codice 4.7: *Il kernel invocato dal codice 4.5*

```
template <typename T>
void testWarpPrefixScan(uint32_t size, sycl::nd_item<3> item,
    sycl::stream sycl_stream, T *c, T *co, int subgroupSize) {
  if(size > 32){
    sycl_stream << "failed ..." << sycl::endl; return; }

  auto i = item.get_local_id(2);
  c[i] = 1;
  item.barrier();

  warpPrefixScan(c, co, i, item, subgroupSize);
  item.barrier();

  if(1 != c[0]){
    sycl_stream << "failed ..." << sycl::endl; return; }
  if(1 != co[0]){
    sycl_stream << "failed ..." << sycl::endl; return; }
  if (i != 0) {
    if (c[i] != c[i - 1] + 1)
      sycl_stream << "failed ..." << sycl::endl;
    if(c[i] != c[i - 1] + 1){
      sycl_stream << "failed ..." << sycl::endl; return; }
    if(c[i] != i + 1){
      sycl_stream << "failed ..." << sycl::endl; return; }
    if(c[i] != co[i]){
      sycl_stream << "failed ..." << sycl::endl; return; }
  }
}
```

Codice 4.8: *Traduzione in DPC++ del codice 4.7 scritto in CUDA*

Infine si osservi l'algoritmo delle somme prefisse: non si notano differenze importanti tra le due versioni. In particolare, i comandi relativi alla gestione dei CUDA warp sono stati sostituiti con i comandi relativi ai SYCL sub-group. A questo riguardo si noti che la maschera richiesta dalla funzione CUDA `__shfl_up_sync()` non è richiesta dalla corrispondente versione SYCL `shuffle_up()`.

```
template <typename T>
__device__ void __forceinline__ warpPrefixScan(
    T const* __restrict__ ci, T* __restrict__ co,
    uint32_t i, uint32_t mask) {
  auto x = ci[i];
  auto laneId = threadIdx.x & 0x1f;
#pragma unroll
  for (int offset = 1; offset < 32; offset <<= 1) {
    auto y = __shfl_up_sync(mask, x, offset);
    if (laneId >= offset)
      x += y;
  }
  co[i] = x;
}
```

Codice 4.9: *Algoritmo delle somme prefisse implementato in CUDA*

```
template <typename T>
SYCL_EXTERNAL inline __attribute__((always_inline)) void warpPrefixScan(
    T const* __restrict__ ci, T* __restrict__ co, uint32_t i,
    sycl::nd_item<3> item, int subgroup_size) {
  auto x = ci[i];
  auto laneId = item.get_local_id(2) % subgroup_size;
#pragma unroll
  for(unsigned int offset = 1; offset < subgroup_size; offset <<= 1){
```

```
9       auto y = item.get_sub_group().shuffle_up(x, offset);
10      if (laneId >= offset)
11        x += y;
12    }
13    co[i] = x;
14  }
```

Codice 4.10: *Traduzione in DPC++ del codice 4.9 scritto in CUDA*

4.3 Analisi dei risultati

Il processo di conversione da CUDA a SYCL del software Pixeltrack Standalone non è ancora terminato. A fine Novembre 2020 risultano tradotti e funzionanti il framework e gli algoritmi del modulo *digis*. Molti progressi sono stati fatti anche per gli algoritmi del modulo *clusters* il quale, però, non gira ancora correttamente su tutti i dispositivi a nostra disposizione.

In questa sezione si riportano inizialmente i risultati di performance ottenuti con il codice DPC++; segue un'analisi di tali risultati in confronto ai tempi richiesti dalla corrispondente versione nativa eseguita solo sulla CPU. Si noti che, allo stato attuale, le misure presentate non hanno la pretesa di mostrare significativi risultati di performance: la traduzione del codice, infatti, ha richiesto molto più tempo del previsto. Inoltre, non è stato possibile effettuare test di performance comparativi con il codice CUDA di partenza in quanto, oltre a non aver terminato il porting del codice, le architetture utilizzate per i test non sono comparabili con le schede grafiche NVIDIA.

L'obiettivo che ci si è posti è stato quindi di verificare che la trasformazione da CUDA a SYCL fosse possibile, e ad oggi siamo ancora in uno stadio di valutazione della correttezza del software ottenuto. Invece, è stato importante verificare che, senza fare

modifiche al sorgente, il codice sia effettivamente compilabile ed eseguibile su dispositivi eterogenei e su back-end diversi.

Si trova, in conclusione di questa sezione, una descrizione delle difficoltà riscontrate con un commento critico sull'implementazione di SYCL utilizzata in questo progetto.

4.3.1 Analisi delle performance

Abbiamo condotto i test delle performance del codice SYCL sugli algoritmi del modulo *digis*. Questo ha permesso non solo di ottenere i tempi di esecuzione di tale modulo, ma anche di effettuare test sul framework[6].

Abbiamo eseguito il codice sulle architetture fornite da Intel DevCloud[7] [49]; in particolare si sono utilizzate:

- la CPU Intel(R) Xeon(R) E-2176G CPU 3.70GHz con driver OpenCL (versione 2020.11.10.0.05);
- la GPU integrata Gen9 HD Graphics con back-end OpenCL e Level Zero (versione 20.41.18123);

Per esaminare l'effettiva portabilità del software su dispositivi eterogenei abbiamo effettuato test su diversi acceleratori e con diversi back-end; i risultati ottenuti sono riportati in tabella 4.1. In dettaglio, le prove effettuate hanno riguardato inizialmente le performance del modulo *digis* eseguito nella sua versione nativa C++ sulla sola CPU, senza l'utilizzo di acceleratori. Successivamente abbiamo testato la versione DPC++:

- su CPU con back-end OpenCL

[6]Ottenere dei risultati per il solo framework non è molto significativo in quanto, per poter essere testato, esso necessita di eseguire uno o più moduli. Di conseguenza i risultati sulle sue performance possono essere inclusi nelle tempistiche di esecuzione del modulo *digis*.

[7]Intel DevCloud è una piattaforma che mette a disposizione dei programmatori un cluster con gli ultimi modelli hardware prodotti da Intel; questi sono completi delle librerie, dei tool e dei framewrok più popolari e di recente sviluppo, compreso oneAPI.

Tabella 4.1: *Performance di Patatrack Pixel Tracking. I valori riportati, espressi in microsecondi, sono la media di dieci misurazioni effettuate sulle architetture di Intel DevCloud*

dispositivo	**codice CPU nativo**	**CPU**	**GPU**	**GPU**	**SYCL host**	**FPGA emulator**
interfaccia	Single thread	OpenCL	OpenCL	Level Zero	-	OpenCL
work-group	1	9	24	24	1	9
work-item per work-group	1	4096	256	256	1	4096
tempo per evento (ms)	0.45 ± 0.01	17.07 ± 0.03	2.67 ± 0.01	4.75 ± 0.04	3.6 ± 0.2	13.0 ± 0.7

- su GPU con back-end OpenCL
- su GPU con back-end Level Zero
- sul SYCL host device
- sull'emulatore di FPGA

È importante sottolineare che i tempi riportati in tabella sono frutto di una media di dieci rilevazioni. Inoltre, abbiamo deciso di adattare il numero e le dimensioni dei work-group alle capacità massime di ogni acceleratore.

Dalla tabella 4.1 si nota che la versione nativa per CPU è quella che impiega il tempo minore. Questo risultato è dovuto probabilmente a diverse cause. Una prima supposizione che spieghi questi tempi riguarda l'ottimizzazione del codice. In particolare, l'utilizzo delle code in-order comporta che avvenga una sincronizzazione al termine dell'esecuzione di ogni command group; questo causa, con molta probabilità, un peggioramento delle performance.

Inoltre, le dimensioni dei work-group non sono ottimizzate in base all'architettura dell'acceleratore. Questa è una conseguenza intrinseca del problema dell'eterogeneità

del codice: se si dovessero ottimizzare gli algoritmi per ogni architettura sulla quale potrebbero essere eseguiti si perderebbero tutti i vantaggi che si stanno cercando di ottenere con questo lavoro. Tuttavia si potrebbero implementare degli appositi costrutti condizionali che, previa interrogazione del tipo di device selezionato per l'esecuzione dei kernel, impostino di conseguenza le dimensioni dell'ND-range.

Infine, i back-end delle architetture forniscono spazi di indici non molto estesi. Ad esempio, la limitazione a 256 thread per work-group imposta dalla GPU integrata è sicuramente inferiore alle capacità delle schede NVIDIA.

Nonostante i risultati di performance ottenuti con il codice SYCL siano peggiori di 1-2 ordini di grandezza rispetto alla sua versione base che non impiega acceleratori, siamo riusciti a dimostrare che grazie a SYCL è possibile realizzare un unico codice che possa essere eseguito su architetture eterogenee anche molto diverse tra loro. In particolare, si evidenzia una differenza di un ordine di grandezza fra il codice SYCL eseguito su CPU e sull'emulatore di FPGA rispetto allo stesso codice eseguito su GPU e sul SYCL host device. Quello che ci aspettiamo, però, è che se si ottimizza il codice seguendo i punti riportati nel capitolo successivo, le performance miglioreranno su tutti i dispositivi, e le differenze fra le prestazioni che oggi si ottengono su dispositivi diversi andranno ad assottigliarsi.

4.3.2 Difficoltà riscontrate

Nel procedimento di traduzione del codice abbiamo incontrato diverse difficoltà. La maggior parte di esse sono dovute al fatto che oneAPI e DPC++ sono delle tecnologie ancora in fase di sviluppo. Questo ha avuto alcune importanti conseguenze. Per prima cosa, nel momento in cui è iniziato il progetto, la documentazione di oneAPI era quasi del tutto assente: erano disponibili solamente alcune pagine Web piuttosto scarne e la bozza dei primi quattro capitoli di un manuale su DPC++ [36] in preparazione. Il documento con le specifiche di SYCL [32] è stato la principale fonte di informazione;

esso, però, non contiene indicazioni su oneAPI, né tanto meno esempi o spiegazioni inerenti alle tecniche di programmazione per questo standard. Anche sui principali blog di programmazione non si trovano ancora informazioni su problemi comuni o discussioni sul linguaggio DPC++.

La situazione è tuttavia migliorata con il passare del tempo: Intel, infatti, si sta impegnando a estendere la documentazione online di oneAPI. Molto più significativa è, invece, la recentissima pubblicazione del manuale su DPC++ succitato. Questo agevolerà sicuramente i programmatori che inizieranno a misurarsi con questa nuova tecnologia.

Un secondo aspetto significativo riguarda il processo di sviluppo di oneAPI. Come ho accennato nelle sezioni precedenti, questa è una tecnologia estremamente nuova e non ancora completamente stabile. Nel momento in cui sono iniziate le prime fasi della conversione del software, l'unica sua versione disponibile era la Beta7. Per ogni implementazione rilasciata, è stato eseguito l'aggiornamento del sistema sul quale si è sviluppata la versione SYCL di Pixeltrack Standalone. Se questo rilascio frequente di nuove versioni ha permesso di risolvere velocemente i problemi riscontrati nello sviluppo, ha quasi sempre introdotto ulteriori bug e, conseguentemente, ha spesso comportato la revisione di alcune parti del codice.

Inoltre, durante il processo di conversione, abbiamo individuato degli errori in alcune istruzioni di oneAPI; in questi casi abbiamo notificato il problema direttamente agli sviluppatori di Intel. In alcune situazioni essi sono stati in grado di fornirci una possibile soluzione che ci permettesse di risolvere il problema.

Un esempio che evidenzia queste difficoltà è dato dall'istruzione che definisce la dimensione dei sub-group. Tale istruzione è evoluta nel corso delle successive versioni e, grazie al dialogo con gli sviluppatori di DPC++, è stato possibile aggiornare il codice con la sua versione più corretta.

Di seguito si trova un elenco dei principali problemi riscontrati durante lo sviluppo del software:

- Per quanto riguarda il Compatibility Tool, la difficoltà maggiore è stata rilevata nel momento in cui il processo di conversione si arresta in modo anomalo: in questi casi, i messaggi di errore restituiti non evidenziano in modo chiaro quale problema abbia causato la sua terminazione; anche la documentazione online è piuttosto scarna per quanto concerne questo aspetto.

- Se si vuole eseguire un kernel cercando di sfruttare appieno il parallelismo di un'architettura, è necessario creare un ND-range che contenga il massimo numero di work-item possibili. Questa operazione, però, ha comportato diversi problemi: prima di tutto, il numero massimo di work-item e il numero massimo work-item per work-group dipendono strettamente dall'hardware dell'acceleratore e questo potrebbe rompere l'eterogeneità del codice. SYCL, tuttavia, fornisce la funzione template `get_info()` che permette di determinare a runtime questi valori, ad esempio:

```
auto maxWorkItemSizes = queue.get_device()
    .get_info<sycl::info::device::max_work_item_sizes>();
```

  In questo modo è possibile parametrizzare un'applicazione in modo che si creino ND-range di dimensioni variabili in base all'acceleratore selezionato a runtime e alle caratteristiche del kernel che si vuole eseguire. Nonostante ciò, ci si è accorti che il numero massimo di work-item per work-group della CPU utilizzata per i test non è 8192 come indica tale funzione, ma 4096. Pertanto, si è reso necessario modificare ogni sorgente che utilizza questa informazione affinché adoperi il numero corretto.

- Se si osservano i codici 4.7 e 4.8, si nota che gli `assert` del codice CUDA sono stati convertiti in strutture di controllo `if-else` nel codice DPC++. Il motivo di questa scelta è stato dettato dal fatto che il linguaggio dell'Intel ancora non supporta gli `assert` all'interno delle sue applicazioni. Questo problema è stato

riportato ai suoi sviluppatori, ma al momento della scrittura di questa tesi non è ancora stato risolto.

- L'algoritmo delle somme prefisse riportato nella sezione precedente richiede l'utilizzo dei sub-group. Come per il numero massimo di work-item per work-group, anche la dimensione dei sub-group è strettamente dipendente dall'architettura del dispositivo; di conseguenza, si potrebbe parametrizzare tale valore al fine di stabilirlo a runtime in base al dispositivo selezionato per l'esecuzione dei kernel. Il codice seguente mostra come ottenere tale valore grazie all'utilizzo della funzione `get_info()`:

```
auto subgroupSizes = queue.get_device()
    .get_info<sycl::info::device::sub_group_sizes>();
int maxSubgroupSizes =
    *std::max_element(std::begin(subgroupSizes),
                      std::end(subgroupSizes));
```

Questa soluzione, però, non è così semplice come ci si potrebbe aspettare: la dimensione dei sub-group è utilizzata come parametro dell'attributo della lambda expression che definisce la kernel function (riga 15 del codice 4.6)

```
[[cl::intel_reqd_sub_group_size(<sub-group-dim>)]]
```

L'implementazione attuale richiede che tale valore debba essere noto a compile time. Di conseguenza il suo beneficio per lo sviluppo di codici eterogenei è al momento limitato.

Una soluzione a questo problema è stata individuata durante il *oneAPI Developer Summit 2020* [50]: è possibile implementare delle classi template con dimensio-

ne dei sub-group esplicita. Così facendo, a seconda dell'acceleratore in uso, sarà possibile invocare la classe template con il corretto valore.

- Gli algoritmi del modulo *clusters* non segnalano problemi di compilazione, ma la loro esecuzione blocca completamente la GPU. Per individuare il problema si è provato ad utilizzare il debugger GDB per oneAPI; questo, tuttavia, è ancora in fase di sviluppo e non ha permesso di trovare informazioni a riguardo. È in atto un dialogo con l'Intel per cercare di risolvere il problema.

Questa sezione si conclude con un elenco di alcuni aspetti di oneAPI che non si sono trovati particolarmente comodi:

- I nomi delle variabili utilizzate dal Compatibility Tool sono piuttosto lunghi: ogni nome, infatti, termina con il suffisso `_ct1`. Questo, se si somma alla generale verbosità di DPC++, produce codici poco leggibili.

- La sintassi della definizione dell'ND-range e dell'invocazione dei kernel è estremamente verbosa, soprattutto in confronto a quella utilizzata da CUDA. In queste parti il codice è decisamente poco leggibile. Oltre a ciò, questa verbosità non aiuta il programmatore nella corretta implementazione del codice rendendo più facile la possibilità di commettere errori.

- Gli indici dell'ND-range seguono l'ordine inverso da 2 a 0 utilizzato da OpenCL; questo ordine non è intuitivo e sulla documentazione non si trovano molte informazioni a riguardo.

- La conversione degli stream CUDA in code in-order non è sempre la soluzione migliore. Questo comportamento, però, non è specificato dalla documentazione di oneAPI, né, tanto meno, dall'output di DPCT. Probabilmente una nota a riguardo aiuterebbe i programmatori nella verifica dell'ottimizzazione dell'ordine di esecuzione dei kernel.

Conclusioni e sviluppi futuri

Per aumentare il potenziale della ricerca nell'ambito della fisica delle particelle, il Large Hadron Collider del CERN sarà oggetto nel prossimo futuro di un imponente aggiornamento che sta richiedendo uno sforza globale da parte di tutte le agenzie che lo finanziano. Non sono solo le componenti meccaniche che necessitano di sostituzioni e miglioramenti; anche le risorse di calcolo e il software hanno bisogno di essere rivisti in modo da supportare il considerevolmente aumento della quantità di dati prodotti dagli esperimenti in funzione all'acceleratore.

I gruppi di ricerca e sviluppo dell'esperimento CMS si stanno occupando di studiare nuove strategie che permettano di aumentare la potenza computazionale a disposizione dell'esperimento mantenendo sostanzialmente invariato il budget dedicato a questo aspetto. Dagli studi emerge chiaramente che i centri di calcolo dei prossimi anni faranno un largo utilizzo di acceleratori, quali GPU e FPGA. Di conseguenza il software deve essere aggiornato al fine di poter essere compilato ed eseguito su un ampio spettro di architetture eterogenee. A questo scopo sono state individuate alcune tecnologie promettenti: Kokkos, Alpaka, SYCL/oneAPI. È stata quindi realizzata una versione semplificata del software utilizzato da CMS, *Pixeltrack Standalone*, che si presti agli studi richiesti. Pixeltrack Standalone può essere implementato utilizzando ciascuna delle tecnologie in esame: in questo modo è possibile effettuare un confronto delle loro performance su diversi dispositivi. L'obiettivo è quello di valutare queste tecnologie al fine di trovare quella che meglio si adatta alle esigenze di CMS.

Con il lavoro di questa tesi ho avuto modo di approfondire SYCL, il nuovo modello di programmazione astratto per dispositivi eterogenei definito da Khronos Group. Ho studiato sia la specifica di questo standard, sia alcune sue implementazioni, concentrandomi in particolar modo su quella sviluppata da Intel: oneAPI DPC++. Il progetto di tesi ha previsto poi l'implementazione di Pixeltrack Standalone in DPC++, partendo dalla sua versione in CUDA. Lo sviluppo è ancora in corso: fino ad oggi sono stati implementati solo il framework, che costituisce la spina dorsale dell'applicazione, e due dei numerosi moduli che compongono l'applicazione stessa; tuttavia, il lavoro fin qui svolto è già sufficiente per poter esprimere un parere su questa tecnologia, anche alla luce delle esigenze di CMS.

SYCL è sicuramente una tecnologia molto promettente: l'idea di avere un modello di programmazione che permetta di implementare in un unico sorgente un programma da eseguire su un vasto insieme di dispositivi eterogenei è sicuramente molto interessante per diversi ambiti dell'ICT. A supporto di ciò vi è l'interesse che la comunità scientifica sta dimostrando nei confronti di questa tecnologia in previsione dell'aumento della potenza computazionale necessaria nel prossimo futuro.

La specifica di SYCL, però, deve essere supportata da una sua valida implementazione. Nel 2020 si sono ottenuti importanti risultati grazie all'ecosistema oneAPI: la qualità dell'implementazione del compilatore DPC++ è migliorata considerevolmente nei diversi rilasci del software e i progressi sono stati evidenti. Nonostante ciò, molta strada deve essere ancora percorsa per avere un prodotto stabile: è necessario estendere il supporto ad un numero sempre più vasto di acceleratori e occorre migliorare le prestazioni che attualmente si riescono ad ottenere con questa tecnologia. Inoltre, DPC++ può essere ulteriormente perfezionato: alcuni errori non sono ancora stati risolti e l'insieme delle sue funzionalità può essere esteso.

Per quanto riguarda il possibile impiego dei prodotti che fanno parte di oneAPI nel software di CMS, è ancora troppo presto per dare giudizi definitivi. I risultati di perfor-

mance ottenuti con il codice implementato per questa tesi non sono in linea nemmeno con la versione base che non impiega acceleratori. Tuttavia, non è stato fatto alcun lavoro di ottimizzazione e ci si è largamente affidati al codice prodotto automaticamente a seguito della conversione dal codice CUDA. Entrambi i tool utilizzati, DPCT per la conversione e DPC++ per la compilazione, saranno sicuramente oggetto di miglioramenti nel prossimo futuro.

Gli sviluppi del progetto esposto in questa tesi sono diversi. Innanzitutto è necessario terminare la scrittura della versione SYCL dell'applicazione Pixeltrack Standalone; questo permetterà di avere una visione più ampia riguardo l'aderenza di questa tecnologia con il codice di CMS. Allo stesso tempo è importante capire come ottimizzare il codice prodotto fino a questo momento. Sono quattro i punti principali sui quali si dovrà lavorare:

- È importante individuare la dimensione ottimale dello spazio degli indici per ogni dispositivo sul quale si esegue il codice. Investire su questo aspetto, se da un lato comporterebbe una complicazione dell'applicazione, dall'altro garantirebbe di ottenere prestazioni ottimizzate per ogni acceleratore in uso.

- Più specificatamente, è importante che il codice individui la corretta definizione della dimensione dei sub-group. Attualmente questo valore è impostato staticamente per un limite del compilatore, ma sarebbe sicuramente meglio parametrizzarlo in modo dinamico; in subordine si può cercare di aggirare il problema sfruttando qualche tecnica di template metaprogramming.

- Sarebbe interessante sostituire le code in-order con le code out-of-order, adottando opportune tecniche di sincronizzazione. Questa modifica permetterebbe di eliminare le costose sincronizzazioni con la CPU che attualmente si verificano al termine di ogni kernel. Questo aspetto è probabilmente la causa che ha maggiore impatto sulle prestazioni ottenute.

- Quando DPC++ lo consentirà, sarà opportuno riportare le strutture di controllo `if-else` agli `assert` corrispondenti.

Una volta terminata la conversione in SYCL di tutti i moduli dell'applicazione Pixeltrack Standalone e dopo aver apportato le ottimizzazioni sopra riportate al codice ottenuto, sarà possibile effettuare delle misure di prestazione più significative. In particolare, l'applicazione dovrà essere valutata su diversi acceleratori e su diversi back-end in modo da avere delle misure sulle quali effettuare analisi di portabilità delle prestazioni. A quel punto sarà possibile confrontare i risultati ottenuti dalla versione SYCL del software con quelli rilevati utilizzando Kokkos e Alpaka. Questo sarà sicuramente un importante fattore che permetterà di decidere su quale tecnologia fare affidamento per il Run4 di HL-LHC.

Elenco delle figure

Ringraziamenti

Sono diverse le persone che hanno contribuito a questa tesi e voglio utilizzare questo spazio per ringraziarle una ad una.

Primo fra tutti ringrazio Francesco Giacomini, senza il quale questo elaborato non si sarebbe concretizzato. In lui ho trovato una guida e una fonte indispensabile di consigli, sia per il mio percorso accademico, sia (si spera) per quello che ne seguirà.

È doveroso dedicare un ringraziamento anche al mio relatore, Renzo Davoli, per la disponibilità che ha dimostrato durante tutto il percorso che ha portato alla stesura di questa tesi.

Ringrazio poi Andrea Bocci e Felice Pantaleo che mi hanno accolta nel mondo del CERN nonostante la distanza fisica che ci ha separato. Mi hanno fatta sentire fin da subito parte del loro gruppo e mi hanno trasmesso una grande passione per il lavoro che fanno. Spero che prima o poi ci sarà l'occasione di incontrarsi dal vivo. Devo ringraziare Andrea anche per la pazienza che ha avuto nell'aiutarmi a preparare tutti i talk che mi ha "proposto". Nonostante il mio inglese molto traballante, grazie al suo aiuto ne sono uscita meglio di quanto potessi sperare.

Nei ringraziamenti non può mancare Viola Cavallini, mia compagna di viaggio nell'esperienza al CERN. Mi ritengo molto fortunata di aver avuto la possibilità di condividere con qualcuno questa avventura, soprattutto con una persona che sono riuscita ad incontrare e conoscere personalmente.

Un grande ringraziamento anche ad Anastasiia Lazuka, Andrew Purcell, Kristina

Gunne e a tutti i membri del CERN openlab. Grazie a loro perché mi hanno guidato nell'esperienza della Summer School e perché, ancora oggi, stanno assistendo con molta dedizione me e gli altri ragazzi nel progetto Intel oneAPI.

Ringrazio poi Russel Beutler e Jaclyn Jones di Intel, che dagli Stati Uniti hanno coordinato egregiamente il progetto di social media marketing che sta completando l'esperienza estiva. Conseguentemente, ringrazio anche Alex Burwell, Jesse Radonsky, Bob Duffy e tutto lo staff di Caffelli per i consigli e gli insegnamenti che mi hanno trasmesso riguardo la comunicazione sui social.

E ora passo alla mia famiglia: un enorme grazie ai miei genitori perché, durante il mio percorso di studi, non mi hanno fatto mancare mai il loro supporto, mi hanno incoraggiato nei momenti difficili e hanno valorizzato i miei successi con il loro entusiasmo.

Grazie anche a Silvia: con i racconti delle sue storie e delle sue disavventure riesce sempre a farmi ridere e a farmi sentire una persona fortunata; so che posso sempre contare su di lei nei momenti di difficoltà. E grazie a Pietro perché mi fa sentire importante e con la sua apparente (non sempre) ingenuità rende più leggere le mie giornate.

Concludo la carrellata di famigliari con i miei quattro nonni. A loro va un enorme ringraziamento per tutto quello che mi hanno insegnato e perché sono sempre i primi che si preoccupano per il mio futuro. So che siete fieri di me e io sono molto fiera di avere dei nonni come voi.

E per ultimo, ma non per importanza, ringrazio Nicola. Senza di lui non sarei mai arrivata in fondo a questo traguardo: con il suo aiuto sto imparando a pormi piccoli obbiettivi quotidiani che mi permettono di affrontare le giornate con più serenità. Un enorme grazie perché riesci sempre a vedere, e a farmi vedere, la parte migliore di me.

Bibliografia

[1] Ronan Keryell, Maria Rovatsou e Lee Howes, cur. *SYCL Specification - Generic heterogeneous computing for modern C++*. provisional version, revision 1. Khronos SYCL Working Group. Giu. 2020.

[2] *SYCL*. 2020. URL: https://www.sycl.tech/.

[3] *Programming languages — C++ ISO/IEC 14882:2017*. Mar. 2017. URL: https://wg21.link/std17.

[4] *Standalone Patatrack pixel tracking*. 2020. URL: https://github.com/cms-patatrack/pixeltrack-standalone/.

[5] *CERN - Accelerating science*. CERN. 2020. URL: https://home.cern/.

[6] David Halliday, Robert Resnick e Jearl Walker. *Fondamenti di Fisica - Meccanica e Termologia*. 6ª ed. Milano: Ambrosiana, 2006.

[7] David Halliday, Robert Resnick e Jearl Walker. *Fondamenti di Fisica - Elettrologia, magnetismo, ottica*. 6ª ed. Milano: Ambrosiana, 2006.

[8] *CERN openlab - online education activities*. CERN. 2020. URL: https://openlab.cern/online-learning.

[9] *CMS experiment*. CERN. 2014. URL: http://cms.web.cern.ch/.

[10] *HiLumi - HL-LHC project*. CERN. 2020. URL: https://hilumilhc.web.cern.ch/.

[11] *Longer term LHC schedule*. CERN. 2020. URL: https://lhc-commissioning.web.cern.ch/schedule/HL-LHC-plots.htm.

[12] *CMSSW on GitHub*. 2020. URL: https://github.com/cms-sw/cmssw/.

[13] *Intel oneAPI Threading Building Blocks*. Intel Corporation. 2020. URL: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onetbb.html.

[14] Andrea Bocci, Matti Kortelainen e Felice Pantaleo for the CMS collaboration. «the CMS Patatrack reconstruction as an Intel oneAPI use case». In: *Intel OneAPI Kick-off meeting*. CERN, Svizzera, ott. 2020.

[15] Tommaso Boccali. «CMS Software and Offline preparation for future runs». In: *Journal of Physics: Conference Series* 1525 (apr. 2020), p. 012037. DOI: 10.1088/1742-6596/1525/1/012037.

[16] Akanksha Ahuja, Sharad Agarwal e David Lange for the CMS Collaboration. «Growth and Evolution CMS Offline Computing from Run 1 to HL-LHC». In: *ICHEP 2020*. CERN, Svizzera, lug. 2020.

[17] *CUDA Zone*. NVIDIA Corporation. 2020. URL: https://developer.nvidia.com/cuda-zone.

[18] *AMD ROCm Platform*. Advanced Micro Devices. 2020. URL: https://rocmdocs.amd.com/en/latest/.

[19] *oneAPI programming model*. 2020. URL: https://www.oneapi.com/.

[20] *OpenCL Overview - the Khronos group*. Khronos Group Inc. 2020. URL: https://www.khronos.org/opencl/.

[21] «IEEE Standard for Verilog Hardware Description Language». In: *IEEE Std 1364-2005* (apr. 2006).

[22] Erik Zenker et al. «Alpaka - An Abstraction Library for Parallel Kernel Acceleration». In: IEEE Computer Society, mag. 2016. arXiv: 1602.08477. URL: http://arxiv.org/abs/1602.08477.

[23] *alpaka - Abstraction Library for Parallel Kernel Acceleration*. 2020. URL: https://github.com/alpaka-group/alpaka.

[24] *AMD - HIP Documentation*. Advanced Micro Devices. 2020. URL: https://rocmdocs.amd.com/en/latest/Programming_Guides/Programming-Guides.html.

[25] *OpenMP - The OpenMP API specification for parallel programming*. 2019. URL: https://www.openmp.org/.

[26] *cupla - C++ User interface for the Platform independent Library Alpaka*. 2020. URL: https://github.com/alpaka-group/cupla.

[27] *Sandia National Laboratories*. National Technology e Engineering Solutions of Sandia. 2020. URL: https://www.sandia.gov/about/index.html.

[28] H. Carter Edwards, Christian R. Trott e Daniel Sunderland. «Kokkos: Enabling manycore performance portability through polymorphic memory access patterns». In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202 –3216. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003. URL: http://www.sciencedirect.com/science/article/pii/S0743731514001257.

[29] *Kokkos: Core Libraries*. 2020. URL: https://github.com/kokkos/kokkos.

[30] *Khronos Group - connecting software to silicon*. Khronos Group Inc. 2020. URL: https://www.khronos.org/.

[31] *Direction for ISO C++*. P0939. Prague, gen. 2020.

[32] Ronan Keryell, Maria Rovatsou e Lee Howes, cur. *SYCL Specification - SYCL integrates OpenCL devices with modern C++*. Version 1.2.1, revision 7. Khronos SYCL Working Group. Apr. 2020.

[33] *ComputeCpp Community Edition*. Codeplay Software. 2020. URL: https://developer.codeplay.com/products/computecpp/ce/home.

[34] Codeplay. *SYCL Academy*. 2020. URL: https://github.com/codeplaysoftware/syclacademy.

[35] *IWOCL and SYCLcon*. 2020. URL: https://www.iwocl.org/.

[36] James Reinders et al. *Data Parallel C++ - Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. by Intel Corporation. Apres Open, 2021.

[37] *Codeplay Developer Website*. Codeplay Software. 2020. URL: https://developer.codeplay.com/.

[38] *triSYCL*. 2020. URL: https://github.com/triSYCL/triSYCL.

[39] *The LLVM Compiler Infrastructure*. 2020. URL: http://llvm.org/.

[40] Aksel Alpay e Vincent Heuveline. «SYCL beyond OpenCL: The Architecture, Current State and Future Direction of HipSYCL». In: *Proceedings of the International Workshop on OpenCL*. IWOCL '20. Munich, Germany: Association for Computing Machinery, 2020. ISBN: 9781450375313. DOI: 10.1145/3388333.3388658. URL: https://doi.org/10.1145/3388333.3388658.

[41] *hipSYCL - a SYCL implementation for CPUs and GPUs*. 2020. URL: https://github.com/illuhad/hipSYCL.

[42] *Intel oneAPI Toolkits - Intel Developer Zone*. Intel Corporation. 2020. URL: https://software.intel.com/content/www/us/en/develop/tools/oneapi.html.

[43] *Intel Project for LLVM technology*. 2020. URL: https://github.com/intel/llvm.

[44] *Fermilab*. Fermi Research Alliance. 2020. URL: https://www.fnal.gov/.

[45] Andrea Bocci et al. *Heterogeneous reconstruction of tracks and primary vertices with the CMS pixel tracker*. 2020. arXiv: 2008.13461 [physics.ins-det].

[46] *NVIDIA T4 - Tensor Core GPU*. NVIDIA Corporation. 2020. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf.

[47] *Intel DPC++ Compatibility Tool*. Intel Corporation. 2020. URL: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/dpc-compatibility-tool.html.

[48] Alan A. Bertossi. *Algoritmi Paralleli - Sincroni, Concorrenti, Distribuiti*. Pitagora Editrice Bologna, 2009, p. 29.

[49] *Intel DevCloud: A Development Sandbox for Data Center to Edge Workloads*. 2020. URL: https://software.intel.com/content/www/us/en/develop/tools/devcloud.html.

[50] Sujata Tiberwala et al. «oneAPI Developer Summit 2020». In: Intel Corporation. Nov. 2020. URL: https://www.oneapi.com/events/devcon2020/.