Documentazione Progetto

Maione Simone, Rossi Michela
Calcolo Parallelo e Distribuito

Indice

1 Obiettivo del progetto

Implementare e valutare algoritmi di calcolo parallelo su GPU e CPU eterogenee utilizzando gli ambienti SYCL e Vulkan e confrontarli con approccio OpenMp.

2 Obiettivo esteso e chiarimento

Implementare algoritmi di calcolo parallelo (come gaussian blur e regola del 30) in Vulkan e Sycl e valutarne performance, limiti e opportunità su GPU Nvidia, Intel e AMD.

3 Difficoltà incontrate

3.1 Utilizzo dei linguaggi `Vulkan` e `Sycl`

La prima difficoltà riscontrata è legata al fatto che, prima di iniziare questo progetto, non avevamo mai utilizzato i linguaggi richiesti. Quindi, come prima cosa, abbiamo cercato e consultato le relative documentazioni Vulkan, Sycl.

Ci siamo dunque informati sulle librerie da scaricare per utilizzare entrambi i linguaggi. Per quanto riguarda Vulkan, è stato necessario installare `glslangvalidator`, cioè il compilatore per gli `shader`[1] Vulkan.

Sycl invece richiedeva l'installazione di un compilatore; scelta che è ricaduta su una soluzione open source `AdaptiveCpp`.

Seguendo i passaggi presenti nella guida, abbiamo capito che era necessario installare la libreria `boost` con i moduli `context` e `fiber`.

Dopo averli scaricati, siamo riusciti ad installare correttamente anche `AdaptiveCpp` sulle tre macchine (*hotmetal*, *veryhotmetal* e *flamausim*); tuttavia, avendo fatto delle prove preliminari ci siamo resi conto che sulla macchina *veryhotmetal*, al momento dell'esecuzione di un codice di prova, venivano stampati degli errori riguardo il mancato

[1] Uno shader in Vulkan è un programma che viene eseguito sulla GPU; ne esistono diversi tipi, ma nel nostro caso è stato necessario usare un *compute shader*, il cui compito è quello di eseguire calcoli massivamente paralleli.

collegamento tra il codice e *ROCm*[2]. Dopo aver trascorso un paio di giorni a risolvere il problema, abbiamo optato per la disinstallazione e la conseguente reinstallazione di `AdaptiveCpp`, specificando manualmente tutti i percorsi delle librerie necessarie.

A questo punto dopo aver eseguito diversi test e aver verificato che tutto funzionasse correttamente, abbiamo iniziato con l'implementazione dei vari codici.

4 Algoritmi implementati

Per questo progetto abbiamo scelto due algoritmi tipici del calcolo parallelo, ovvero il Gaussian Blur e l'automa cellulare Regola del 30.

4.1 Gaussian blur

La sfocatura gaussiana è un tipo di sfocatura molto utilizzata nell'ambito della manipolazione delle immagini. Le sfocature, così come altri effetti, vengono applicati tramite una *matrice di convoluzione* (o *kernel*) che viene applicata ad ogni pixel dell'immagine, agendo come uno stencil e seguendo la funzione gaussiana bidimensionale

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-} \frac{x^2 + y^2}{2\sigma^2}.$$

4.2 Regola del 30

La regola del 30 è un automa cellulare elementare, cioè un modello matematico usato per descrivere l'evoluzione di sistemi complessi discreti; in particolare, per la regola del 30, si tratta di un vettore unidimensionale le cui celle possono essere valorizzate solo con 0 o 1 (dove 0 indica lo stato di "*morte*" e 1 lo stato di "*vita*") e l'evoluzione di ogni cella dipende dallo stato delle celle adiacenti; la regola del 30 evolve nel seguente modo:

modello attuale	111	110	101	100	011	010	001	000
nuovo stato per la cella centrale	0	0	0	1	1	1	1	0

[2] *ROCm* è il software di AMD progettato per sfruttare la potenza delle schede video AMD per fare calcoli complessi.

5 Linguaggi utilizzati

5.1 Vulkan

Vulkan è una API progettata per il calcolo parallelo ad alte prestazioni su GPU. I programmi Vulkan si compongono di due parti fondamentali:

- Un `main` scritto in C++ che serve principalmente ad inizializzare tutte le variabili necessarie al calcolo, caricare immagini di input, calcolare il kernel (per gaussian blur), inizializzare il contesto Vulkan, allocare i buffer di memoria;
- Uno `shader`, scritto in GLSL[3]; noi abbiamo utilizzato un *compute shader*, che si occupa di eseguire i calcoli veri e propri su GPU dopo aver ricevuto i dati necessari.

L'inizializzazione di Vulkan è particolarmente lunga e verbosa ma, con la relativa guida, tutorial ed esempi, siamo riusciti ad implementare il codice che applica il gaussian blur ad una qualsiasi immagine in input e, successivamente, anche il codice per la regola del 30. Per l'importazione e la gestione delle immagini abbiamo utilizzato la libreria `stb_image` la cui installazione è molto semplice poiché consiste di un solo file di intestazione che deve essere incluso direttamente nel progetto.

Il file `main` è sostanzialmente identico sia per gaussian blur che per regola del 30. La differenza principale sta nello shader, che è il vero cuore del programma e che abbiamo implementato prendendo spunto da codici di esempio già esistenti, senza troppe difficoltà.

5.2 Sycl

Sycl è un modello di programmazione e astrazione ad alto livello basato su C++. A differenza di approcci come Vulkan o OpenCL, che richiedono una separazione tra il codice host e il codice degli shader, Sycl adotta l'approccio *single-source*. Questo significa che il codice per la CPU e quello per la GPU sono scritti nello stesso file C++. Abbiamo quindi replicato quanto fatto per Vulkan in Sycl. Una cosa interessante da notare è che,

[3]GLSL è un linguaggio di programmazione ad alto livello per gli shader basato sul C.

a parità di implementazione, il codice Sycl è molto meno verboso di quello Vulkan e di conseguenza più breve.

6 Errori riscontrati

A questo punto avevamo delle implementazioni di gaussian blur e regola del 30 sia in Vulkan che in Sycl funzionanti su tutte e tre le macchine. Abbiamo perciò iniziato a svolgere i primi test di performance con misurazione del tempo di calcolo.

6.1 Regola del 30

Per quanto riguarda regola del 30 non abbiamo avuto grandi problemi se non per il fatto che, sia in Vulkan che in Sycl, poiché il codice produce come output l'immagine che rappresenta l'evoluzione dell'automa, quando il numero di iterazioni (che corrispondono all'altezza dell'immagine) è troppo grande, il programma non riesce a produrla e va in crash. Questo problema è dovuto forse al modo in cui salviamo le immagini, ovvero utilizzando la libreria `stb_image`. Probabilmente salvando le immagini in un altro modo, tale problema non si sarebbe presentato (non ci siamo concentrati su di esso perché non ci sembrava particolarmente rilevante in quanto dai test effettuati il codice funziona correttamente fino a un numero sufficientemente grande di iterazioni). Per questo motivo abbiamo modificato il codice aggiungendo un controllo che evita di creare un'immagine quando il numero di iterazioni è troppo alto. In questo modo il programma funziona con dimensioni anche molto grandi senza dare problemi, arrivando a termine in pochi secondi.

Un'altra limitazione dell'attuale implementazione Vulkan della regola 30 è che, con dimensioni di input molto grandi (come ad esempio `65536` con `32768` iterazioni), il programma va in crash. Questo problema è quasi certamente legato al tipo di dato utilizzato per gestire le dimensioni del vettore che, in questo caso, consiste di interi a 32 bit (`uint_32`) e, con dimensioni così elevate, si eccede il limite massimo rappresentabile. Un modo per risolvere il problema sarebbe cambiare il tipo di dato utilizzato (passando a interi a 64

bit), tuttavia non abbiamo attuato questa correzione poiché, per come è strutturato il nostro programma, avrebbe richiesto modifiche sostanziali in diverse parti del codice.

6.2 Gaussian blur

Per gaussian blur abbiamo riscontrato più problemi. In particolare, in Vulkan, con dimensioni del kernel fino a `501 × 501` il codice funziona producendo un'immagine sfocata correttamente in qualche secondo. Oltre questa dimensione, il codice arriva a termine senza produrre messaggi di errore, ma l'immagine di output presenta delle grandi zone di pixel vuoti. Dopo aver provato per diverso tempo a cercare di capire quale fosse la causa, abbiamo implementato una seconda versione del codice che sfrutta la separabilità del kernel gaussiano[4], il che consente di applicare il kernel all'immagine non come una matrice di dimensione $n \times n$ ma come due matrici $n \times 1$ e $1 \times n$. Questa modifica complica un po' la scrittura del codice perché lo shader deve essere invocato due volte per ottenere l'immagine correttamente sfocata ma, in questo modo, non solo il codice funziona con dimensioni nettamente superiori a `501 × 501`, ma l'esecuzione del codice viene velocizzata moltissimo.

Per quanto riguarda Sycl, il problema era simile; infatti, con dimensioni superiori a `501 × 501` il codice non arriva a termine ma, anche stavolta, non produce alcun tipo di errore. Allo stesso modo, sfruttando la separabilità del kernel gaussiano, il problema è stato risolto e, come per Vulkan, il codice arriva a termine anche con dimensioni molto maggiori di `501 × 501` in pochi secondi.

6.3 Cause e soluzioni della prima versione di gaussian blur

Nonostante a questo punto avessimo risolto i problemi legati alla prima implementazione di gaussian blur, abbiamo comunque investigato le cause e le possibili soluzioni alternative al problema. Questo perché la risoluzione del problema consisteva nello sfruttare una caratteristica intrinseca del kernel gaussiano (separabilità); tale caratteristica è presente

[4]Un kernel si dice separabile se la matrice che lo rappresenta può essere scomposta come prodotto di un vettore colonna e un vettore riga.

anche in altri tipi di kernel, ma non in tutti. L'obiettivo era quindi identificare strategie di ottimizzazione universali, valide per qualsiasi algoritmo che presenti criticità simili. Per approfondire la questione, abbiamo avviato una discussione sui canali Reddit specializzati in Sycl e Vulkan, ricevendo molti riscontri da altri utenti.

6.3.1 Sycl

Di seguito il nostro post su reddit sul canale Sycl.

SYCL (AdaptiveCpp) Kernel hangs indefinitely with large kernel sizes (601x601)

Hi everyone,

I am working on a university project implementing a Non-Separable Gaussian Blur (the assignment explicitly requires a non-separable implementation, so I cannot switch to a separable approach) using SYCL. I am running on a Linux headless server using AdaptiveCpp as my compiler. The GPU is an Intel Arc A770.

I have implemented a standard brute-force 2D convolution kernel.

When I run the program with small or medium kernels (e.g., 31x31), the code works perfectly and produces the correct image.

However, when I test it with a large kernel size (specifically 601x601, which is required for a stress test assignment), the application hangs indefinitely at q.wait(). It never returns, no error is thrown, and I have to kill the process manually.

My Question: I haven't changed the logic or the memory management, only the kernel size variable.

Does anyone know what could be causing this hang only when the kernel size is large? And most importantly, does anyone know how to resolve this to make the kernel finish execution successfully?

Code Snippet:

```
// ... buffer setup ...
q.submit([&](handler& h) {
    // ... accessors ...
    h.parallel_for(range<2>(height, width), [=](id<2> idx) {
        int y = idx[0];
        int x = idx[1];

        // ... clamping logic ...

        for (int c = 0; c < channels; c++) {
            float sum = 0.f;
            // The heavy loop: 601 * 601 iterations
            for (int ky = -radius; ky <= radius; ky++) {
                for (int kx = -radius; kx <= radius; kx++) {
                    // ... index calculation ...
                    sum += acc_in[...] * acc_kernel[...];
                }
            }
            acc_out[...] = sum;
        }
    });
});
q.wait(); // <--- THE PROGRAM HANGS HERE
```

Thanks in advance for your help!

Dopo aver ricevuto diverse risposte, alcune delle quali ci hanno fatto presente che la dimensione del kernel è estremamente grande e che quindi il problema dipende proprio dalle sue dimensioni unitamente all'implementazione "naive" del codice, abbiamo ricevuto un messaggio che ci suggeriva di provare un altro compilatore Sycl diverso da AdaptiveCpp.

Kike328 • 17d ago

dpc++ just released a linux build that doesn't require installation or sudo, in my opinion it is worth it to check if is your fault or adaptivecpp fault as a person who has worked with sycl before i can assure that the ecosystem is everything but stable.

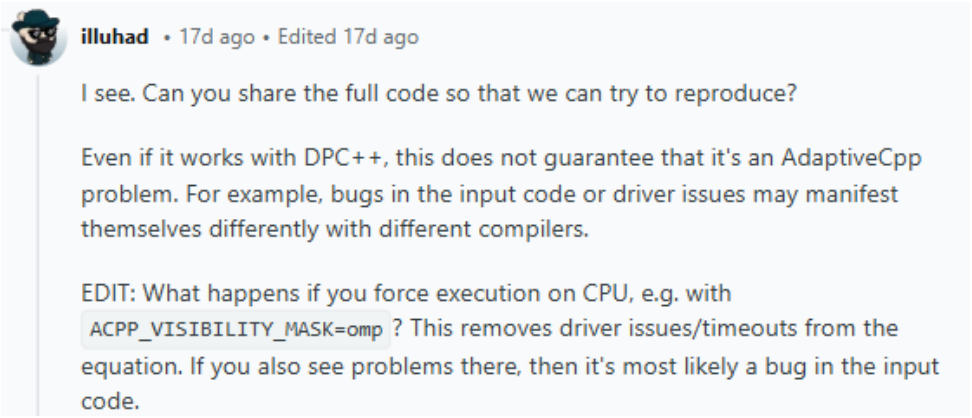
https://github.com/intel/llvm/releases/tag/v6.2.1

Download the linux build (not necessary to build it from source), point your LD_LIBRARY_PATH to the lib source in the zip folder, the PATH to the bin folder and that is, you can just compile with dpc++ (you should use clang++ -O3 -fsycl -fsycl-targets=amdgcn-amd-amdhsa -Xsycl-target-backend --offload-arch=gfx906 (your architecture here))

The full guide is here:
https://github.com/intel/llvm/blob/sycl/sycl/doc/GetStartedGuide.md#use-dpc-toolchain

Dopo aver scaricato questo compilatore abbiamo provato ad utilizzarlo ma anche così il problema non è stato risolto.

Un altro utente ci ha invece chiesto di inviargli il codice cosicché potesse provarlo.



illuhad • 17d ago • Edited 17d ago

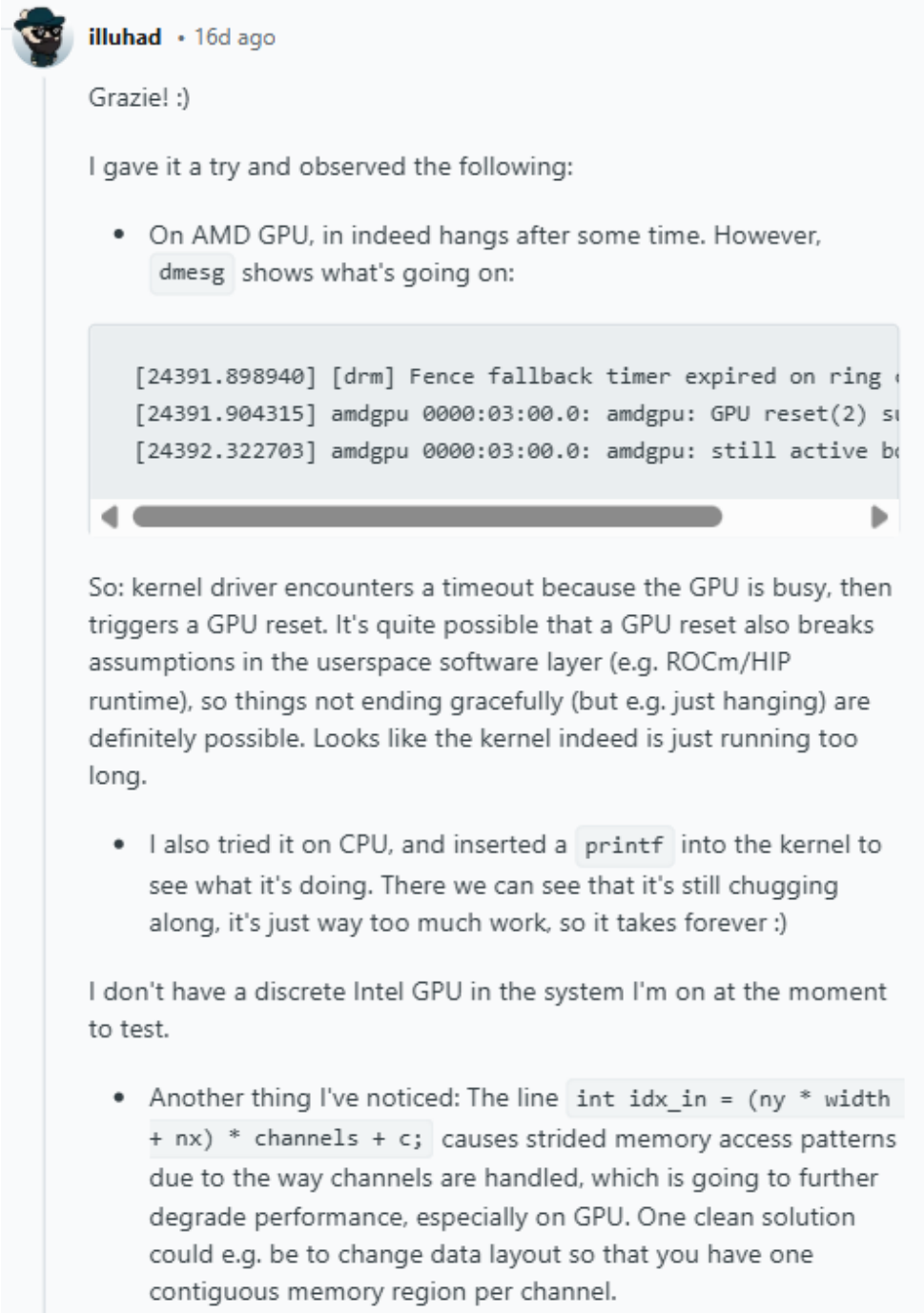
I see. Can you share the full code so that we can try to reproduce?

Even if it works with DPC++, this does not guarantee that it's an AdaptiveCpp problem. For example, bugs in the input code or driver issues may manifest themselves differently with different compilers.

EDIT: What happens if you force execution on CPU, e.g. with `ACPP_VISIBILITY_MASK=omp`? This removes driver issues/timeouts from the equation. If you also see problems there, then it's most likely a bug in the input code.

Dopo qualche tempo, questo utente ci ha spiegato che il problema è dovuto al timeout della GPU. Il codice infatti, per essere eseguito, impiega più tempo di quello consentito dal driver, che quindi resetta la scheda video interrompendo il processo. Oltre a questo, l'utente ha notato un'inefficienza nella gestione della memoria che ne rallenta ulteriormente l'esecuzione. Abbiamo quindi corretto questa inefficienza ma neanche così il problema

è stato risolto, lasciandoci quindi con l'idea che il blocco sia legato esclusivamente al timeout della GPU.



illuhad • 16d ago

Grazie! :)

I gave it a try and observed the following:

- On AMD GPU, in indeed hangs after some time. However, `dmesg` shows what's going on:

```
[24391.898940] [drm] Fence fallback timer expired on ring
[24391.904315] amdgpu 0000:03:00.0: amdgpu: GPU reset(2) su
[24392.322703] amdgpu 0000:03:00.0: amdgpu: still active bo
```

So: kernel driver encounters a timeout because the GPU is busy, then triggers a GPU reset. It's quite possible that a GPU reset also breaks assumptions in the userspace software layer (e.g. ROCm/HIP runtime), so things not ending gracefully (but e.g. just hanging) are definitely possible. Looks like the kernel indeed is just running too long.

- I also tried it on CPU, and inserted a `printf` into the kernel to see what it's doing. There we can see that it's still chugging along, it's just way too much work, so it takes forever :)

I don't have a discrete Intel GPU in the system I'm on at the moment to test.

- Another thing I've noticed: The line `int idx_in = (ny * width + nx) * channels + c;` causes strided memory access patterns due to the way channels are handled, which is going to further degrade performance, especially on GPU. One clean solution could e.g. be to change data layout so that you have one contiguous memory region per channel.

EDIT: nonostante il fatto che a vari utenti su reddit non funzionasse con input superiore a `501` $\times$ `501` (ovvero, l'esecuzione si bloccava), provandolo su flamausim e veryhotmetal, ad oggi, 26 dicembre, l'esecuzione va a buon fine con output corretto, mentre su hotmetal non ancora. Questo è forse dovuto agli aggiornamenti effettuati sulle macchine poiché non abbiamo modificato il codice in questione.

6.3.2 Vulkan

Abbiamo quindi fatto un post su reddit anche per Vulkan.

[Help] Vulkan Compute Shader: Artifacts and empty pixels appear when using very large kernels (601x601)

Hi everyone,

I am working on a university project where I need to implement a Non-Separable Gaussian Blur using Vulkan Compute Shaders. I am running the application on a headless Linux server.

I have implemented a standard brute-force 2D convolution shader. I use SSBOs for the input image, output image, and the kernel data.

When I run the program with small or medium kernels (e.g., 15x15, 31x31), everything works perfectly. The image is blurred correctly.

However, when I test it with a large kernel size (specifically 601x601), the output image is corrupted. Large sections of the image appear "empty" (transparent/black) while other parts seem processed correctly.

My Shader Implementation: The shader uses a standard nested loop approach. Here is the relevant part of the GLSL code:

version 450

layout(local_size_x = 16, local_size_y = 16) in;

layout(std430, binding = 0) readonly buffer InputImage { uint data[]; } inputImage; layout(std430, binding = 1) writeonly buffer OutputImage { uint data[]; } outputImage; layout(std430, binding = 2) readonly buffer KernelBuffer { float kernel[]; };

layout(push_constant) uniform PushConsts { int width; int height; int kerDim; // Tested with 601 } pushConsts;

void main() { ivec2 gid = ivec2(gl_GlobalInvocationID.xy); if (gid.x >= pushConsts.width || gid.y >= pushConsts.height) return;

```
vec4 color = vec4(0.0);
int radius = (pushConsts.kerDim - 1) / 2;

// Convolution loop
for (int i = -radius; i <= radius; i++) {
    for (int j = -radius; j <= radius; j++) {
        // Coordinate clamping and index calculation...
        // Accumulate color...
        color += unpackRGBA(inputImage.data[nidx]) * kernel[kidx];
    }
}

outputImage.data[idx] = packRGBA(color);
```

}

I haven't changed the logic or the memory synchronization, only the kernel size (and the corresponding `kerDim` push constant).

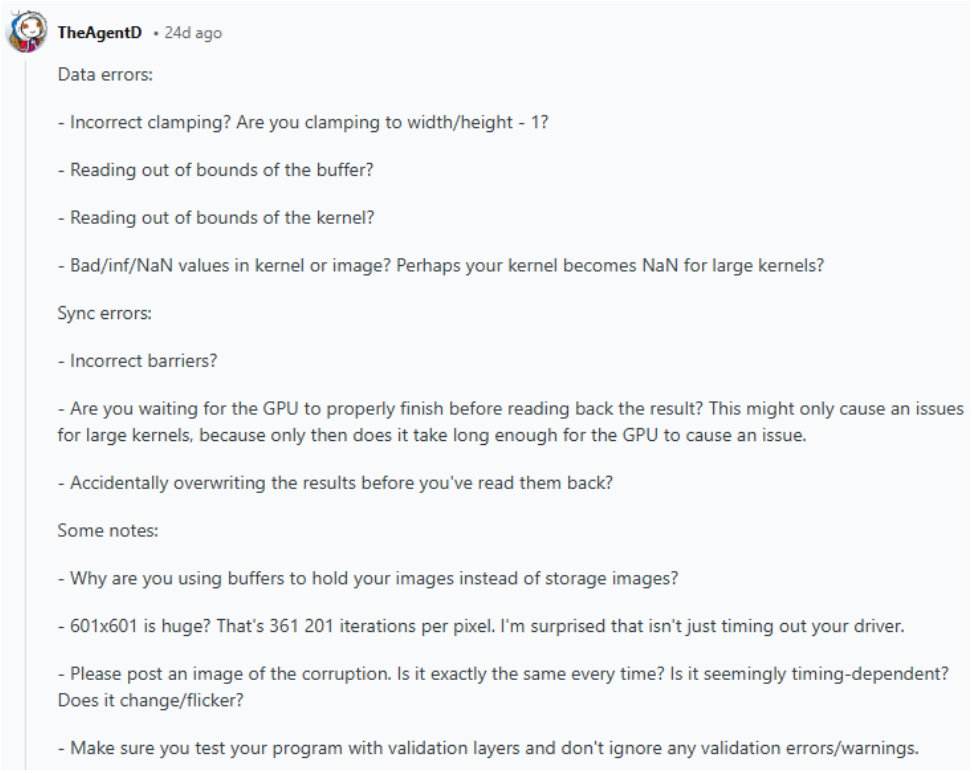
Why does the shader fail or produce incomplete output only when the kernel size is large? What could be causing these artifacts?

Does anyone know how to solve this problem without switching to a separable kernel? (I am required to strictly use a non-separable approach for this project).

Thanks in advance for your help!

Anche in questo caso abbiamo ricevuto diverse risposte interessanti, alcune delle quali ci suggerivano semplicemente, di nuovo, di ridurre la dimensione del kernel. Un utente in

particolare ci ha elencato una serie di possibili cause del problema



TheAgentD • 24d ago

Data errors:

- Incorrect clamping? Are you clamping to width/height - 1?

- Reading out of bounds of the buffer?

- Reading out of bounds of the kernel?

- Bad/inf/NaN values in kernel or image? Perhaps your kernel becomes NaN for large kernels?

Sync errors:

- Incorrect barriers?

- Are you waiting for the GPU to properly finish before reading back the result? This might only cause an issues for large kernels, because only then does it take long enough for the GPU to cause an issue.

- Accidentally overwriting the results before you've read them back?

Some notes:

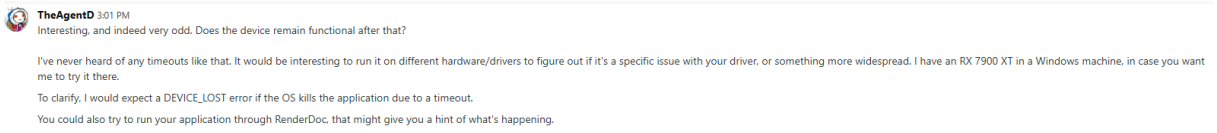
- Why are you using buffers to hold your images instead of storage images?

- 601x601 is huge? That's 361 201 iterations per pixel. I'm surprised that isn't just timing out your driver.

- Please post an image of the corruption. Is it exactly the same every time? Is it seemingly timing-dependent? Does it change/flicker?

- Make sure you test your program with validation layers and don't ignore any validation errors/warnings.

Dopo aver controllato tutti gli elementi della lista, l'utente ci ha chiesto, come accaduto per Sycl, di inviargli il nostro codice in modo che potesse provarlo.



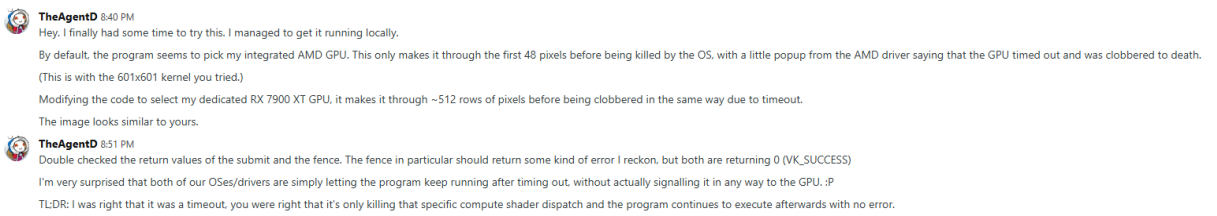
TheAgentD 3:01 PM
Interesting, and indeed very odd. Does the device remain functional after that?

I've never heard of any timeouts like that. It would be interesting to run it on different hardware/drivers to figure out if it's a specific issue with your driver, or something more widespread. I have an RX 7900 XT in a Windows machine, in case you want me to try it there.

To clarify, I would expect a DEVICE_LOST error if the OS kills the application due to a timeout.

You could also try to run your application through RenderDoc, that might give you a hint of what's happening.

Qualche giorno dopo l'utente ci ha risposto dicendoci che, anche in questo caso, il problema è dovuto al timeout della GPU. Tuttavia, l'utente ha notato un comportamento anomalo: il codice restituisce **VK_SUCCESS**, cioè le API Vulkan non riscontrano alcun problema nonostante l'esecuzione venga interrotta prematuramente dal timeout.



TheAgentD 8:40 PM
Hey. I finally had some time to try this. I managed to get it running locally.

By default, the program seems to pick my integrated AMD GPU. This only makes it through the first 48 pixels before being killed by the OS, with a little popup from the AMD driver saying that the GPU timed out and was clobbered to death.

(This is with the 601x601 kernel you tried.)

Modifying the code to select my dedicated RX 7900 XT GPU, it makes it through ~512 rows of pixels before being clobbered in the same way due to timeout.

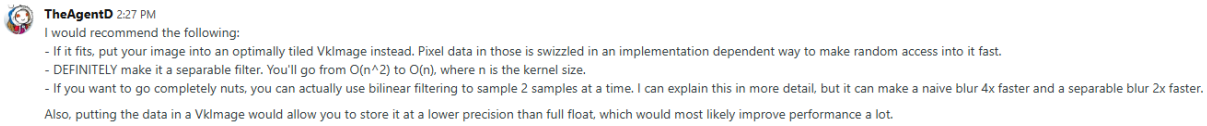
The image looks similar to yours.

TheAgentD 8:51 PM
Double checked the return values of the submit and the fence. The fence in particular should return some kind of error I reckon, but both are returning 0 (VK_SUCCESS)

I'm very surprised that both of our OSes/drivers are simply letting the program keep running after timing out, without actually signalling it in any way to the GPU. :P

TL;DR: I was right that it was a timeout, you were right that it's only killing that specific compute shader dispatch and the program continues to execute afterwards with no error.

L'utente, sebbene non sappia dare una risposta certa a tale comportamento, ci ha comunque suggerito una serie di possibili migliorie, evidenziando in particolare l'utilizzo della separabilità del kernel gaussiano (alternativa che avevamo già implementato).


TheAgentD 2:27 PM
I would recommend the following:
- If it fits, put your image into an optimally tiled VkImage instead. Pixel data in those is swizzled in an implementation dependent way to make random access into it fast.
- DEFINITELY make it a separable filter. You'll go from O(n^2) to O(n), where n is the kernel size.
- If you want to go completely nuts, you can actually use bilinear filtering to sample 2 samples at a time. I can explain this in more detail, but it can make a naive blur 4x faster and a separable blur 2x faster.

Also, putting the data in a VkImage would allow you to store it at a lower precision than full float, which would most likely improve performance a lot.

6.4 Considerazioni finali

Dopo aver esplorato tutti i suggerimenti degli utenti senza successo siamo arrivati alla conclusione che questa versione del codice, nonostante le inefficienze messe in evidenza, non può funzionare se non provando ad aumentare il timeout della GPU. Tuttavia non abbiamo tentato questa strada anche perché, per questo codice nello specifico, esiste un modo migliore per aggirare il problema, ovvero sfruttare la separabilità del kernel. Ad ogni modo, per altre tipologie di codice, può valere la pena provare a modificare il timeout della GPU.

7 Test di performance

A questo punto abbiamo implementato 3 versioni del codice gaussian blur e una versione del codice regola del 30, sia in Vulkan che in Sycl. Abbiamo quindi testato questi codici sulle tre macchine a nostra disposizione, ognuna con una GPU diversa:

- hotmetal: Intel Arc A770 GPU;
- veryhotmetal: AMD Navi 31 Radeon RX7900 GRE GPU;
- flamausim: NVIDIA GeForce RTX 4060 Ti GPU;

Seguono dei grafici che mostrano, per ogni codice, il tempo di esecuzione in secondi al variare dell'input.

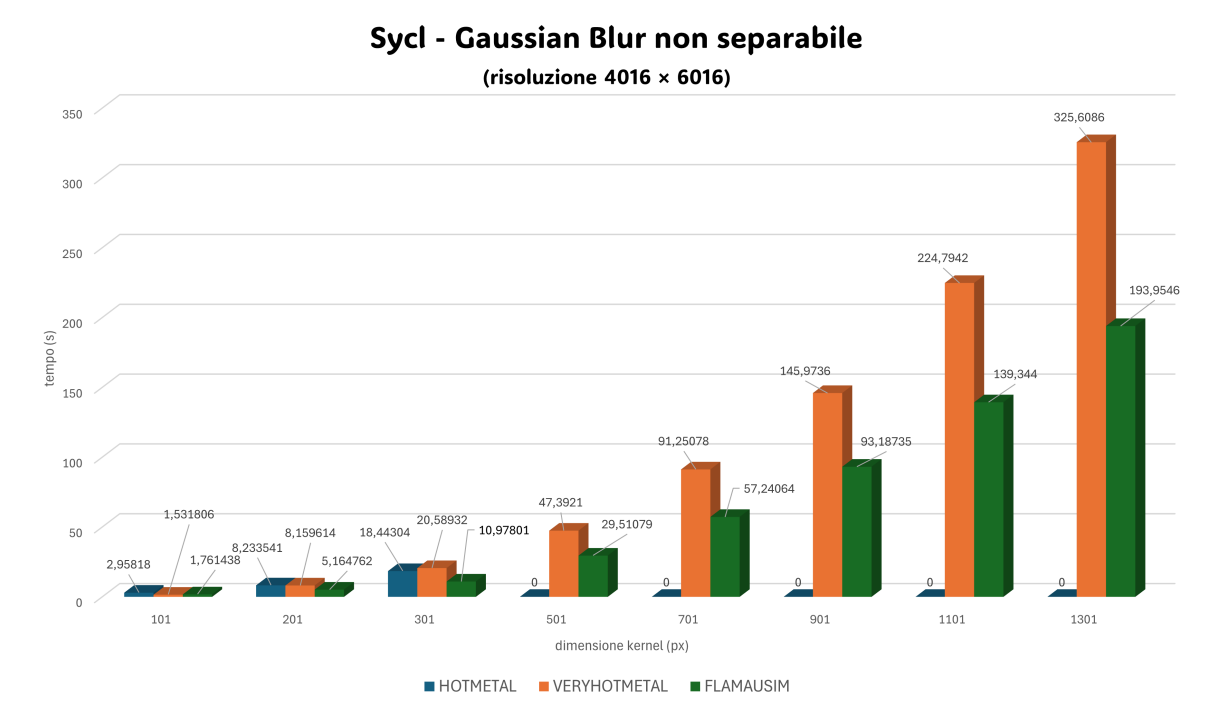
7.1 Gaussian blur

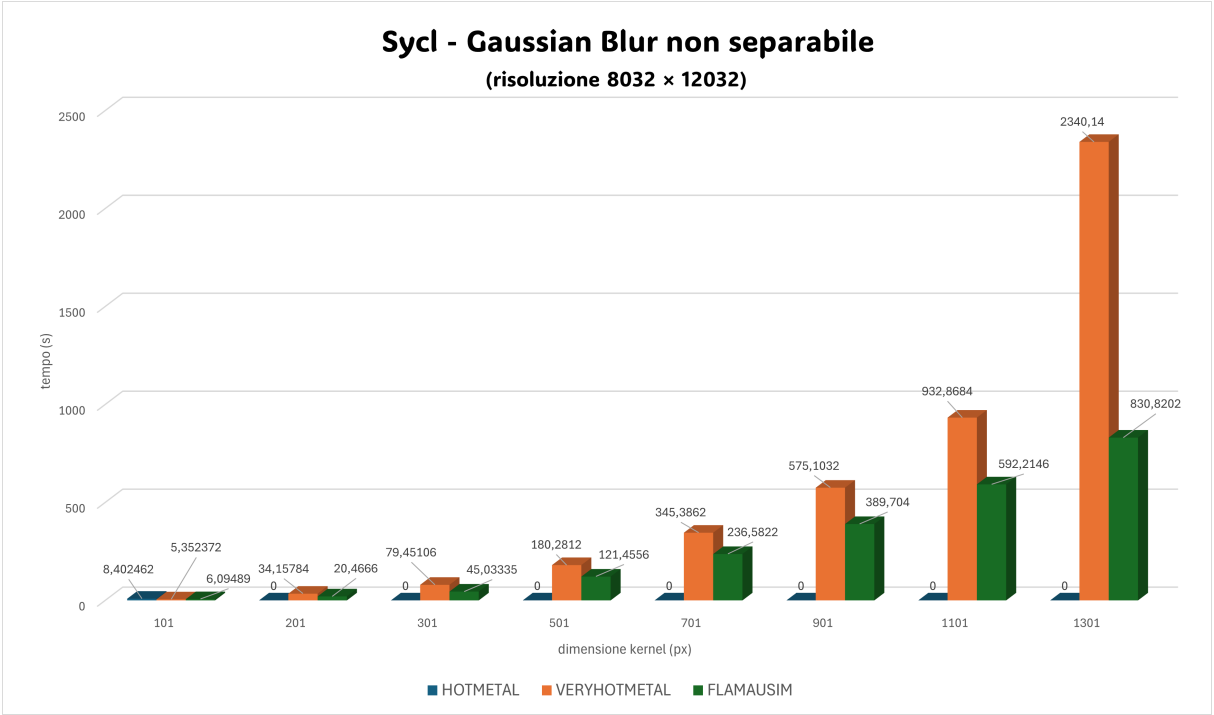
Come già detto, per entrambi i linguaggi abbiamo tre versioni:

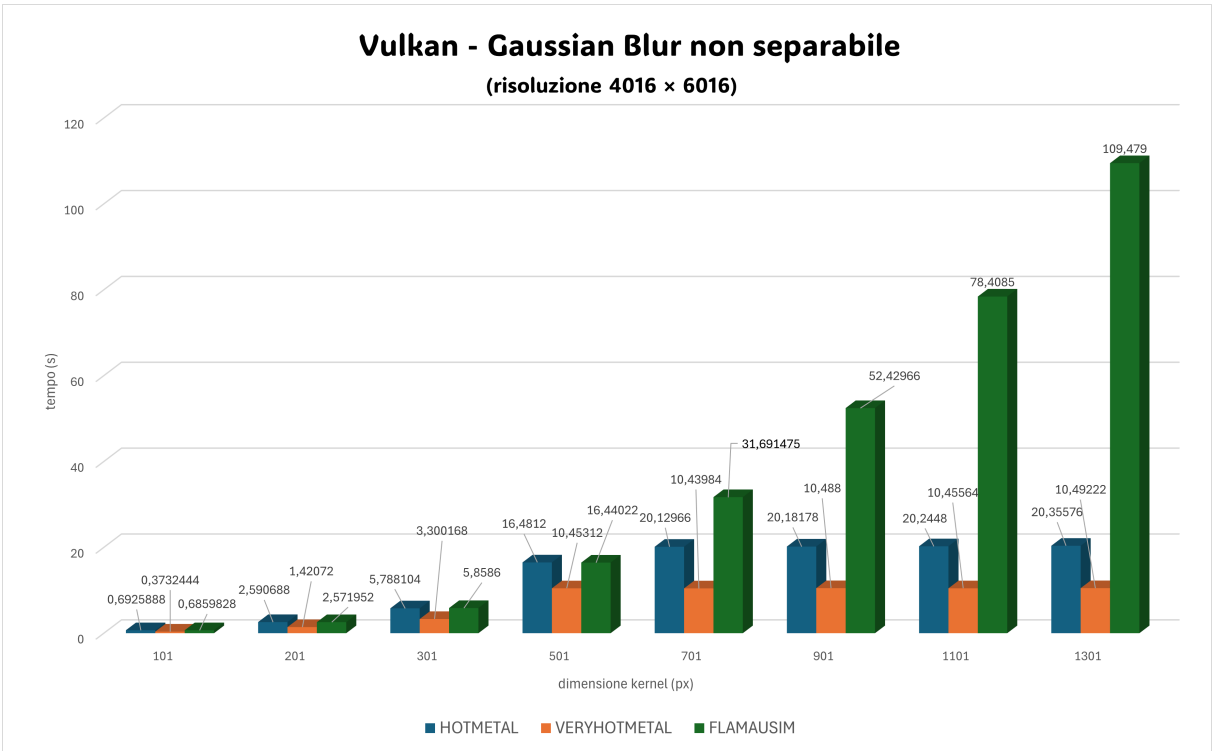
1. la prima versione, che non sfrutta la separabilità del kernel e che presenta il problema di esecuzione (output corrotto per Vulkan e loop infinito per Sycl);
2. la seconda, che non sfrutta la separabilità del kernel ma che invia l'immagine di input divisa in chunk;
3. la terza, che invece sfrutta la separabilità del kernel gaussiano;

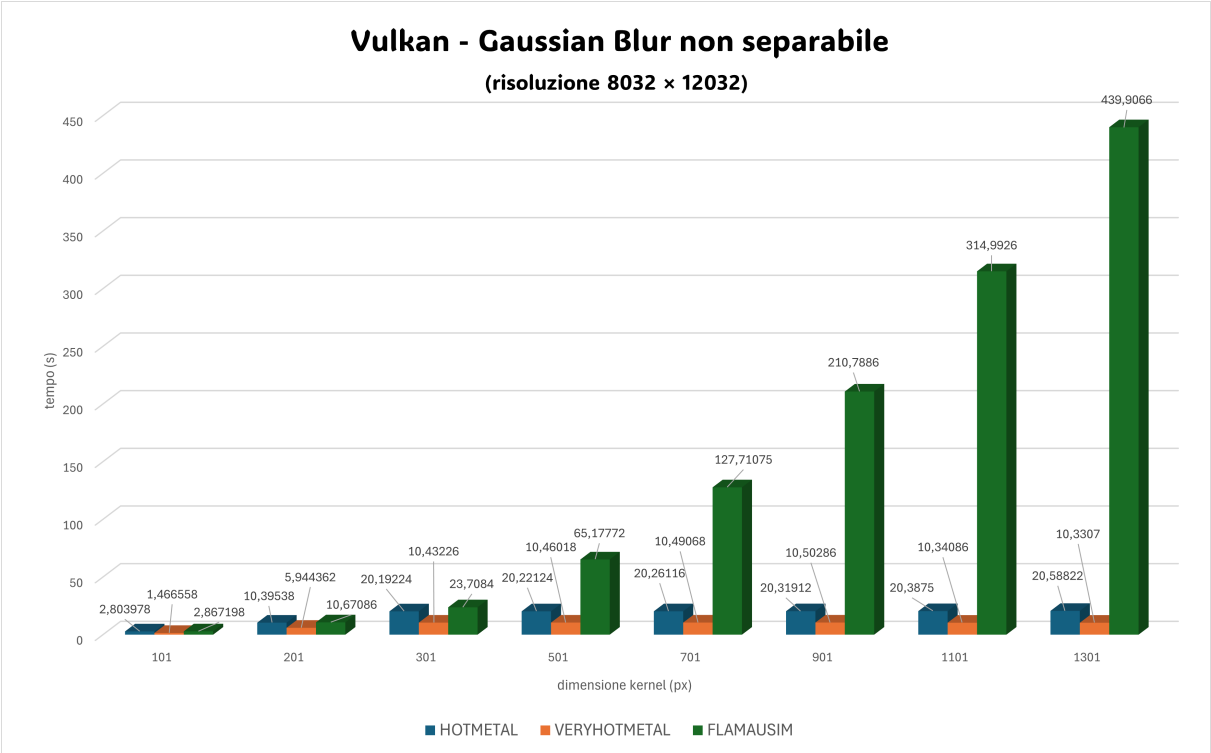
   Ogni versione è stata provata su due immagini di input, la prima di dimensioni `4016` × `6016` e la seconda `8032` × `12032` e variando le dimensioni del kernel; in particolare abbiamo testato le dimensioni che vanno da `101` × `101` a `1301` × `1301`.

7.1.1 Gaussian blur



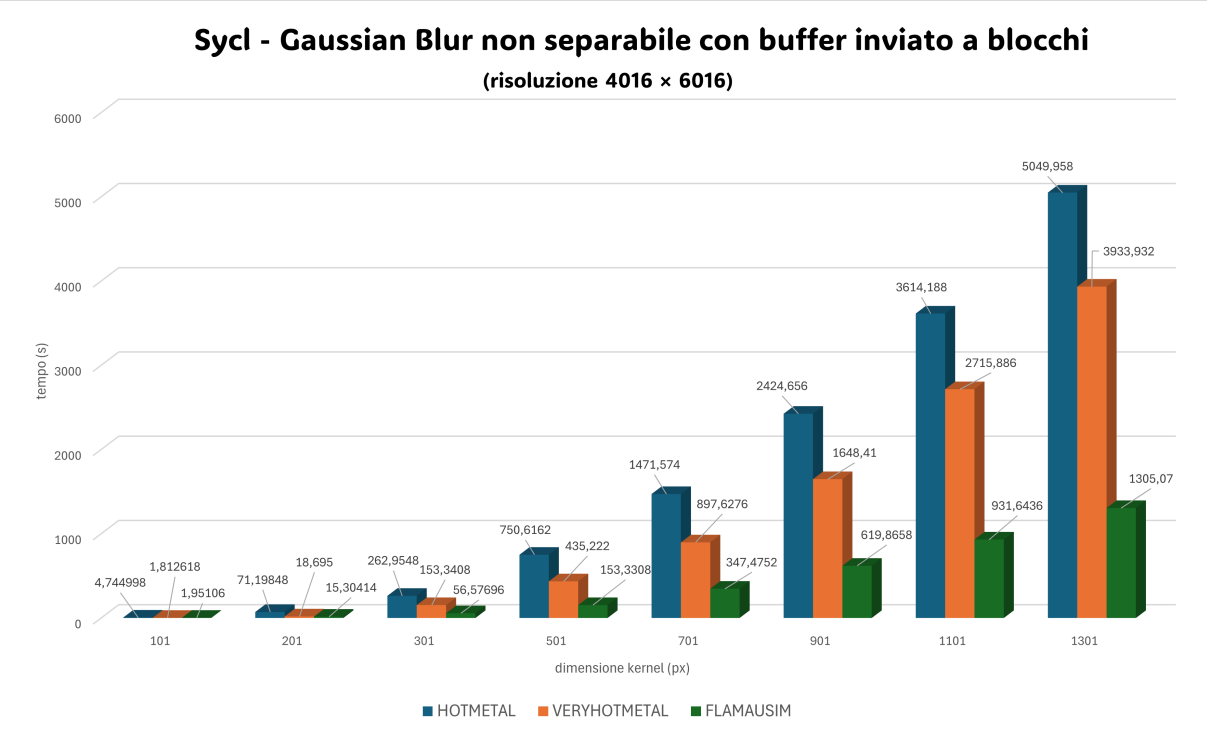

Sycl - Gaussian Blur non separabile
(risoluzione 8032 × 12032)
tempo (s)
2500
2000
1500
1000
500
0
8,402462
5,352372
6,09489
34,15784
0
20,4666
79,45106
0
45,03335
180,2812
0
121,4556
345,3862
0
236,5822
575,1032
0
389,704
932,8684
0
592,2146
2340,14
0
830,8202
101
201
301
501
701
901
1101
1301
dimensione kernel (px)
HOTMETAL
VERYHOTMETAL
FLAMAUSIM

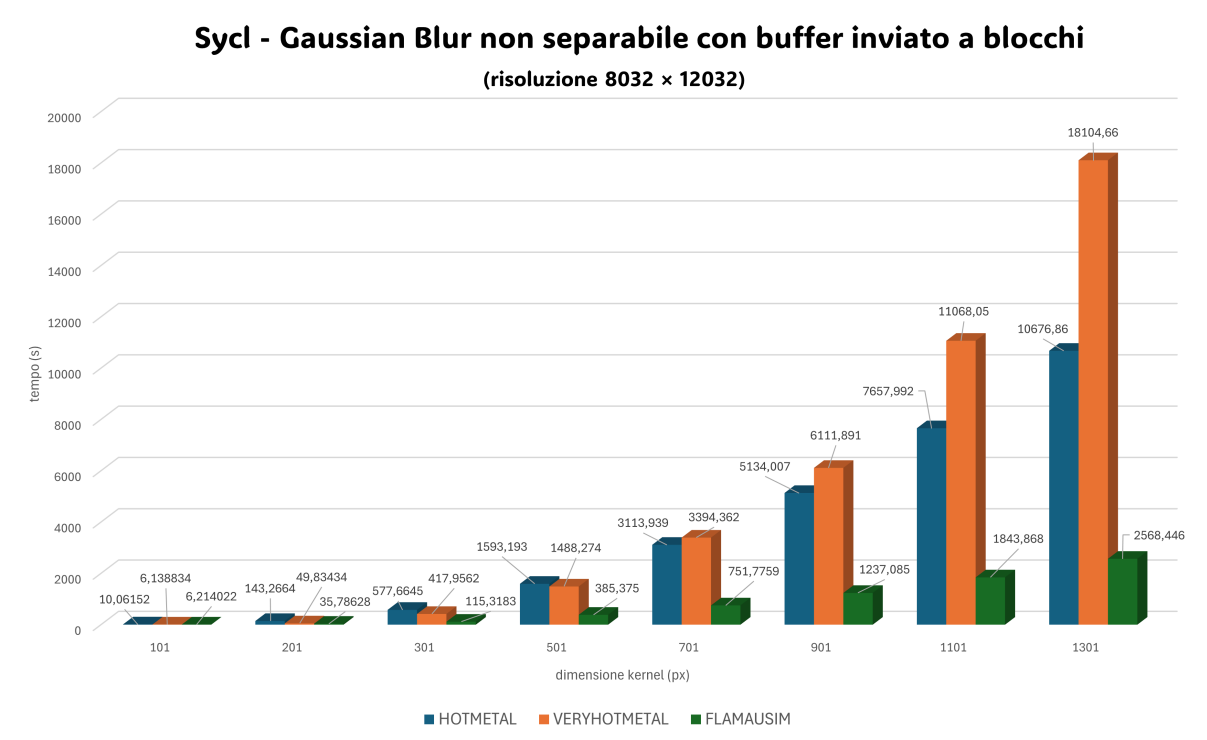

Vulkan - Gaussian Blur non separabile
(risoluzione 4016 × 6016)
tempo (s)
120
100
80
60
40
20
0
0,6925888
0,3732444
0,6859828
2,590688
1,42072
2,571952
5,788104
3,300168
5,8586
16,4812
10,45312
16,44022
20,12966
10,43984
31,691475
20,18178
10,488
52,42966
20,2448
10,45564
78,4085
20,35576
10,49222
109,479
101
201
301
501
701
901
1101
1301
dimensione kernel (px)
HOTMETAL
VERYHOTMETAL
FLAMAUSIM

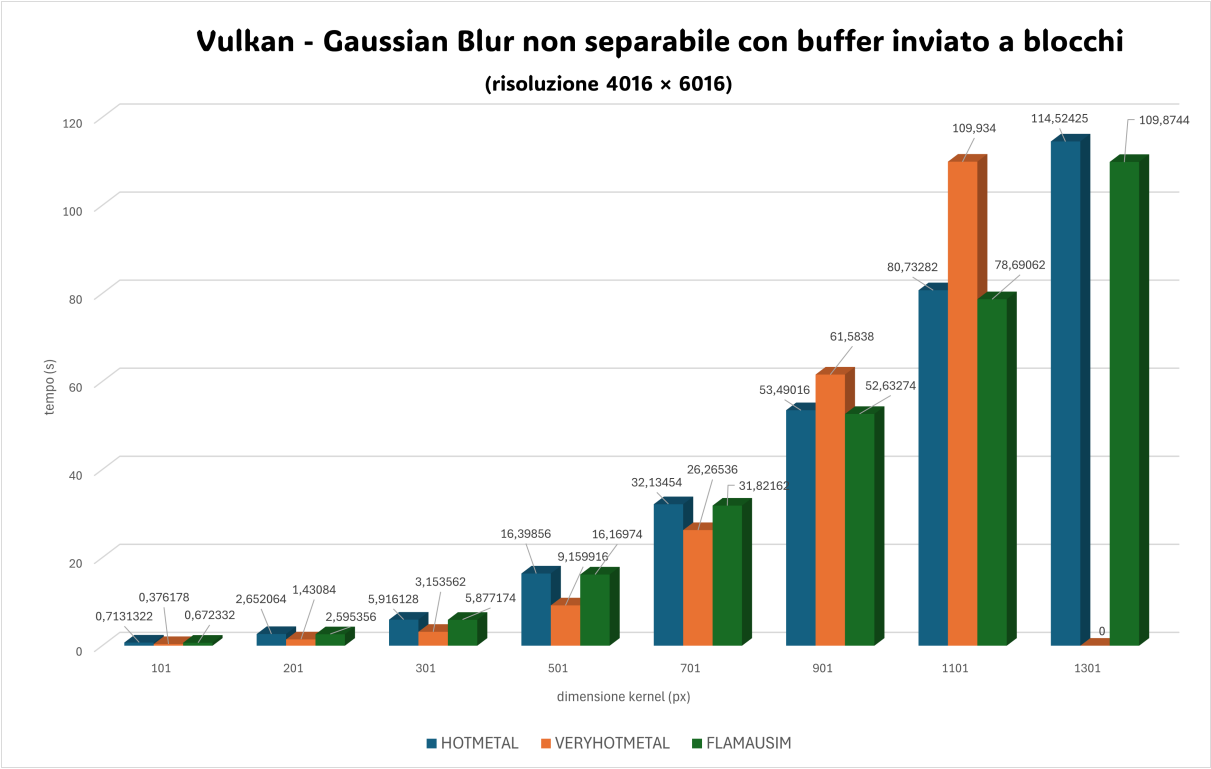


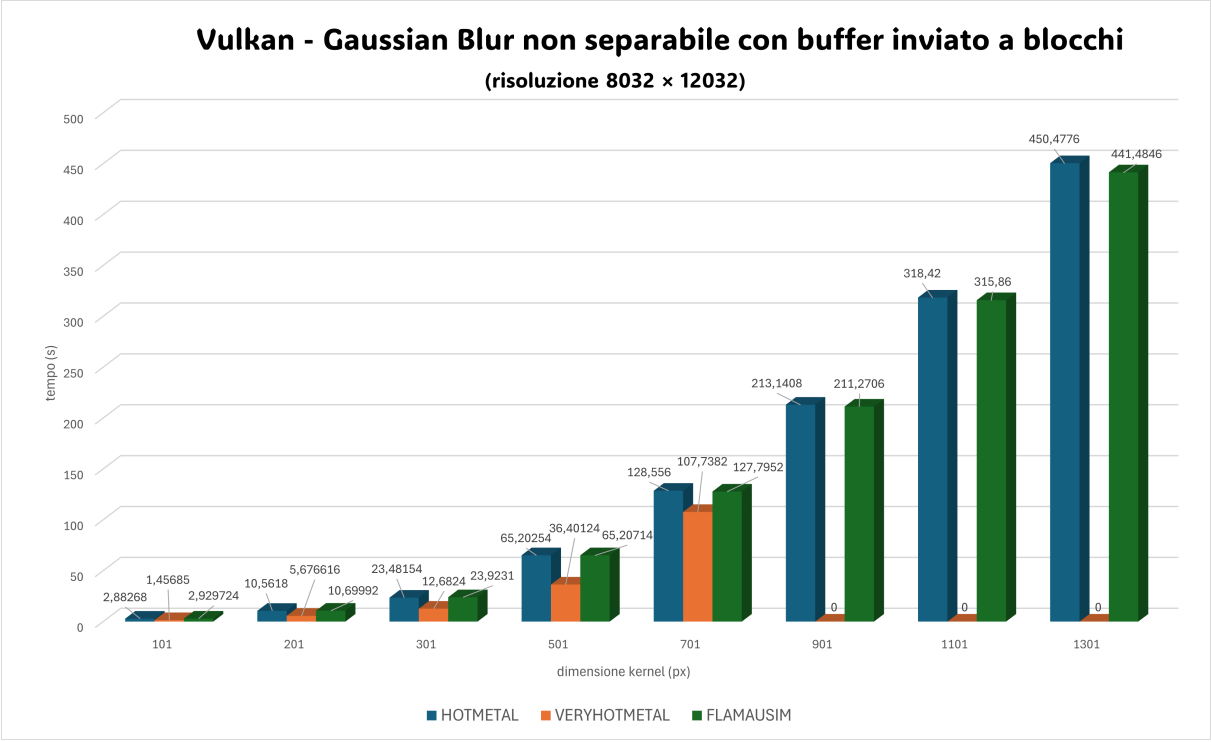
Nota. Nei grafici Vulkan, da una certa dimensione in poi, il tempo di esecuzione nelle macchine hotmetal e veryhotmetal è costante (20 s per hotmetal e 10.5 s per veryhotmetal): questo accade perché l'immagine prodotta non è corretta e presenta delle zone non elaborate come discusso sopra.

7.1.2 Gaussian blur con immagine inviata in chunk







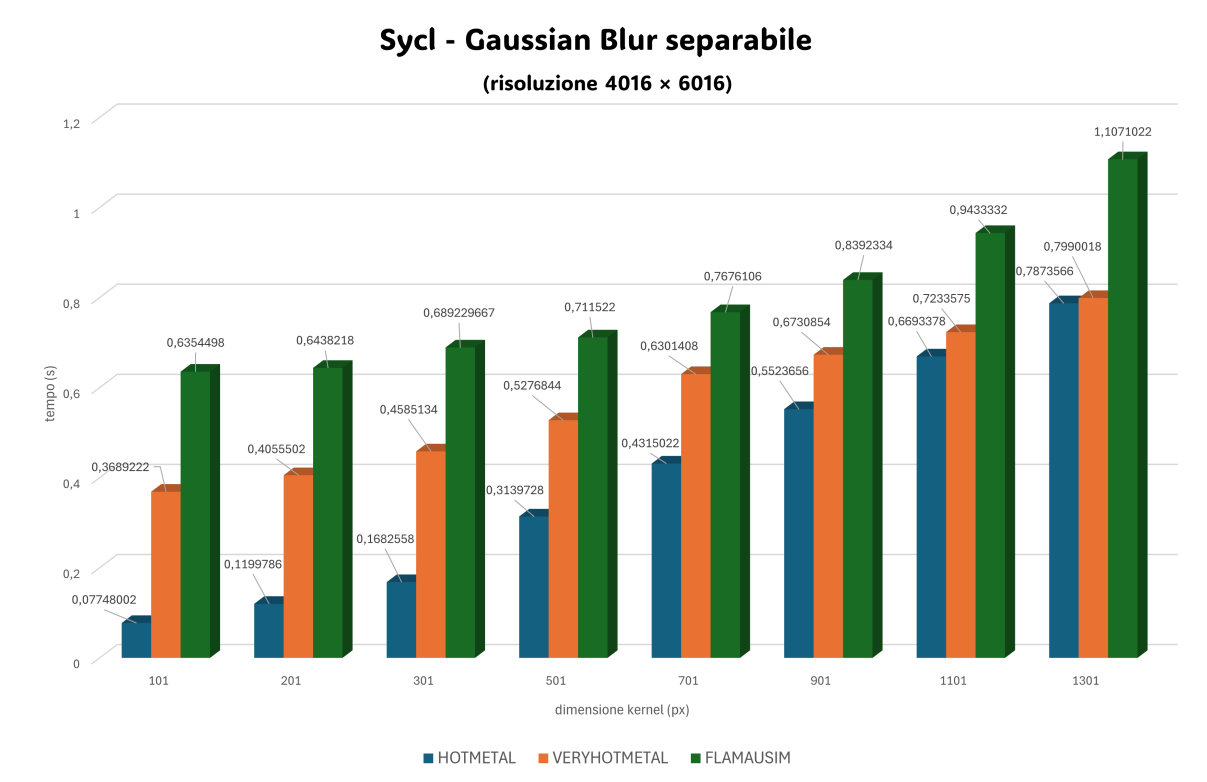


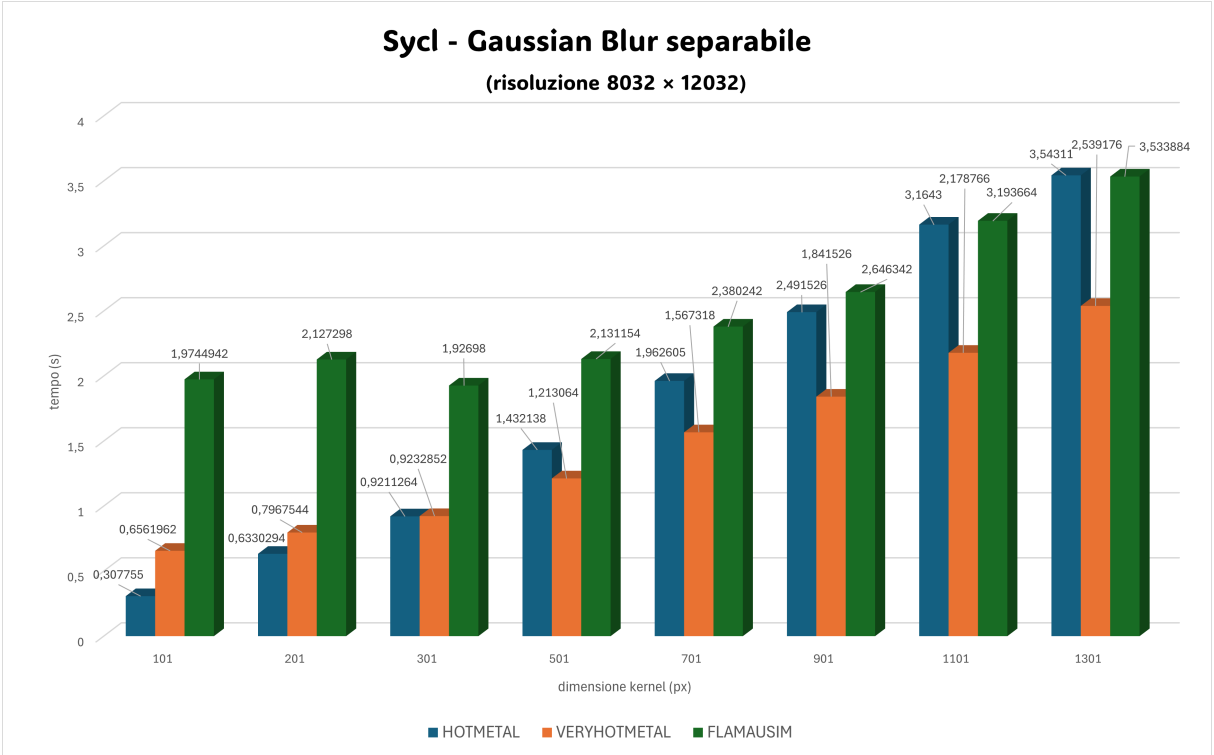
Nota. Nei grafici Vulkan di questa versione, sulla macchina veryhotmetal, va in crash con dimensioni del kernel troppo grandi restituendo il seguente errore

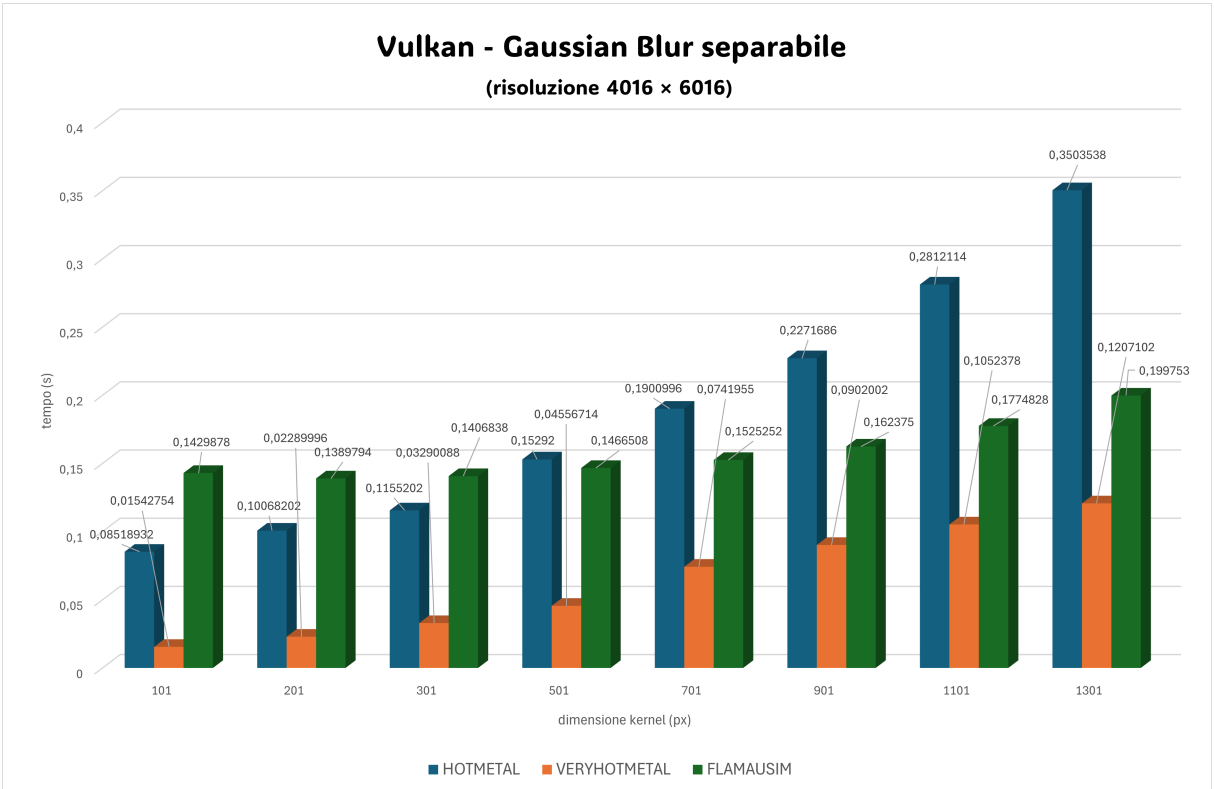
```
radv/amdgpu: the cs has been cancelled because the context is lost. this
    context is guilty of a hard recovery. terminate called after throwing an
     instance of 'std::runtime_error' what(): impossibile inviare comandi
    alla coda. aborted (core dumped)
```

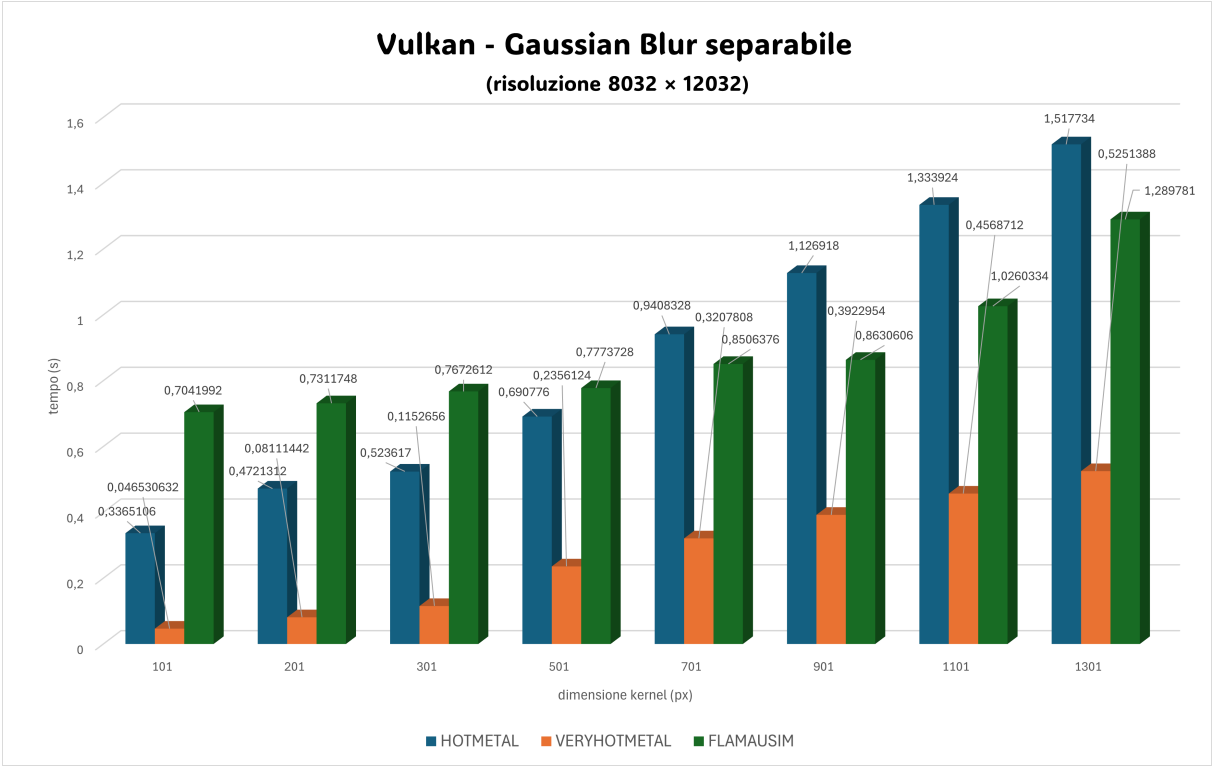
Questo errore è certamente dovuto, ancora una volta, al timeout della GPU.

7.1.3 Gaussian blur separabile



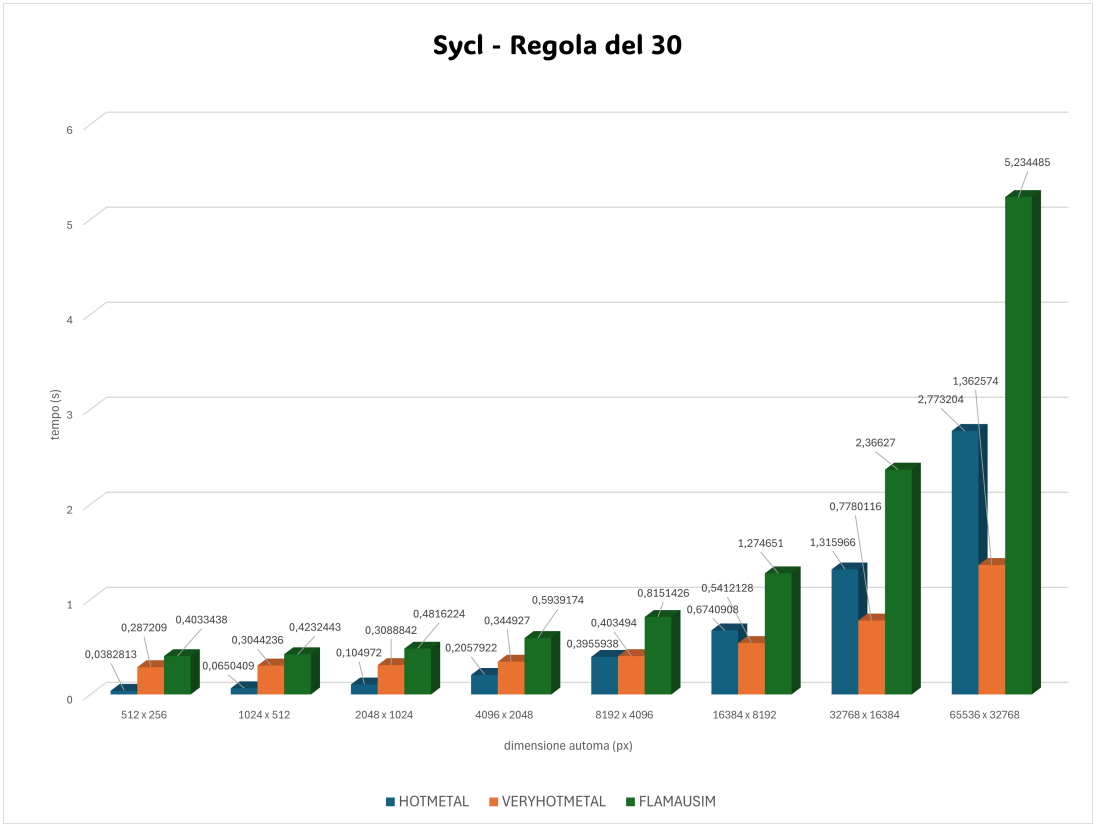

Sycl - Gaussian Blur separabile
(risoluzione 8032 × 12032)
tempo (s)
dimensione kernel (px)
HOTMETAL
VERYHOTMETAL
FLAMAUSIM
0,307755
0,6561962
1,9744942
0,6330294
0,7967544
2,127298
0,9211264
0,9232852
1,92698
1,432138
1,213064
2,131154
1,962605
1,567318
2,380242
2,491526
1,841526
2,646342
3,1643
2,178766
3,193664
3,54311
2,539176
3,533884

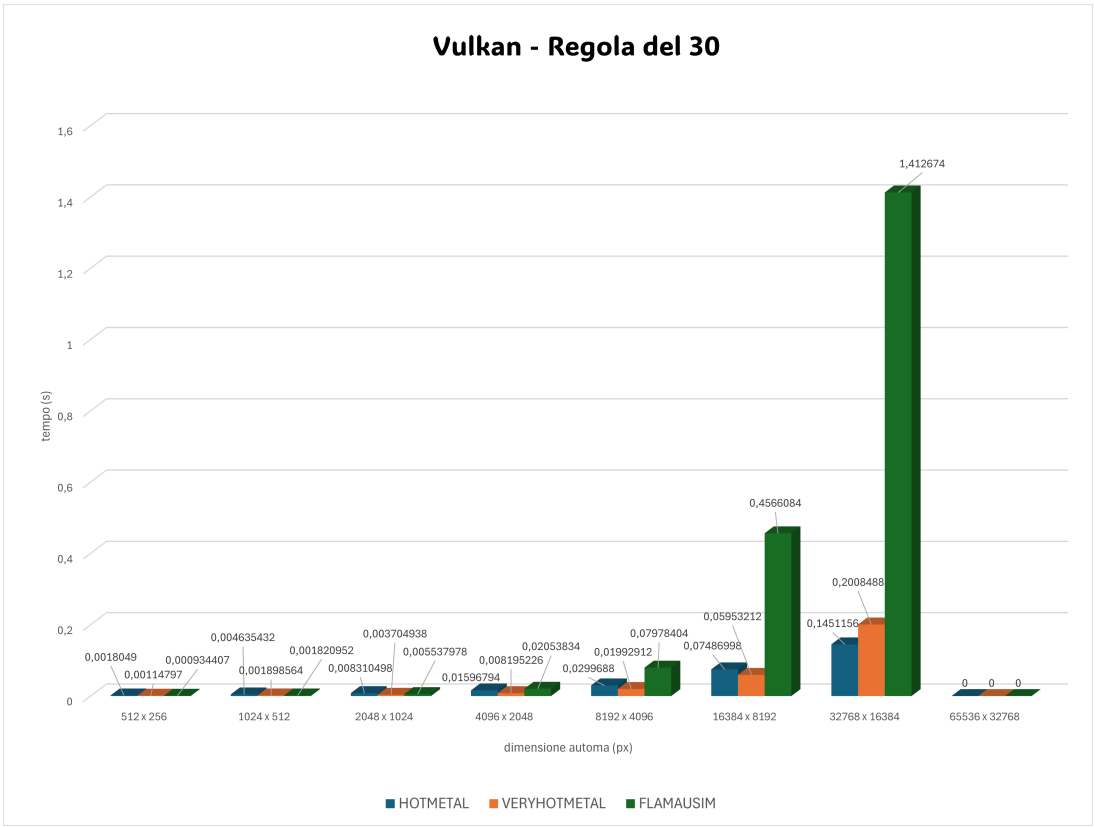

Vulkan - Gaussian Blur separabile
(risoluzione 4016 × 6016)
tempo (s)
dimensione kernel (px)
HOTMETAL
VERYHOTMETAL
FLAMAUSIM
0,08518932
0,01542754
0,1429878
0,10068202
0,02289996
0,1389794
0,1155202
0,03290088
0,1406838
0,15292
0,04556714
0,1466508
0,1900996
0,0741955
0,1525252
0,2271686
0,0902002
0,162375
0,2812114
0,1052378
0,1774828
0,3503538
0,1207102
0,199753



7.2 Regola del 30

Per quanto riguarda regola del 30 abbiamo un'unica versione ed è funzionante con qualsiasi input che sia una potenza di 2, ad eccezione di Vulkan con input `65536` ma, come già detto, dovrebbe dipendere dal modo in cui gestiamo la memorizzazione dei dati.


Sycl - Regola del 30
tempo (s)
0
1
2
3
4
5
6
0,0382813
0,287209
0,4033438
0,0650409
0,3044236
0,4232443
0,104972
0,3088842
0,4816224
0,2057922
0,344927
0,5939174
0,3955938
0,403494
0,8151426
0,6740908
0,5412128
1,274651
1,315966
0,7780116
2,36627
2,773204
1,362574
5,234485
512 x 256
1024 x 512
2048 x 1024
4096 x 2048
8192 x 4096
16384 x 8192
32768 x 16384
65536 x 32768
dimensione automa (px)
HOTMETAL
VERYHOTMETAL
FLAMAUSIM


Vulkan - Regola del 30
tempo (s)
0
0,2
0,4
0,6
0,8
1
1,2
1,4
1,6
0,0018049
0,00114797
0,000934407
0,004635432
0,001898564
0,001820952
0,003704938
0,008310498
0,005537978
0,01596794
0,008195226
0,02053834
0,0299688
0,01992912
0,07978404
0,07486998
0,05953212
0,4566084
0,1451156
0,2008488
1,412674
0
0
0
512 x 256
1024 x 512
2048 x 1024
4096 x 2048
8192 x 4096
16384 x 8192
32768 x 16384
65536 x 32768
dimensione automa (px)
HOTMETAL
VERYHOTMETAL
FLAMAUSIM

8 Conclusioni

Il progetto ha avuto una serie di difficoltà. In primo luogo abbiamo dovuto imparare ad utilizzare i due linguaggi richiesti dalla traccia del progetto, compito che ha richiesto la consultazione delle varie documentazioni e risorse presenti online. Siamo passati poi alla progettazione degli algoritmi, processo che, specialmente per Vulkan, richiede non poco tempo. Infine, dopo aver implementato i codici è arrivata la fase di debug e ottimizzazione che ha richiesto diversi giorni anche per via dell'attesa necessaria per le risposte ai nostri post su reddit.

Questo lavoro ci ha permesso di acquisire una conoscenza pratica su:

- gestione avanzata dell'ambiente Linux tramite terminale;
- programmazione parallela su GPU utilizzando sia API a basso livello (Vulkan) che framework ad astrazione superiore (SYCL);
- l'acquisizione di tecniche di programmazione di base di calcolo parallelo su GPU;
- tecniche di ottimizzazione algoritmica, come la riduzione della complessità computazionale tramite kernel separabili;
- comprensione dei vincoli hardware, i limiti degli interi e i meccanismi di timeout del driver;
- monitoraggio delle performance su server remoti
- approccio alla risoluzione dei problemi tramite analisi della documentazione e confronto con community tecniche.