**Motivation**

Qualitätskontrollen von allen möglichen Produkten, wie Spritzen, Autoteilen, Verpackungen usw. sind aus der heutigen Welt nicht mehr wegzudenken. Dabei wird geprüft ob Einzelteile vorhanden sind, die Form des Produkts passt, der Durchmesser im richtigen Bereich liegt und noch vieles mehr. Das muss natürlich alles automatisiert und mit möglichst hoher Geschwindigkeit ablaufen. Deshalb werden solche Qualitätskontrollen häufig durch Bildverarbeitung gelöst. Da ich dieses Semester mein „Betreutes Praktisches Studienprojekt“ bei NeuroCheck (einem Bildverarbeitungsunternehmen) durchführe, habe ich bereits einige Einblicke in das Thema Bildverarbeitung bekommen. Bei einer typischen Qualitätskontrolle von Prüfteilen werden verschiedene Schritte durchgeführt. Dazu gehört beispielsweise das Laden des aufgenommenen Bildes mit der Kamera in NeuroCheck, die Definition von Arbeitsbereichen und die Vorverarbeitung der Bilder. Komplexere Prüfprogramme können aus hunderten von Einzelschritten bestehen.

Da es bei Prüfteilen im Fertigungsbetrieb häufig dazu kommen kann, dass die Prüfteile an einer anderen Stelle vorliegen oder sogar gedreht sind, kann nicht immer an derselben Stelle geprüft werden. Die Position der Objekte muss neu bestimmt werden. Wie finde ich also ein Objekt in einem Bild zuverlässig? Ein häufig verwendetes Verfahren nutzt den Mustervergleich und nennt sich Template Matching.

**Definition Problemstellung**

Innerhalb eines mit einer Kamera aufgenommenen Bildes oder innerhalb eines abgespeicherten Bildes soll ein abgespeichertes Referenzmuster möglichst schnell wiedergefunden werden. Das abgespeicherte Referenzmuster entspricht nicht zu hundert Prozent dem entsprechenden Bildausschnitt aus dem Kamerabild. Um die Ähnlichkeit der Bildausschnitte zu quantifizieren wird ein Vergleichsmaß definiert. Das Referenzmuster wird durch das Kamerabild geschoben und gibt an jeder zu vergleichenden Stelle das Maß aus. Das Maß mit dem besten Wert ist dann der gesuchte Bildausschnitt. Eine Erweiterung hiervon ist, mehrere Referenzbilder durch das Bild laufen zu lassen. Ein einfacher Anwendungsfall hierfür wäre beispielsweise Bauteile/Objekte oder sogar Buchstaben auf allen möglichen Oberflächen wie Papier, Metall etc. zu klassifizieren.

**Lösungsansatz**

Das Referenzmuster kann aus einem aufgenommenen Kamerabild oder einem abgespeicherten Bild ausgeschnitten werden. Das Referenzmuster durchläuft jetzt bei jedem gegebenen Bild mit vordefinierten Schrittgrößen die einzelnen Pixel und vergleicht jeden Pixel im gegebenen Bild mit dem äquivalenten Pixel im Referenzmuster. Das kann man sich so vorstellen, dass das Muster über das Bild gelegt wird und alle übereinanderliegenden Pixel verglichen werden. Die Differenz der Pixelwerte wird gebildet und die Differenzen werden alle addiert, bis das Referenzmuster wieder verschoben wird. Die geringste absolute Zahl stellt dann die größte Ähnlichkeit mit dem Referenzmuster dar und ist meist auch der gesuchte Bildausschnitt.

Ein Bildausschnitt, der sich stark vom Hintergrund abhebt und zusätzlich nicht so leicht mit anderen Teilen im Bild verwechselbar ist, wäre also optimal und wäre vergleichsweise einfach zu finden.

Um die Pixelwerte vergleichen zu können, müssen sowohl das Referenzmuster, als auch das zu vergleichende Bild als Array vorliegen (Vergleich der RGB-Werte).

Jetzt wäre es natürlich noch gut zu wissen, was letztendlich für ein Teilausschnitt des Bildes gefunden wurde. Sozusagen eine Umrahmung des gefundenen Teilausschnitts, das Bild mit der Umrahmung wird dann als neues Bild erstellt.

Da dieser Ansatz auch gute Möglichkeiten zur Parallelisierung und allgemein für Performance-Verbesserungen hergibt (Indexierung von Arrays, Addition von Arraywerten (SIMD?) usw.), bietet er sich für Optimierungen hervorragend an. ­­­

Bei der Erweiterung mit mehreren Referenzbildern die durch ein Bild laufen, lässt sich der Ansatz mit der Parallelisierung noch viel besser verfolgen, da jeder Thread genau ein Referenzbild (Template) übernehmen kann. Bei Bildern in denen erkannt werden soll um welches Objekt oder welchen Buchstaben es sich handelt, wird das Referenzbild mit der besten Übereinstimmung genommen, also das mit der geringsten absoluten Zahl (auch Loss genannt).

Um die Parallelisierung noch weiter zu beschleunigen, kann der gesamte BruteForce-Algorithmus für das Pattern-Matching auf der GPU laufen, dies wird mit OpenCL realisiert.

**Implementierung inklusive Schwierigkeiten**

Es wurde durchgehend mit der Entwicklungsumgebung Visual Studio 2019 gearbeitet.

Der erste große Schritt ist die Implementierung der Funktionalität ohne Performanceverbesserungen wie Parallelisierung, SIMD usw. Um überhaupt das Bild mit dem Referenzbild vergleichen zu können muss das Bild geladen werden. STBI\_IMAGE stellt eine dazu passende Funktionalität mit der „Load“ Methode bereit. Das Bild soll als zweidimensionales Integer-Array vorliegen und indexiert werden. Die erste Dimension stellt die Breite und Höhe des Bildes dar und die zweite Dimension die verschiedenen Channels. Pro Pixel in der Breite und Höhe gibt es nämlich drei Channels für die entsprechende Farbe: Rot, Grün und Blau (RGB). Der Speicherplatz für dieses „Int-Array“ muss aber vorher allokiert werden, dies wird durch „malloc“ erreicht. Es werden Pointer mit der Größe der ersten Dimension des Arrays erstellt, die dann wiederum intern auf die Größe der zweiten Dimension zeigen (bei RGB 3mal die Größe von Integer). Das wird mit einer „For-Schleife“ erreicht. Das Template (Referenzbild) kann zu Testzwecken direkt aus dem Array mit dem Bild zugewiesen werden. Der „Loss“ ist dann natürlich beim Durchlaufen irgendwann immer 0, weil dieser Ausschnitt im Bild exakt so existiert. Das Bild, das Referenzbild und die entsprechenden Größen dazu werden dem „BruteForce-Algorithmus“ übergeben. Der „BruteForce-Algorithmus“ berechnet jetzt zu jeder Position des Referenzbildes die addierte Differenz zwischen den Pixelwerten im Referenzbild. Dazu sind im Standardfall 4 „For-Schleifen“ nötig, es wäre aber mit entsprechender Indexierung auch mit minimal 2 „For-Schleifen“ möglich. Das Referenzbild springt in X- und Y-Richtung immer um 5 oder sogar 10 Positionen, um Performanceproblemen vorzubeugen. Die X- und Y-Position des Referenzbildes stellen deren linke obere Ecke bzw. dessen Startpunkt dar. Wenn man sich nämlich nur um beispielsweise 1 Pixel in X-Richtung bewegt, ändert sich der Loss des Referenzbildes nicht wesentlich und es wird dadurch auch kein wirklich neues Muster gefunden. Innerhalb des Referenzbildes muss aber jeder Pixel indexiert werden, um den „Loss“ richtig zu berechnen. Nach jeder Gesamtberechnung des „Loss“ innerhalb des Templates wird überprüft ob dieser „Loss“ kleiner als der bisherig beste „Loss“ ist. Dann wird das Referenzbild wieder im großen Bild bewegt und der nächste „Loss“ berechnet. Das Referenzbild darf dabei maximal nur bis zur Größe des Bildes abzüglich der Größe des Referenzbildes geschoben werden. Sobald das Referenzbild in der unteren rechten Ecke angekommen ist, wird aus der Methode rausgesprungen, der beste „Loss“ wird dabei zurückgegeben. Bei der Indexierung der absoluten Position der Pixel im Template gab es zuerst einige Probleme, weil die Adressierung im Template natürlich anders funktioniert als im Gesamtbild. Im Gesamtbild muss der Startpunkt des Templates, die Breite des Gesamtbildes und die Verschiebung der Pixel im Template selbst mit einbezogen werden. Bis hierhin stand der gesamte Code in einer „cpp-Datei“, deshalb wird eine Modularisierung mit verschiedenen Headerdateien, cpp-Dateien und Methoden vorgenommen. Jede größere Funktionalität steht jetzt in einer eigenen Datei (und hat die entsprechende Header-Datei dazu) oder zumindest in einer eigenen Methode. Beim Laden des Referenzbildes von außen (also als Bild), gab es Probleme mit dem besten berechneten „Loss“. Dieser betrug nämlich nicht wie erwartet 0, sondern lag bei ca. 80.000 Grauwerten, was vergleichsweise immer noch gering ist, aber eine große Differenz zur 0 darstellt. Die Lösung dazu wird ein paar Zeilen später besprochen. Der nächste Schritt ist jetzt eine Zeitmessung einzubauen und dann die Parallelisierung mit OpenMP. OpenMP muss erst in Visual Studio für den Visual Studio C++ Compiler in den Projekteinstellungen für die entsprechenden Modi (Debug-Modus, Release-Modus, 64-Bit, 32-Bit) die verwendet werden, aktiviert werden. Nun kann man den parallelen Sektionsteil mit einer beliebigen selbst angegebenen Anzahl von Threads definieren. In diesem Teil werden verschiedene Templates für dasselbe Bild durchlaufen und der beste „Loss“ wird in ein Array geschrieben. Nach der parallelen Sektion wird das Array durchlaufen und geschaut, was dort wiederum der beste „Loss“ ist. Wenn man die einzelnen Aufrufe des BruteForce-Algorithmus einfach in die parallele Sektion schreibt, ruft jeder Thread von OpenMP jede Methode auf. Das ist natürlich sinnlos und bietet keine Performance-Verbesserungen, ganz im Gegenteil, der Overhead sorgt dafür, dass es noch länger braucht. Jeder Methodenaufruf des BruteForce-Algorithmus muss also als weiterer paralleler Sektionsteil definiert werden, damit weiß OpenMP, jedem Thread wird genau ein Methodenaufruf zugeordnet. Eine weitere Möglichkeit ist, eine „For-Schleife“ zu durchlaufen, die innerhalb den BruteForce-Algorithmus aufruft. Hier wurde zur Einfachheit immer dasselbe Template aufgerufen (Für Performancemessungen trotzdem genauso gut geeignet). Zur besseren Visualisierung des gefundenen Bildausschnitts soll ein neues Bild mit einer Umrahmung um diesen rausgeschrieben werden. Die erste Überlegung war, die Umrahmung als Overlay über das Bild zu zeichnen. Das würde allerdings zu viel Zeit erfordern, deutlich effizienter ist es, die entsprechenden Randpixel des Referenzbildes in eine markante Farbe (rot) abzuändern. Dazu werden die RGB-Werte in die Stellen im Img-Array geschrieben, richtig wäre es allerdings ein neues Img-Array zum Rausschreiben zu allokieren, da der nächste BruteForce-Aufruf jetzt mit dem veränderten Bild arbeitet und die Ergebnisse dadurch natürlich verändert sind. Das „Int-Array“ des Image muss danach in ein „unsigned char Array“ umgewandelt werden und kann dann mit „stbi\_write“ als Bild ins Filesystem geschrieben werden. Bei der Indexierung des „Int-Arrays“ muss man hierbei aufpassen, da die letzten Punkte in X- und Y-Richtung im Array eine Stelle weiter vorne stehen (Templatebreite bzw. Templatehöhe -1). Zurück zum Laden des Templates: Der ausgeschnittene Teil des Bildes (Template) verwendet ein anderes Subsampling-Verhältnis (Datenreduktion) als das große Bild (Vergleich der Bilder mit IrfanView), der richtige Bildausschnitt wird aber dennoch gefunden (siehe Output-Bild mit Umrahmung).

Der letzte große Ansatz ist die Nutzung von OpenCL und der GPU: OpenCL muss zuerst in das Visual-Studio Projekt eingebunden werden. Jetzt das richtige Device auswählen und dann kann die Pipeline für die GPU-Behandlung programmiert werden, dazu erst einmal die CL-Datei laden, die zum aktuellen Zeitpunkt nur den BruteForce-Algorithmus mit noch allen „For-Schleifen“ beinhaltet (Pipeline soll erstmal funktionieren). Anschließend kommt der typische Ablauf für die Erstellung einer GPU-Anwendung mit OpenCL: Erstellung eines Kontexts und der Kommandowarteschleife, das Programm (der Kernel) wird geladen und „gebaut“. Die Puffer müssen mit den entsprechenden Größen für die Arrays mit den RGB-Werten erstellt werden und die Arrays müssen in die Puffer geschrieben werden. Jetzt müssen noch die Parameter an den Kernel übergeben werden und der Kernel ausgeführt werden mit der richtigen „Global Size“. Falls man die berechneten Informationen wieder auf dem Host haben möchte, liest man die Buffer der GPU aus und gibt am Ende alle Ressourcen wieder frei. In den Puffer kann nicht geschrieben werden, da es eine Zugriffsverletzung gibt. Nach langem Suchen wird das Problem mit dem zweidimensionalen Allokieren des „Int-Arrays“ gefunden. Weil die Pointer auf verschiedene Speicherbereiche zeigen, die nicht zusammenhängen, kann OpenCL nicht die Größe des Gesamtspeicherbereichs erkennen. Doch der nächste Fehler tritt auf: Es kann nicht aus dem Puffer gelesen werden. Alle auszulesenden Speicherbereiche müssen im Kernel nämlich als „\_\_global“ definiert werden, zusätzlich müssen Speicherbereiche immer mit „malloc“ angelegt werden. Wenn man die C++-Syntax verwendet um Arrays zu erstellen, wird es nicht von OpenCL erkannt. Die Pipeline funktioniert jetzt soweit, die „For-Schleifen“ müssen jetzt durch „get\_global\_id“ bzw. vielleicht sogar auch durch „get\_local\_id“ ersetzt werden. Dazu muss das „Global Size“ Array die richtigen Dimensionen, bzw. „Workdimensions“ für die „Laufvariablen“ haben. 4 Dimensionen sind leider nicht möglich, der Kernel muss bei der Adressierung also mit 3 „For-Schleifen“ auskommen. Weiteres Problem: Referenzbild läuft mit Sprüngen, also geht beispielsweise um 5 in X- und Y-Richtung. Abbildung dieser Sprünge in weiterem Array-Puffer auf der GPU, weil „get\_global\_id“ kann keine Sprünge abbilden. Einfache Möglichkeit, die die beiden inneren „For-Schleifen“ ersetzt funktioniert, hierbei wird allerdings nur der Loss zu einer bestimmten Position des Referenzbildes berechnet. Mit 3 „For-Schleifen“ deutlich problematischer, vor allem, weil der Loss nach jeder Positionsveränderung des Templates zurückgesetzt werden muss. Da aber alles parallel aufgerufen wird, ist es nicht wirklich kontrollierbar wann zurückgesetzt wird. Versuch mit if-Abfragen etc., aber leider kein Erfolg. Ungefährer Zeitaufwand für die GPU bis hierhin: 14 Stunden.

Dieser Ansatz war aber zu überambitioniert, es ergibt sich auch schon eine enorme Performanceverbesserung, wenn nur die verschiedenen Positionen des Templates parallel berechnet werden und die beiden inneren For-Schleifen (Mustervergleich) erhalten bleiben. Die Sprünge werden weiterhin durch einen Array-Puffer abgebildet und der innere Teil bleibt soweit gleich wie beim originalen Algorithmus. Die „Losses“ jeder Template-Position werden in einen Puffer geschrieben und auf der CPU wird dann der beste „Loss“ überprüft und ausgegeben. Jetzt kann das „Profiling“ entsprechend der OpenCL-Struktur programmiert werden. Dabei darf das Attribut „CL\_QUEUE\_PROFILING\_ENABLE“ in der „Commandqueue“ nicht vergessen werden. Sowohl die Transferzeit der Daten auf die GPU und zurück auf die CPU, als auch die Berechnungszeit des Algorithmus kann jetzt ausgegeben werden. Es ist hier schon eine enorme Performanceverbesserung zu erkennen (siehe nächster Abschnitt für Details).

Die Implementierung für das Rausschreiben des Bildes mit der roten Umrahmung für die Templates wird jetzt auch für die GPU geschrieben. Dabei wird zugleich auch der bereits angesprochene Fehler behoben, dass in dasselbe Bild geschrieben wird, das später wieder für weitere Algorithmusaufrufe verwendet wird. Es wird ein weiteres Array allokiert und die Werte des originalen Bildes werden hier reingeschrieben. Bei der Indexierung muss man hier natürlich etwas aufpassen, da das Bild-Array bei der GPU-Implementierung eindimensional ist. Die rote Umrahmung wird jetzt in das neue Array geschrieben, was extra nur für die Ausgabe erstellt wurde, für den Algorithmus wird immer das unveränderte Bild genommen. Letztendlich wird das veränderte Bild mit den Umrahmungen erfolgreich in das Dateisystem geschrieben. Derselbe Fehler wird jetzt auch noch in der normalen Implementierung vom Template Matching verbessert.

Es kann noch eine verbesserte Kernelfunktion geschrieben werden, indem man die Anfangsadressen der Zeilen im Template und im entsprechenden Bildausschnitt jeweils nur innerhalb der äußeren For-Schleife berechnet und sich damit die Multiplikationen in der inneren For-Schleife erspart. Dadurch werden noch bessere Laufzeiten erreicht (siehe nächster Abschnitt für Details).

Eine weitere mögliche Verbesserung wäre die Reduktion der **beiden** For-Schleifen in **eine** For-Schleife. Im Template kann hier einfach immer um eins hochgezählt werden, während im Bild abgefragt werden muss, wenn man am Ende der Zeile im Template angekommen ist. Dann wird der Index für das Bild wieder auf den Anfang der darauffolgenden Zeile im Template gesetzt. Diese Kernelfunktion ist allerdings sogar langsamer als die originale Kernelfunktion (siehe nächster Abschnitt für Details).

Schlussendlich werden noch überflüssige Codeteile zur besseren Lesbarkeit entfernt und einige Kommentare zur besseren Struktur des Codes ergänzt.

Der Code wurde fortlaufend auf das GitLab der HFT gepusht, dazu musste das dort erstellte Repository mit dem lokal erstellten Repository in Visual Studio 2019 verknüpft werden.

**Bewertung des Ansatzes und der performance-limitierenden Faktoren**

Wie bereits angesprochen eignet sich das Template Matching ziemlich gut zur Performance-Optimierung, da es viele Indexierungen von Arrays und Addition und Subtraktion verwendet. Das kann man auch daran erkennen, dass die Optimierung vom GCC-Compiler mit –O3 einen extremen Performance-Unterschied mit sich bringt. Der sequentielle Durchlauf mit 1 Template dauert beim Visual C++ Compiler mit maximaler Optimierung im Release-Modus ca. 0,10759s (ca. Median), mit dem GCC-Compiler ohne –O3 Optimierung dauert es natürlich deutlich länger (keine Optimierungen), mit –O3 dauert es nur ca.0,08s. Hier wird zum Beispiel SIMD unter der Haube verwendet, mehrere Wertpaare werden also gleichzeitig in die CPU geladen und dann wird nur einmal die Operation auf alle Werte ausgeführt und nicht jedes mal aufs Neue für jedes Wertpaar. Die Optimierung vom Visual C++ Compiler scheint in diesem Fall einfach nicht so effizient zu sein. Dieser Performance-Test fand noch vor der Modularisierung der Funktionalitäten statt.

Der Ansatz ist außerdem gut parallelisierbar, vor allem mit der angesprochenen Erweiterung von mehreren Templates die durch ein Bild oder auch mehrere Bilder laufen. Zusätzlich wäre es noch möglich, den Bereich eines Bildes in mehrere Teile zu Teilen (zum Beispiel 4) und dann in jedem Bereich genau einen Thread nach der besten Übereinstimmung suchen lassen. Aus diesen besten Übereinstimmungen muss dann natürlich noch die beste entnommen werden. Problematisch ist dabei natürlich, dass der gesuchte Bildausschnitt auch genau in der Mitte zwischen zwei Bereichen liegen kann, das Referenzbild muss also immer noch zur Hälfte in den nächsten Bereich „reinfahren“.

Dennoch gibt es irgendwann eine Limitierung der Performance, was die Zeitmessungen in Bezug auf die Parallelität zeigen. Hier werden parallel mehrere Templates durch ein Bild „geschoben“, die Zeitmessungen finden, wenn nicht angegeben im Release-Modus mit dem Visual C++ Compiler statt. Die Zeitmessungen inkludieren zusätzlich den Durchlauf durch das Array mit den verschiedenen besten „Losses“ der Referenzbilder und die beste Auswahl daraus und deren Ausgabe. Dieser Wert ist aber vergleichsweise gering und wird sequentiell durchlaufen (ca. 0,00047s bei 8 Templates). Die Veränderung der Umrahmungspixel für den gefundenen Bildausschnitt im Array des Bildes wird nach dem BruteForce-Algorithmus auch einmal aufgerufen ist aber vergleichsweise auch sehr gering und es handelt sich immer um eine ähnliche Zeit (im Bereich 10-5s):

**Laufzeitmessungen** **OpenMP**  
(alle Werte in Sekunden)

|  |  |  |  |
| --- | --- | --- | --- |
| **Anzahl Threads und Anzahl Templates:** | **Zeit für Sektionen werden selbst hintereinander definiert:** | **Zeit für Parallele “For-Schleife” wird definiert (OpenMP kümmert sich selbst um Behandlung):** | **Zeit für sequentielle Ausführung (immer 1 Thread):** |
| 2 | 0,085726 | 0,117 | 0,214338 |
| 4 | 0,101572 | 0,133 | 0,419883 |
| 6 | 0,112834 | 0,133 | 0,635738 |
| 8 | 0,130668  (ca. 4,4s im Debug-Modus wenn OpenMP nicht aktiviert ist, wenn aktiviert ca. 0,7s) | 0,137 | 0,838645 |
| 16 | 0,185 | 0,177 | 1,67309 |
| 32 | 0,390 | 0,352 | 3,34336 |
| 64 | 0,780 | 0,700 | 6,6823 |
| 128 | 1,562 | 1,4 | 13,4266 |
| 256 | 3,119 | 2,8 | 26,7324 |
| 512 | 6,242 | 5,6 | 53,6993 |

Diese Zeitmessungen schwanken deutlich (6 Threads mit 6 Templates auszuführen, ist in seltenen Fällen auch mal besser als 4 Threads mit 4 Templates auszuführen). Im Mittel stimmt die Zeit aber meistens so wie es angegeben wurde, es hat also einen proportionalen Verlauf. Der Overhead wird nämlich mit mehr Threads für OpenMP größer. Wenn mehr Threads verwendet werden, als es Templates zu bearbeiten gibt, steigt die Zeit für die Ausführung, denn der Overhead wird auch hier größer, nur können die zusätzlichen Threads keine Aufgaben übernehmen und sind untätig. Wenn es weniger Threads als Templates gibt, muss sich mindestens ein Thread um 2 Templates kümmern und die Ausführungszeit steigt, da immer auf den letzten Thread gewartet werden muss.

Die Tabelle zeigt, dass der Overhead für parallele „For-Schleifen“ zwar anfangs größer ist als die eigenen definierten Sektionen und die Ausführungszeit somit größer ist, aber ab 16 Threads wird die Ausführungszeit geringer, das heißt für sehr viele Threads lohnt sich dieser Ansatz. Bei der sequentiellen Ausführung verdoppelt sich die Zeit immer mit verdoppelter Templatezahl. Bei der mitsteigenden Anzahl der Threads wird die Ausführungzeit immer nur minimal durch den Overhead größer, die verschiedenen Threads müssen ja irgendwie „gehandelt“ werden. Von 8 auf 16 Threads gibt es bei der Ausführungszeit aber bei beiden parallelen Möglichkeiten einen Sprung, denn der PC auf dem das Programm getestet wurde, hat „nur“ 8 physische CPU-Kerne. Hier kann die Parallelität dann nicht mehr so gut ausgenutzt werden, aber die Laufzeit verdoppelt sich dennoch nicht, weil Hyperthreading genutzt werden kann, da 16 virtuelle Kerne existieren. Ab 16 Threads wird keine Performance-Verbesserung mehr durch Parallelisierung erreicht, da keine weiteren virtuellen Kerne zur Verfügung stehen. Die Laufzeit verdoppelt sich jeweils, wenn die Templateanzahl verdoppelt wird, also erreicht man dasselbe Ergebnis wie bei einer sequentiellen Ausführung. Durch CPU-Scheduling wird zwar möglicherweise eine minimal bessere Durchlaufzeit erreicht, diese gleicht sich aber wieder durch den Overhead der Threads aus.

OpenMP hat für seine Threads zusätzliche interne Optimierungen, denn der sequentielle Durchlauf mit 1 Template (0,10759s) dauert länger, als wenn man 1 Template mit 1 Thread von OpenMP durchlaufen lässt (0,0875941s).

Eine zusätzliche mögliche Verbesserung für den BruteForce-Algorithmus wäre auf jeden Fall abzubrechen und aus der Methode rauszuspringen sobald der aktuell beste „Loss“ 0 beträgt, denn dann kann kein besseres Ergebnis mehr folgen (höchstens dasselbe, aber sehr unwahrscheinlich). Damit wäre die Zeitmessung allerdings nicht mehr wirklich vergleichbar, denn je nachdem was für ein Referenzbild genutzt wird und ab wann (wenn überhaupt) der „Loss“ 0 beträgt, kann schneller aus der Methode rausgesprungen werden und die Laufzeit wird verkürzt. Bei realen Problemen ist ein Loss von „0“ sowieso praktisch ausgeschlossen.

Die GPU führt den Algorithmus bezüglich der Laufzeiten durch die extreme Parallelität nochmal um ein vielfaches schneller aus, obwohl die Aufteilung und die Speicherhierarchie noch lange nicht komplett ausgenutzt wurden.

Auf der GPU sehen die Laufzeiten für die verschiedenen Kernelfunktionen folgendermaßen aus:

**Laufzeitmessungen GPU**(alle Werte in Sekunden)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **template\_match** | **template\_match\_optimized** | **template\_match\_optimized2** |
| Total time: | 0,00195824  (40mal schneller als optimiert sequentiell) | 0,00171219  (47mal schneller als optimiert sequentiell) | 0,00251312  (32mal schneller als optimiert sequentiell) |
| Calculation time: | 0,00179373 | 0,00154720 | 0,00234909 |
| Transfer time: | 0,00016451 | 0,00016499 | 0,00016403 |

Die Kernelfunktion „template\_match“ (als erstes programmiert) hat eine ca. 40mal schnellere Laufzeit als der sequentielle Durchlauf mit der –O3 Optimierung, wenn man die Transferzeit der Daten in die Puffer der GPU und aus den Puffern der GPU mit einberechnet. Die reine Berechnungszeit geht sogar noch etwas schneller. Bei der ersten optimierten Kernelfunktion „template\_match\_optimized“ ist die Laufzeit sogar nochmal kürzer, durch Optimierungen in der Adressierung. Die Gesamtlaufzeit ist fast 50mal schneller als wenn man es sequentiell optimiert durchführt. Die zweite Optimierung der Kernelfunktion funktioniert allerdings interessanterweise nicht mehr so gut. Diese Optimierung ist „nur noch“ ca. 30mal schneller als die optimierte sequentielle Version.   
Woran könnte das liegen? Eine plausible Vermutung wäre, dass **eine einzige** For-Schleife mit mehr Durchläufen in den weiteren Ebenen der GPU nicht mehr so gut aufgeteilt/parallelisiert werden kann. Mit der Parallelisierung der Positionen für das Template werden die „Compute Units“ der GPU genutzt. Wenn man jetzt intern zwei weitere For-Schleifen hat, wird die äußere For-Schleife zum Beispiel auf die „Processing Elements“ aufgeteilt. Bei **einer** For-Schleife ist das aber vielleicht nicht mehr so gut möglich, weil es jetzt 30.000 statt 10.000 Durchläufe (mit zwei Schleifen) sind und diese nicht so gut auf die „Processing Elements“ jeder benutzten „Compute Unit“ aufgeteilt werden können. Es sind beispielsweise einfach zu viele Durchläufe zum Parallelisieren pro „Compute Unit“ oder es gibt sogar noch eine weitere, tiefere Ebene an Elementen in der GPU, die durch zwei For-Schleifen natürlich einfacher auszunutzen ist bezüglich Parallelisierung, da hier die Ebenen quasi schon durch die Verschachtelung der For-Schleifen gegeben sind.

**Erkentnisse, Fazit, nächste Schritte**

Wie man bereits gesehen hat, ist das Template Matching ein Verfahren, dass man sowohl sequentiell (Indexierung, SIMD) gut optimieren kann, als auch in der parallelen Ausführung über OpenMP oder sogar über OpenCL auf der GPU. Die Performanceoptimierungen sind größtenteils lediglich über die Hardware begrenzt. Dieses Verfahren erfordert allerdings auch eine extrem hohe Optimierung, da es sonst viel zu langsam wäre und damit nicht brauchbar im breiten Einsatz und im industriellen Umfeld.

Wie könnte man jetzt weiter verfahren? Einserseits wäre eine weitere Performanceoptimierung auf der GPU noch möglich, indem man die „local size“ zusätzlich zur „global size“ verwendet. Hiermit können die „Work-items“ (auch „Processing Elements“ genannt) in sogenannte „Work-groups“ (auch „Compute Unit“ genannt) eingeteilt werden (dazu muss eine „Blocksize“ definiert werden). Damit wird die Speicherhierarchie deutlich besser ausgenutzt, denn in diesen „Work-groups“ gibt es wieder lokale Speicherbereiche („Local Memory“), auf die um einiges schneller zugegriffen werden kann. Die Laufzeit kann aber nur nochmal deutlich verbessert werden, wenn die Kernelfunktion entsprechend abgeändert wird. Die innere For-Schleife oder die inneren For-Schleifen müssen dann nämlich anders aufgeteilt werden.

Des Weiteren gibt es den bereits angesprochenen Punkt mit der Bildaufteilung und der anschließenden Threadzuweisung zu jedem Teilbild. Und um Objekte noch besser identifizieren zu können, egal ob sie auf dem Bild gedreht sind oder nicht: Man könnte innerhalb des Referenzbildes eine Rotation der Pixel durchführen und damit jede Art von Drehung im Bild finden.

Ansonsten wäre dieses Projekt vom Grundprinzip schon einsatzbereit, logischerweise müsste es dann noch mit deutlich mehr Daten (Bildern und Referenzbildern) getestet werden. Genau aus diesem Grund fand ich dieses Projekt aber auch so interessant, denn es kann in realen Fällen tatsächlich so verwendet werden.

Anmerkung:   
Die Begriffe „Template“ bzw. „Referenzbild“ werden in dieser Dokumentation synonym verwendet, damit ist einfach die Schablone gemeint, die sich über das Bild bewegt.

Link zu Repositories (GPU Implementierung getrennt):

Normale Implementierung:

<https://gitlab.rz.hft-stuttgart.de/92absi1bif/hpc-template-matching>

GPU-Implementierung:

<https://gitlab.rz.hft-stuttgart.de/92absi1bif/hpc-template-matching-gpu>

**Tätigkeitsnachweis**

**Gesamtdauer für das Projekt: 50 Stunden**

Tätigkeit: Größtenteils Programmierung, weiterer großer Teil Dokumentation

Datum: 15.11.2021 – 13.01.2022