دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر
پروژه امتیازی ساختار و زبان کامپیوتر

عنوان:

موازی‌سازی ضرب ماتریس در اسمبلی x86

Optimizing Matrix Multiplication Using x86 SIMD

نگارش

محمدپارسا جعفرنژادی، پوریا رحمانی، محمدمهدی عابدینی، نیما قدیرنیا

بهمن ۱۴۰۳

۱ مقدمه

در این پروژه ما سعی کرده‌ایم تا عمل ضرب ماتریس را با استفاده از SIMD در اسمبلی x86_64 پیاده‌سازی کنیم و زمان اجرای این الگوریتم را با پیاده‌سازی‌های عادی در C و اسمبلی مقایسه کنیم.

۲ گام‌های پروژه

۲-۱ برنامه‌نویسی کد C

در ابتدا الگوریتم ضرب ماتریس را به صورت عادی در زبان C پیاده‌سازی می‌کنیم. این الگوریتم به ازای هر خانه از ماتریس مقصد، مقدار آن را با ضرب داخلی ستون‌ها و سطرهای دو ماتریس ورودی، به دست می‌آورد. این الگوریتم درون تابعی با فرمت زیر پیاده‌سازی می‌شود:

```
void matrix_mul(int* m1, int* m2, int* dest, int n)
```

که در آن m1, m2 ماتریس‌های ورودی،dest پوینتر ماتریس خروجی و n سایز ماتریس‌ها می‌باشد.

۲-۲ برنامه‌نویسی الگوریتم عادی در زبان اسمبلی x86_64

حال همان الگوریتم ساده را در زبان اسمبلی x86 پیاده‌سازی می‌کنیم. این کد باید مانند بخش قبلی تابع `matrix_mul` را پیاده‌سازی کند.

۲-۳ برنامه‌نویسی الگوریتم موازی با استفاده از دستورات SIMD

در آخر به بخش اصلی پروژه میرسیم که در آن باید با استفاده از دستورات Single Instruction Multiple Data ضرب دوماتریس را بهینه سازی کنیم

با توجه به اینکه در صورت پروژه ذکر شده بود که n مضربی از ۴ است ، ما از رجیسترهای xmm استفاده کردیم که در واقع از آنجا که ۱۲۸ بیتی هستند ،توانایی ذخیره سازی ۴ عدد ۳۲ بیتی را دارا هستند.

قابلیتی که این ثبات ها به ما میدهند این است که میتوانند از روی حافظه ۱۶ بایت را به صورت متوالی بخوانند که اگر هر ۴ بایت را یک عدد بگیریم، یعنی می‌توان ۴ عدد را همزمان درون یک ثبات xmm داشت. پس با توجه به دستورات که از SIMD داریم، می‌توانیم به صورت ۴ تا ۴ اعداد را

باهم ضرب کنیم . در واقع با استفاده از دستور pmulld در واقع اگر در xmm0 ۴ عدد ۳۲ بیتی به صورت ABCD ذخیره شده باشد ، و در xmm1 ۴ عدد به صورت XYZT ، با استفاده از این دستور میتوان A*X, B*Y, C*Z, D*T را در یک رجسیتر دیگر ذخیره کرد (توجه کنید که با توجه به فرض سوال که این اعداد حاصل از ضرب ماتریس ها ۳۲ بیتی هستند ، اینکار ممکن است .)
بنابراین از آنجا که n مضرب ۴ است ، میتوان دو بردار n تایی را بدین صورت (یعنی با ۴ تا ۴ تا حرکت کردن و ضرب کردن) در هم ضرب داخلی کرد .توجه کنید که نتیجه حاصل هر ۴ ضرب را درون یک رجسیتر ثابت با دستورpaddd . مثلا در رجیستر xmm0 اضافه میکنیم و در انتها باید این ۴ بخش ۳۲ بیتی این رجیستر را باهم جمع کنیم تا حاصل نهایی ضرب داخلی بدست آید. توجه کنید که ما به جای اینکه هر بار ۴ عدد حاصل از ضرب جمع کنیم ، این کار را در انتها انجام میدهیم تا در سرعت اجرای برنامه بهبود ایجاد کند.(یعنی حتی در جمع کردن نیز به صورت موازی عمل میکنیم)
در واقع در انتها با دستور phaddd که کارش جمع کردن بخش های ۳۲ بیتی مجاور است ، میتوانیم این ۴ بخش را جمع کرده و در کل ، خروجی ضرب داخلی دوبردار را حساب کنیم.

نکته ای که این پیاده‌سازی دارد این است که از انجا که ماتریس دوم باید ستون ها یش در سطر های ماتریس اول ضرب شود ، یعنی با استفاده از دستورات لود کردن نمیتوان ۴ خانه متوالی از ستون های ماتریس دوم دسترسی پیدا کرد (چرا که فاصله هایشان n تایی است) بنابراین لازم است که ابتدا از ماتریس دوم ترانهاده گرفت و در سپس به ضرب داحلی سطر های هر دوماتریس در هم افدام کرد .

۴-۲ برنامه‌نویسی برای اندازه‌گیری زمان اجرای هر پیاده‌سازی

از آنجایی که توابع را با فرمت یکسان تعریف کرده‌ایم، کافی است که برنامه‌ای بنویسیم که به ازای هر اندازه n، ماتریس‌هایی تولید کرده و به عنوان ورودی به این تابع بدهد و اختلاف زمان قبل و بعد از اجرای آن را خروجی دهد.
برنامه `timer.c` همین کار را انجام می‌دهد. به این شکل که این برنامه را موقع کامپایل کردن باید با فایل اسمبل شده تابع‌ها Link کرد. سپس با اجرا کردن فایل کامپایل شده نهایی، می‌توان زمان اجرای هر دور تابع را به طور دقیق و جدا از بخش‌های دیگر برنامه گرفت.
این برنامه چند پارامتر اصلی دارد:

- `MIN_N`: برنامه با ماتریسی از این سایز شروع به زمان گرفتن می‌کند.
- `MAX_N`: برنامه در هر مرحله سایز ماتریس را ۴ واحد افزایش می‌دهد تا به این مقدار برسد
- `N_SAMPLES`: به ازای هر اندازه، برنامه به این تعداد تابع را صدا می‌زند و در نهایت میانگین زمان آن‌ها را به عنوان نتیجه نهایی زمان برای هر سایز در نظر می‌گیرد.

بعد از نمونه‌گیری، برنامه علاوه بر چاپ خروجی‌ها، آن‌ها را در یک فایل داده ذخیره می‌کند.

۲-۵ ثبت داده‌ها و رسم نمودار

با استفاده از کتابخانه Matplotlib در زبان پایتون این داده‌ها را می‌خوانیم و نمایش می‌دهیم.

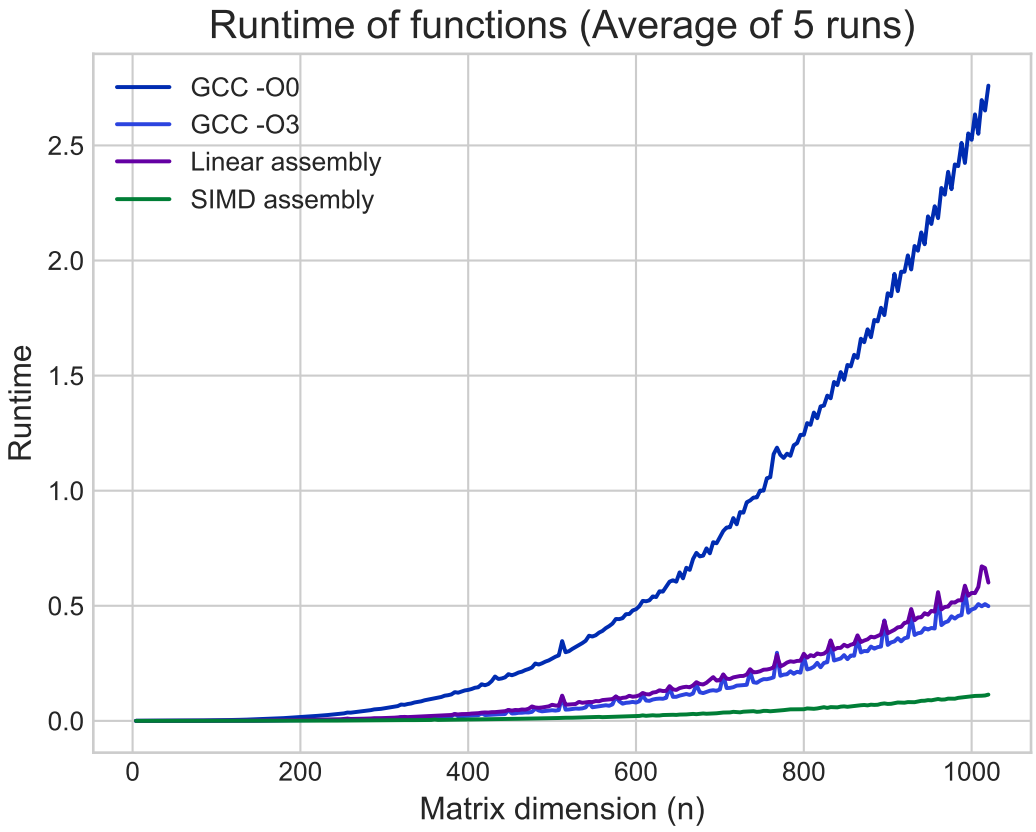
۳ نتایج

به دلیل وقت‌گیر بودن اجرا، ابتدا توابع را به ازای مقادیر ۴ الی ۱۰۲۴ تست کردیم. نمودار حاصل در شکل ۱ نمایش داده شده است. این مقادیر با اجرای برنامه بر روی WSL Ubuntu به دست آمده‌اند.

در این نمودار علاوه بر کدهای اشاره شده، یک بار هم کد سی عادی با استفاده از `gcc -O3` کامپایل شده تا با اسمبلی خالص مقایسه شود. مشاهده می‌شود که روش‌های بهینه‌سازی کامپایلر GCC تا حدی بهتر هستند. همچنین سرعت کد موازی‌سازی شده، به مقدار قابل توجهی (حدود ۵ برابر) بیشتر از پیاده‌سازی عادی در حالت کامپایل بهینه و یا کد اسمبلی می‌باشد. همچنین زمان‌ها به دلیل وقتگیر بودن اجرای بیشتر، تنها تا $n=1024$ سنجیده شده‌اند

۳-۱ مسئلهٔ Cache

همانطور که در شکل ۱ واضح است، کدهای غیرموازی به ازای مقادیری، به خصوص توان‌های ۲، دارای قله هستند. دلیل این اتفاق به طور مختصر این است که به دلیل Caching در پردازنده، هنگام دسترسی به یک داده پردازنده داده‌های مجاور آن را نیز درون حافظه نهان بارگیری می‌کند. مقدار بلوک‌های این حافظه توان‌های دو هستند و وقتی که مقدار به مرز این حجم می‌رسد، باید بلوک‌های

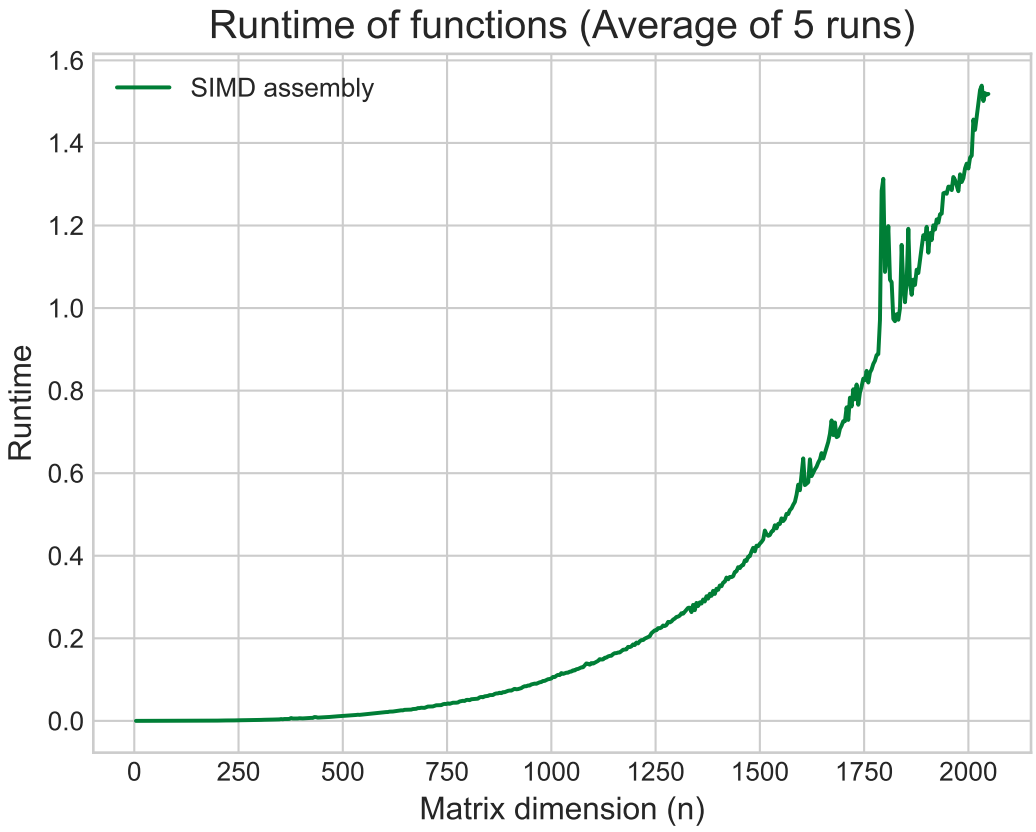


شکل ۱: نمودار میانگین زمان اجرای برنامه‌ها به ازای مضارب ۴ کوچکتر از ۱۰۲۴

بیشتری آماده شوند و به همین دلیل شاهد افزایش زمان اجرا در این مقادیر هستیم.
این مسئله در پیاده‌سازی SIMD مشهود نیست و زمان‌های این الگوریتم تقریبا پیوسته هستند. دلیل اصلی این است که به دلیل ترانهاده کردن ماتریس در ابتدا، دسترسی‌های به حافظه نزدیک به هم شده‌اند چرا که نیاز به پیمایش ستونی در هر مرحله نداریم. اما در مقادیر بالاتر طبق شکل ۲ مشاهده می‌شود که این پیوستگی به هم می‌ریزد. در اینجا دلیل می‌تواند این باشد که حجم L1 Cache پر شده و وارد سطح بعدی حافظه نهان می‌شویم که کندتر است.

۲-۳ پیاده‌سازی Cache-Efficient

همانطور که گفته شد، دلیل ضعف الگوریتم‌های عادی در اعداد خاص، حافظه نهان بود. بعد از این مشاهده متوجه شدیم که الگوریتم عادی ما از لحاظ حافظه نهان بهینه نیست. دلیل آن این است که

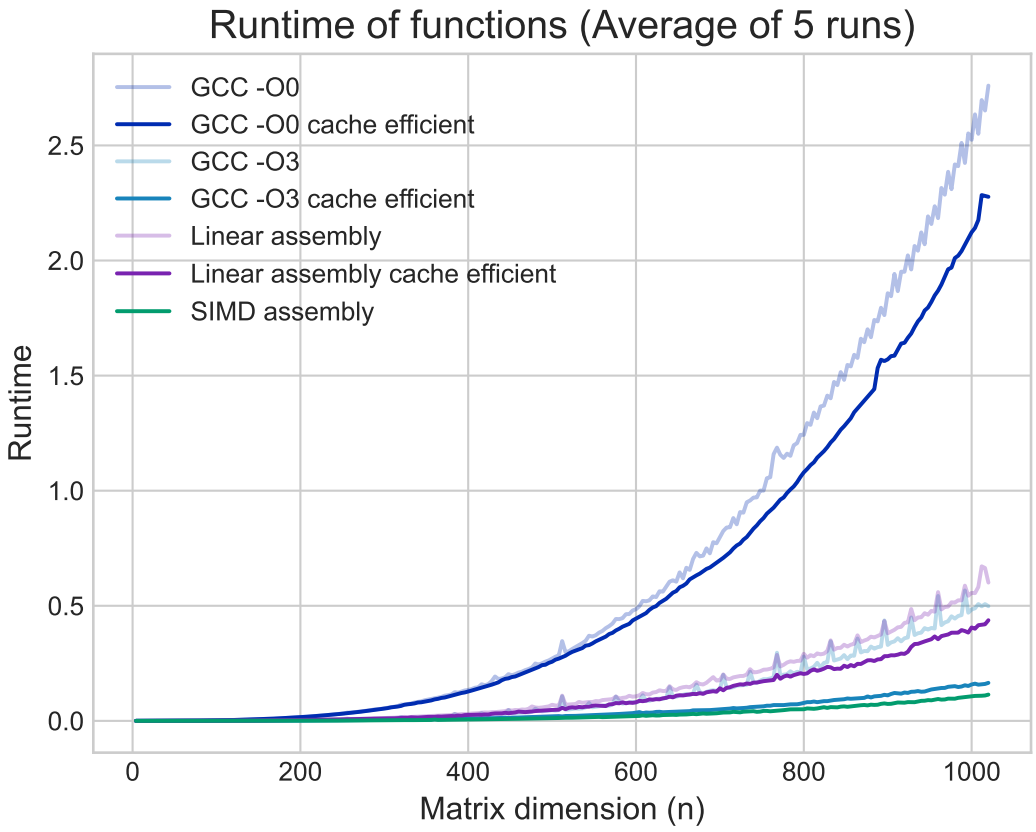


شکل ۲: نمودار میانگین زمان اجرای برنامه SIMD به ازای مقادیر کوچکتر از ۲۰۴۸

این الگوریتم به ازای هر خانه از ماتریس، کل ستون ماتریس دوم را طی می‌کند که یعنی در هر مرحله، چهار برابر اندازه ماتریس باید در حافظه به جلو برویم. به دلیل نزدیک نبودن خانه‌های ماتریس در این حالت، حافظه نهان به شکل بهینه عمل نمی‌کند. این مشکل با عوض کردن ترتیب طی کردن ماتریس‌ها حل می‌شود. به طوری که به ازای هر عضو یک ستون ماتریس دوم، ضرب آن را در اعضای سطر ماتریس اول حساب می‌کنیم و بعد از طی کردن همه آنها به عضو بعدی می‌رویم. تفاوت این پیاده سازی در جابجایی دو متغیر حلقه می‌باشد اما از لحاظ استفاده از حافظه نهان تاثیر بسزایی بر روی زمان اجرا دارد.

پس از تغییر کدهای C و اسمبلی به روش ذکر شده، نتایج زمان‌های اجرای جدید به صورت شکل ۳ می‌باشد.

در این نمودار جدید مشاهده می‌کنیم که قله‌های روی توان‌های دو و ضرایب آنها به دلیل بهینه شدن کارکرد حافظه نهان، از بین رفته‌اند. نکته قابل توجه این نمودار این است که عملکرد GCC -O3 به مقدار زیادی نزدیک به SIMD شده است. پس از disassemble کردن Object file حاصل از



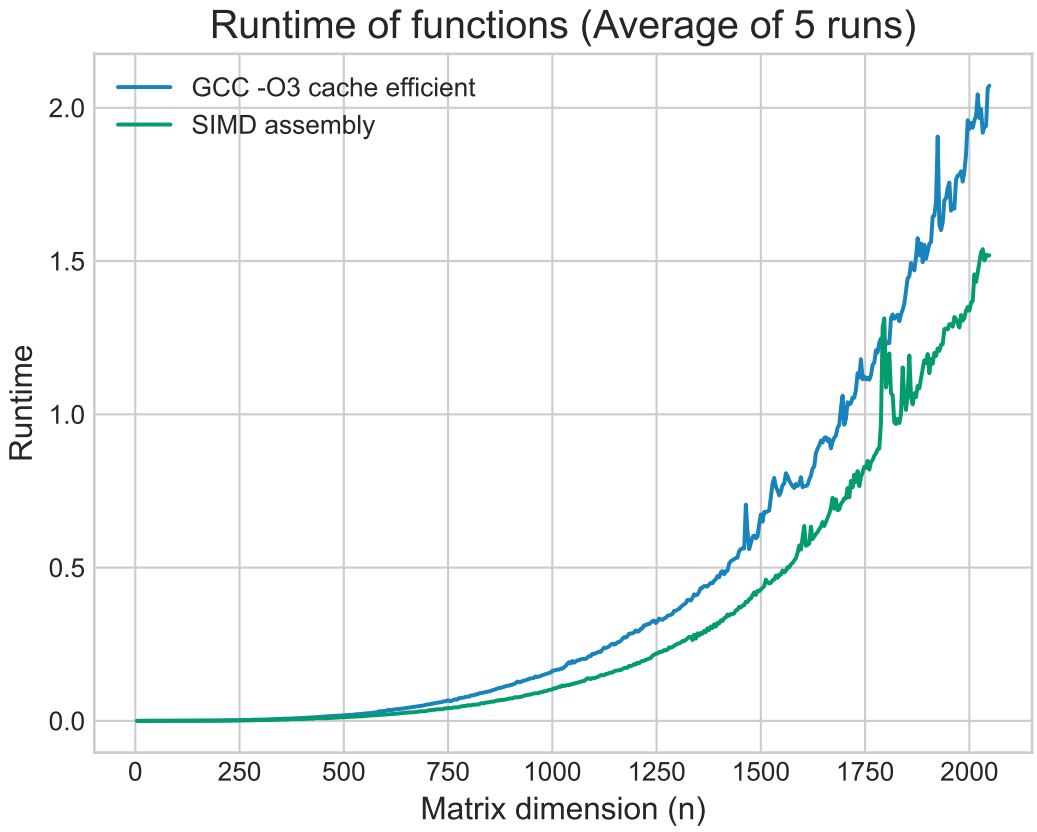
شکل ۳: نمودار میانگین زمان اجرای برنامه‌ها با الگوریتم Cache-Efficient

کامپایلر، مشاهده شد که پس از درست کردن پیمایش، کامپایلر این الگوی ضرب داخلی را تشخیص داده و از دستورات SIMD و رجیسترهای xmm برای بهینه‌سازی عملکرد استفاده کرده است.

۳-۳ مقایسه عملکرد کامپایلر و کد اسمبلی موازی

از آنجایی که زمان این دو تابع نزدیک شد، برای مقایسه دقیق‌تر هردوی آنها را تا ۲۰۴۸ مطابق شکل ۴ بررسی می‌کنیم.

مشاهده می‌شود که برنامه اسمبلی خالص به طور کلی بهتر از کامپایل بهینه GCC عمل کرده است.



شکل ۴: نمودار میانگین زمان اجرای برنامه SIMD در مقایسه با GCC -O3 به ازای مقادیر کوچکتر از ۲۰۴۸

۴-۳ مقایسه نهایی زمان‌ها به ازای مقادیر بزرگ

در جدول ۱ به ازای مقادیر بزرگتر زمان‌های برنامه‌ها نمایش داده شده است.
در جدول ۲ نیز این زمان‌ها به نسبت زمان اجرای تابع SIMD قرار داده شده اند. همانطور که دیده می‌شود به ازای مقدار ۲۰۴۸ که توانی از ۲ است، برنامه‌هایی که از لحاظ حافظه نهان نابهینه هستند افزایش زمان شدید و ناگهانی داشته‌اند که چون فقط برای مقادیر خاص اتفاق میفتد در مقایسه تاثیر نمی‌دهیم. نتایج مهم در مقادیر حول ۲۰۴۸ عبارتند از:

- سرعت برنامه SIMD حدودا ۶ برابر برنامه اسمبلی عادی و حدود دو برابر برنامه اسمبلی Cache-Efficient است.
- بهینه‌سازی کامپایلر GCC در حالت Cache-Efficient سرعتی نزدیک به پیاده‌سازی SIMD دارد اما همچنان پیاده‌سازی موازی در اسمبلی سریع‌تر است.

جدول ۱: زمان‌های میانگین ۵ اجرا برای مقادیر نهایی

برنامه \ اندازه ماتریس	2040	2044	2048
GCC -O0	31.518	37.978	103.587
GCC -O3	8.452	9.002	125.865
Assembly	8.461	9.541	138.483
Cache -O0	17.067	17.664	17.866
Cache -O3	1.957	1.960	2.049
Cache Assembly	3.541	3.514	3.585
SIMD	1.645	1.595	1.635

- در نهایت می‌توان گفت با پیاده‌سازی موازی در زبان اسمبلی، می‌توان برنامه را تقریبا ۲۰ برابر نسبت به نوشتن و کامپایل عادی آن در C سریع‌تر کرد.

جدول ۲: نسبت زمان‌ها به زمان SIMD

برنامه \ اندازه ماتریس	2040	2044	2048
GCC -O0	19.1	23.8	63.4
GCC -O3	5.1	5.6	77.0
Assembly	5.1	6.0	84.7
Cache -O0	10.4	11.1	10.9
Cache -O3	1.2	1.2	1.3
Cache Assembly	2.1	2.2	2.2
SIMD	1	1	1