南開大學
Nankai University

计算机学院

并行程序设计实验报告报告

高斯消去的 SIMD

姓名： 高祎珂

学号：2011743

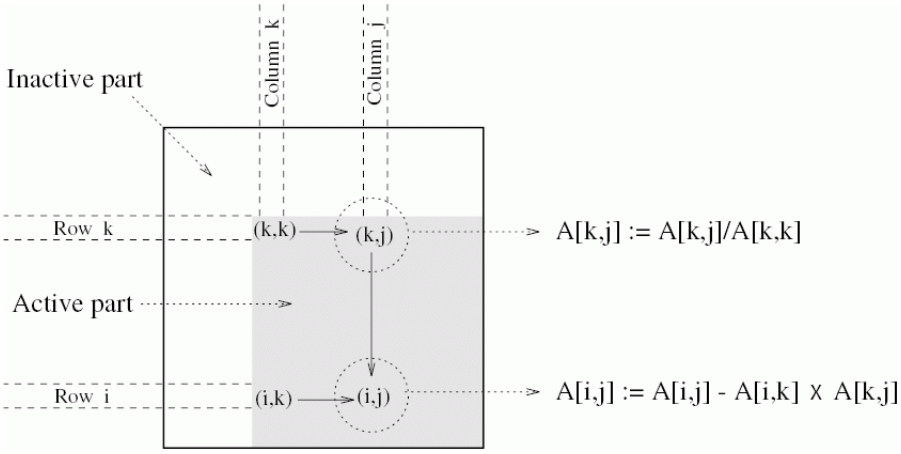
专业：计算机科学与技术

2022 年 4 月 12 日

目录

1 实验介绍

高斯消去可以简单的理解为利用一定的消去方法将矩阵化为上三角矩阵，基本运算可以分为归一和消去两步，消去过程如图5.8所示



可以有串行算法如下

Algorithm 1 普通高斯消去串行算法

Input: 普通矩阵

Output: 上三角矩阵

```
1:  procedure LU(A)
2:      k=1
3:      while k <= n do
4:          j = k + 1
5:          while j <= n do
6:              A[k, j] := A[k, j]/A[k, k];
7:          end while
8:          A[k, k] := 1.0;
9:          i = k + 1
10:         while i <= n do
11:             j = k + 1
12:             while j <= n do
13:                 A[i, j] := A[i, j] − A[i, k]×A[k, j];
14:             end while
15:             A[i, k] := 0;
16:         end while
17:     end while
18: end procedure
```

将串行算法并行化，改善上述代码中的二重、三重循环改为向量运算，即是本实验要完成的操作。

2 ARM 平台 NEON 指令集 SIMD

2.1 伪代码

并行算法的伪代码为

Algorithm 2 SIMD Intrinsic 版本的普通高斯消元

Input: 系数矩阵 A[n,n]

Output: 上三角矩阵 A[n,n]

```
1:  procedure LU(A)
2:      for i = 0 to n − 1 do
3:          vt ← dupTo4Float(A[k, k]);
4:          for j = k + 1; j + 4 <= n; j+ = 4 do
5:              vt ← load4FloatFrom(A[k, j]); // 将四个单精度浮点数从内存加载到向量寄存器
6:              va ← va/vt;// 这里是向量对位相除
7:              store4FloatTo(A[k, j], va);// 将四个单精度浮点数从向量寄存器存储到内存
8:          end for
9:          for j in 剩余所有下标 do
10:             A[k,j]=A[k,j]/A[k,k] ; // 该行结尾处有几个元素还未计算
11:         end for
12:         A[k][k] ← 1.0;
13:         for i = k + 1 to n − 1 do
14:             vaik ← dupToVector4(A[i, k]);
15:             for j = k + 1; j + 4 <= n; j+ = 4 do
16:                 vakj ← load4FloatFrom(A[k, j]);
17:                 vaij ← load4FloatFrom(A[i, j]);
18:                 vx ← vakj ∗ vaik;
19:                 vaij ← vaij − vx;
20:                 store4FloatTo(A[i, j], vaij);
21:             end for
22:             for j in 剩余所有下标 do
23:                 A[i, j] ← A[i, j] − A[k, j] ∗ A[i, k];
24:             end for
25:             A[i, k] ← 0;
26:         end for
27:     end for
28: end procedure
```

2.2 实验结果

函数的github 链接

分别进行串行，只二重循环向量化，只三重循环向量化，全部向量化的测试，记录时间如下表所示

数据规模	串行算法时间/s	二重循环向量化时间/s	三重循环向量化时间/s	全向量化算法时间/s
n=50	0.000321820	0.000319830	0.000227250	0.000225240
n=150	0.008549600	0.008537500	0.005685850	0.005651750
n=250	0.039502900	0.039449600	0.025945500	0.025899570
n=350	0.108241430	0.108104920	0.070733600	0.070553960
n=450	0.233092710	0.232708800	0.151270560	0.150558020
n=550	0.431074580	0.427994370	0.278346940	0.277028040
n=650	0.714748120	0.712125620	0.460866280	0.458571900
n=750	1.103778600	1.096142830	0.705453540	0.703000450
n=850	1.612839160	1.605957250	1.035770980	1.033609820
n=950	2.252910000	2.239647440	1.441904300	1.441328400
n=1050	3.090785860	3.078435810	1.964398240	1.961984900
n=1150	4.063603560	4.055874910	2.572261490	2.570399610
n=1250	5.223857170	5.255930520	3.317144650	3.318738170
n=1350	6.642535740	6.625041400	4.180274300	4.189128940
n=1450	8.351965170	8.353783820	5.187562510	5.190037340

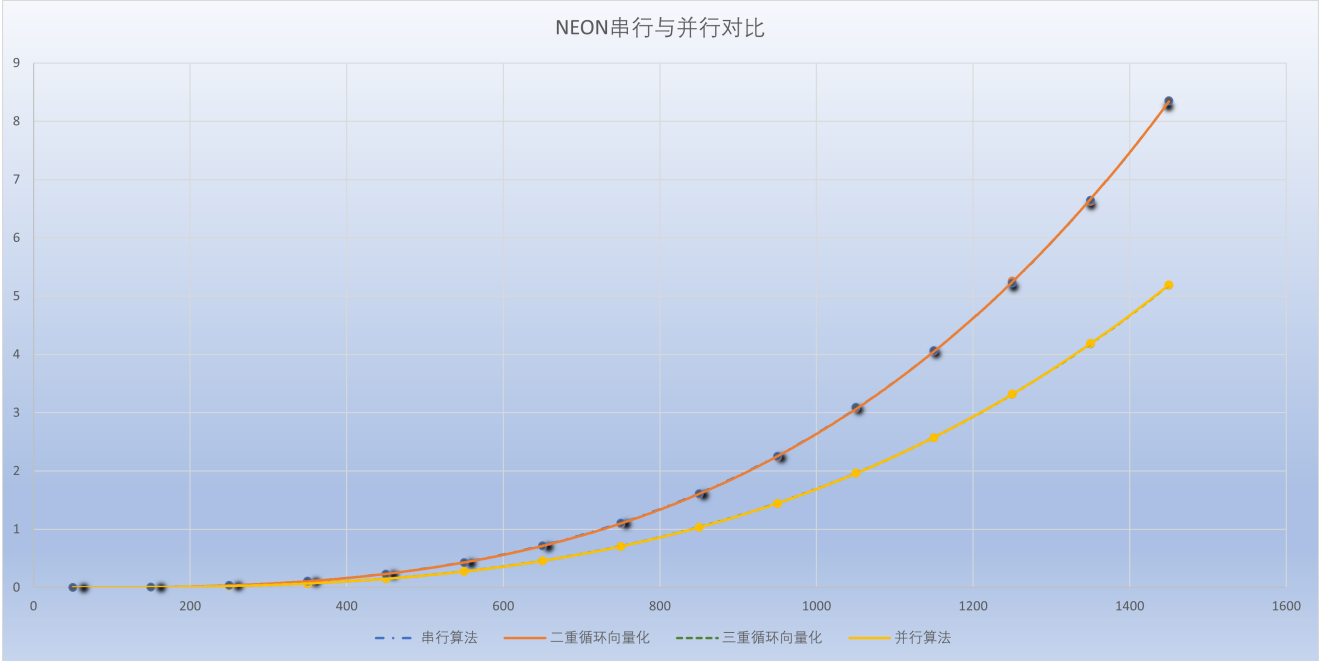


图 2.1: NEON 并行与串行对比

从图上和表中可以看出，二重循环的结果的结果并不是很理想，优化效果并不是很好，但是加入了三重循环，几乎使时间提升了将近一倍，但是根据向量化原理来看，这还没与达到理论上的优化效果，用 perf 分析一下性能。

2.3 性能剖析

Functions	instructions	cycles
串行算法	24.82%	23.47%
二重循环向量化	24.80%	23.27%
三重循环向量化	8.87%	14.81%
并行算法	8.85%	14.69%

使用 perf 工具查看指令和周期情况，如上表所示，可以看出大小顺序还是比较符合理论分析的，但是二重循环的指令和周期并没有达到理论的优化效果，而是和串行算法基本差不多。

3 X86 平台 SSE 指令集 SIMD

3.1 伪代码

算法的伪代码和 NEON 指令集的伪代码一样，此处不再进行展示，项目的代码github 链接

只不过在具体操作翻译时使用各自的指令，并且因为在指令上区别于对齐和不对齐，因此在 SSE 上还进行了是否对齐操作的性能对比。

3.2 实验结果

数据规模	串行时间/s	二重循环向量化/s	三重循环向量化/s	整体不对齐并行/s	整体对齐并行/s
50	0.0002251	0.0001759	0.0001137	0.0000733	0.0000769
150	0.0048795	0.0042116	0.0020682	0.0018364	0.0020058
250	0.0195361	0.020198	0.0097166	0.008969	0.0094646
350	0.0536685	0.0531838	0.0258799	0.0269587	0.0251379
450	0.11663	0.119576	0.0548746	0.0515381	0.055018
550	0.209064	0.202026	0.0978941	0.0976327	0.0984442
650	0.37143	0.37414	0.205088	0.186396	0.168702
750	0.680488	0.669818	0.293359	0.304942	0.277809
850	0.80512	0.827142	0.381502	0.390583	0.362997
950	1.15187	1.20233	0.508118	0.613619	0.510608
1050	1.53572	1.50281	0.674374	0.691375	0.711478
1150	1.99834	2.08319	0.902696	0.917384	0.949259
1250	2.63431	2.56631	1.18537	1.23319	1.2298
1350	3.34285	3.30319	1.54293	1.54062	1.52614
1450	4.13883	4.10596	1.87481	1.93186	1.99835

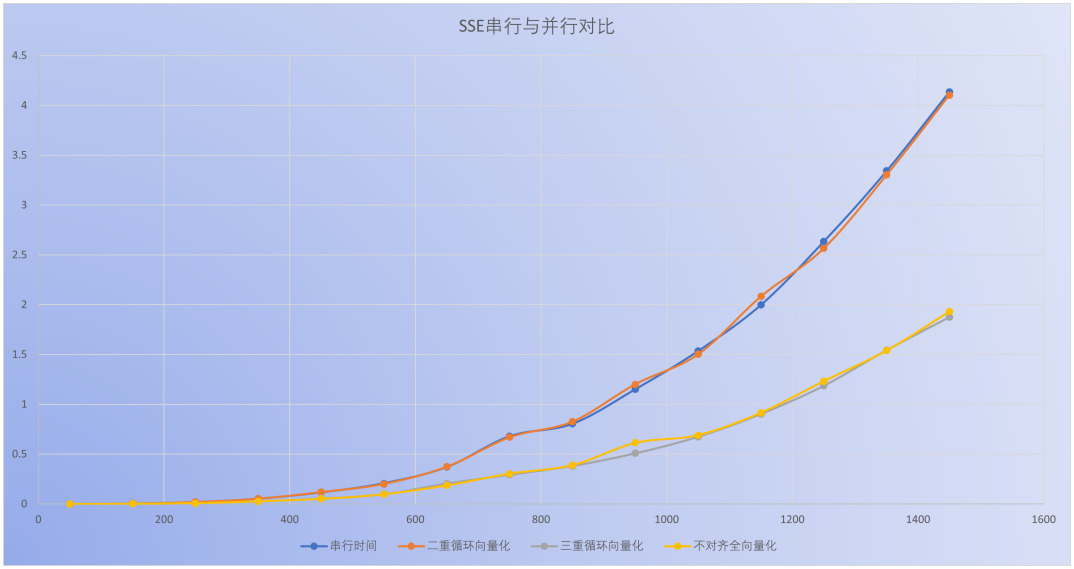


图 3.2: SSE 并行与串行对比

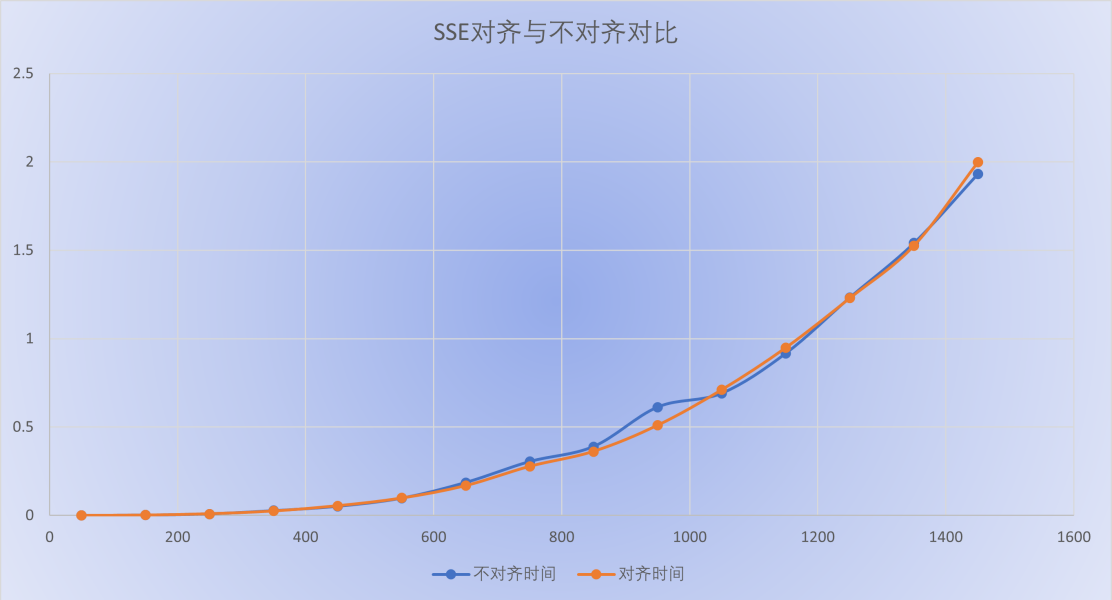


图 3.3: SSE 对齐与不对齐对比

从上图和表中可以看出表现出与 ARM 平台上 NEON 指令集同样的规律，二重循环向量优化效果并不是很好，三重循环起到了主要作用，效率可以到到串行的 3 倍左右。

而对于对齐优化，优化效果并不理想，一些甚至没有未对齐的效率高。用 VTune 工具剖析一下性能。

3.3 性能剖析

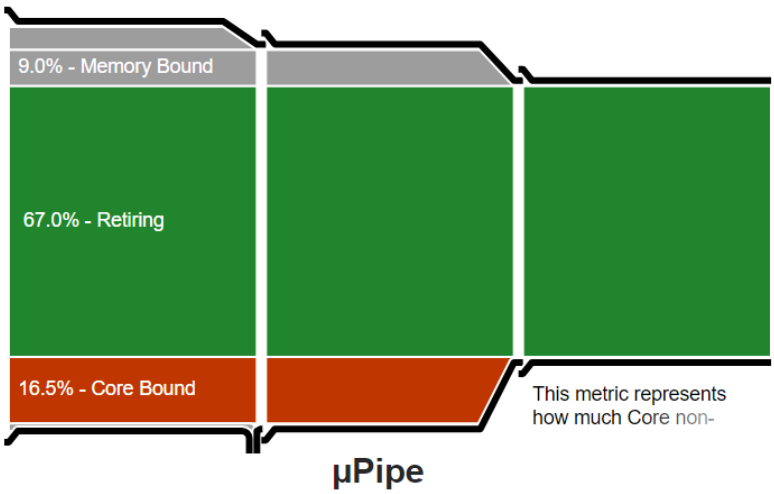


图 3.4: X86SSE 性能分析

Function	Clockticks	CPIRate	Instructions Retired
串行	59,490,600,000	0.277	214,926,400,000
二重循环向量化	60,049,600,000	0.280	214,715,800,000
三重循环向量化	17,654,000,000	0.306	57,644,600,000
不对齐并行	17,399,200,000	0.304	57,187,000,000
对齐并行	17,370,600,000	0.301	57,657,600,000

从上表中可以发现二重循环效率不高主要是因为时钟周期比较长，没有发挥出作用，而三重循环的 CPI 最大，整体表现出来效率比较好。

4 X86 平台 AVX 指令集 SIMD

4.1 伪代码

算法的伪代码和 NEON 指令集的伪代码一样，此处不再进行展示，项目的代码github 链接

只不过在具体操作翻译时使用各自的指令，并且因为在指令上区别于对齐和不对齐，因此在 AVX 上还进行了是否对齐操作的性能对比。

4.2 实验结果

数据规模	串行算法时间/s	二重循环向量化/s	三重循环向量化/s	不对齐并行/s	对齐并行/s
50	0.000155	0.0001735	6.37e-05	4.9e-05	0.0001017
150	0.004247	0.0043198	0.0011917	0.0021263	0.0016881
250	0.0218544	0.0222999	0.0052985	0.0049036	0.0059332
350	0.0586947	0.0609715	0.0157118	0.0154026	0.0151436
450	0.121661	0.12048	0.0317878	0.0329943	0.0307905
550	0.231991	0.242067	0.0558023	0.0539193	0.060859
650	0.383754	0.541514	0.0944239	0.0927	0.0965305
750	0.571349	0.568673	0.14934	0.149082	0.151766
850	1.12414	1.1694	0.30684	0.290054	0.264996
950	1.53163	1.76392	0.414625	0.446874	0.394575
1050	1.93695	1.89431	0.448062	0.447364	0.420537
1150	2.21798	2.26534	0.707945	0.560209	0.590169
1250	2.87411	2.91168	0.72046	0.749333	0.71222
1350	3.84787	3.80693	1.00459	0.981894	0.911308
1450	4.77427	4.53552	1.14372	1.15854	1.14366

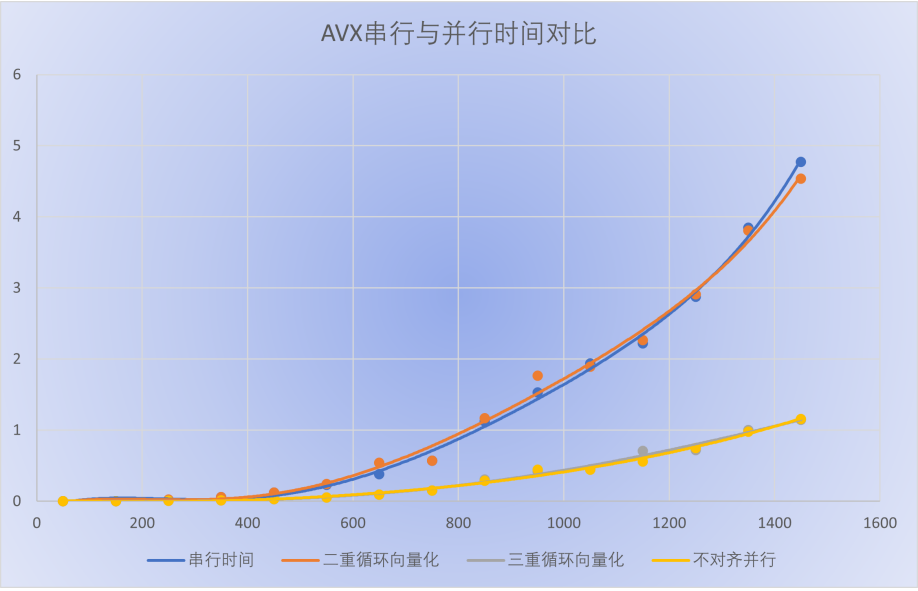


图 4.5: AVX 并行与串行对比

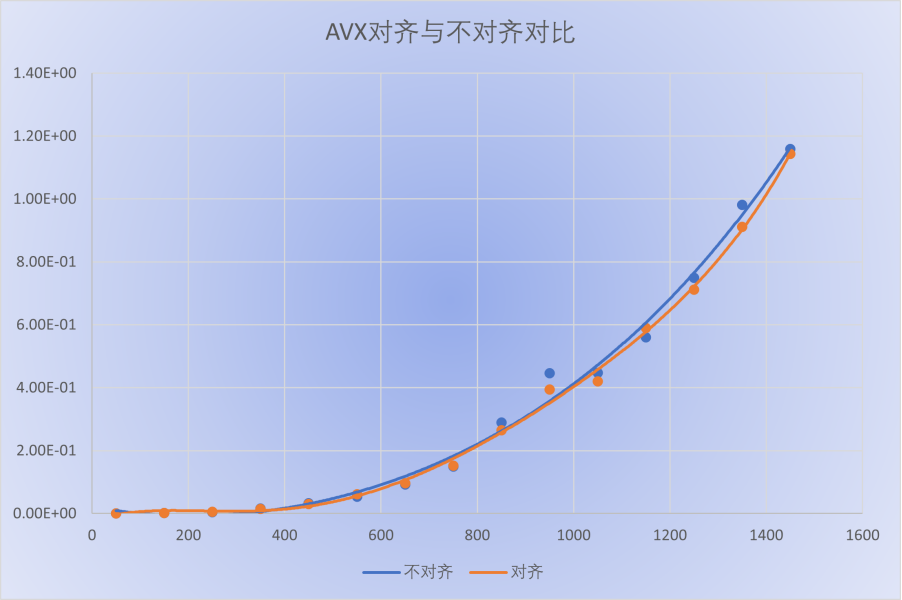


图 4.6: AVX 对齐与不对齐对比

有上图和上表可知，AVX 指令集基本保持着大规律，二重循环向量化作用不大，三重循环对优化起着关键性作用，在 AVX 上是八路向量化，在数据集较大时可以看到，效率提高了大概将近 5 倍，而对齐操作也基本上能有优化效果，虽然可能效果不显著。

4.3 性能剖析

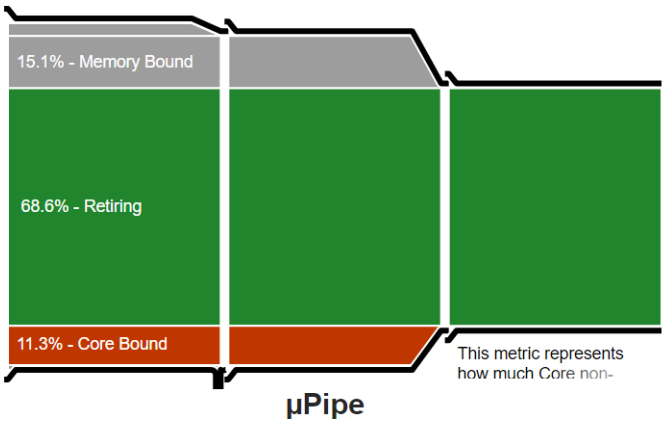


图 4.7: X86AVX 性能分析

Function	Clockticks	CPI Rate	Instructions Retired
串行算法	62,826,400,000	0.293	214,780,800,000
二重循环向量化	61,890,400,000	0.288	214,619,600,000
三重循环向量化	9,458,800,000	0.315	9,458,800,000
不对齐并行	9,032,400,000	0.306	9,032,400,000
对齐并行	9,412,000,000	0.312	9,412,000,000

从上表中可以发现二重循环效率不高主要是因为时钟周期比较长，没有发挥出作用，而三重循环的 CPI 最大，整体表现出来效率比较好，与 SSE 效率很相似。

5 NEON、SSE、AVX 对比分析

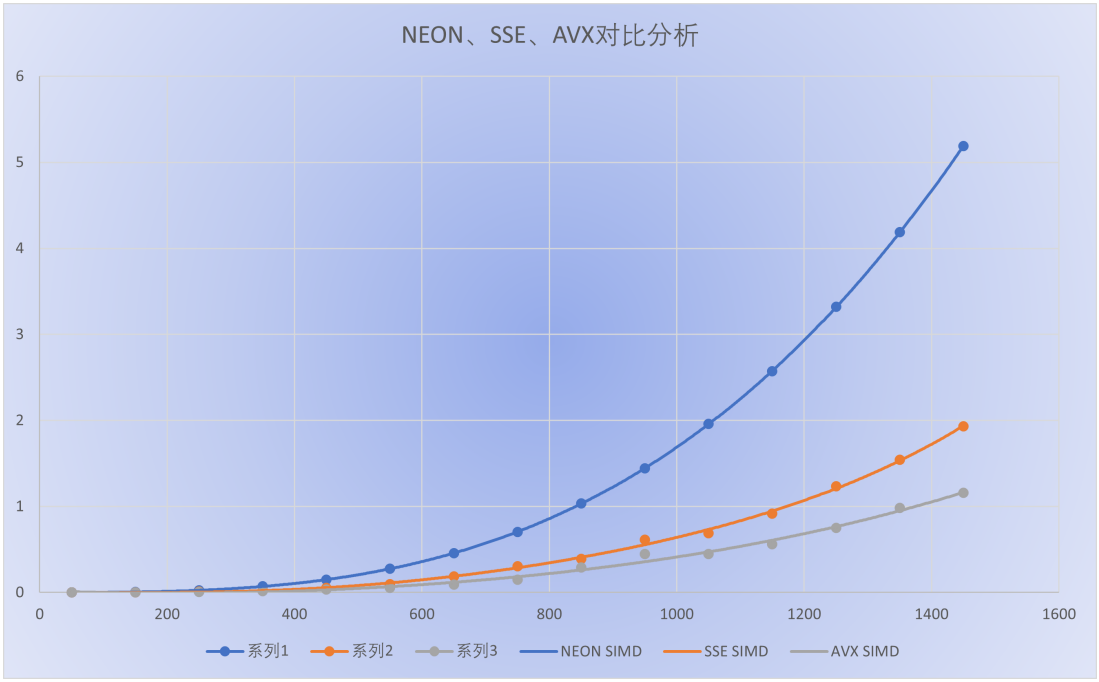


图 5.8: NEON、SSE、AVX 对比分析

从该图中可以看出 X86 平台上的 SSE 比 ARM 平台上的 NEON 四路向量化的效果好很多，本机的 L1cache 比鲲鹏服务器的大是一种原因，指令集也是原因之一。对于 X86 平台上的 AVX 指令集，这采取了 8 路向量化，但是比起 4 路向量化的 SSE，效果并没有达到他的两倍，应该是中间的消耗操作比较多，通过性能分析中间过程可以发现二者的效率。