تمرین ۱ میکروپروسسور

دانشگاه صنعتی شریف

دانشکده مهندسی برق

درس طراحی سیستم های میکروپروسسوری

گرداورنده: **محمد غفوریان**

شماره دانشجویی: ۹۹۱۰۶۴۹۳

نام استاد: دکتر موحدین

مجموعه آزمایشهای بهره گیری از قابلیتهای پردازنده‌ها

مرحله‌ی اول: پوینتر و رجیستر در زبانC

مشخصه ی لپتاپ:

Processor: Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz 2.59 GHz

Installed RAM: 16.0 GB (15.8 GB usable)

System type: 64-bit operating system, x64-based processor

۱)ابتدا زمان ضرب ها را به ازای مقادیر مختلف N به دست میاوریم.

قبل از انجام ازمایش وای فای و انتی ویروس را خاموش میکنیم که عددی نزدیک به واقعیت حساب شود.

999:

-	1	2	3	4	5	mean
Matrix_mult_0	11.746	11.810	11.587	11.756	11.734	11.726
Matrix_mult_1	5.612	5.672	5.519	5.561	5.637	5.600
Performance	2.093	2.082	2.099	2.114	2.081	2.093

P=2.093

1000:

-	1	2	3	4	5	mean
Matrix_mult_0	7.801	7.602	7.708	7.765	7.626	7.700
Matrix_mult_1	3.356	3.346	3.354	3.352	3.313	3.344
Performance	2.324	2.271	2.298	2.317	2.302	2.303

P=2.303

1001:

-	1	2	3	4	5	mean
Matrix_mult_0	10.414	10.324	10.189	10.313	10.232	10.294
Matrix_mult_1	5.483	5.382	5.226	5.262	5.331	5.336
Performance	1.899	1.918	1.950	1.960	1.919	1.929

P=1.929

1010:

-	1	2	3	4	5	mean
Matrix_mult_0	11.397	11.388	11.500	11.458	11.459	11.440
Matrix_mult_1	4.541	4.494	4.594	4.586	4.564	4.555
Performance	2.510	2.534	2.503	2.498	2.511	2.512

P=2.511

1023:

-	1	2	3	4	5	mean
Matrix_mult_0	10.284	10.226	10.224	10.187	10.302	10.244
Matrix_mult_1	5.518	5.474	5.482	5.452	5.453	5.475
Performance	1.864	1.868	1.865	1.868	1.889	1.871

P=1.871

1024:

-	1	2	3	4	5	mean
Matrix_mult_0	24.987	24.977	23.668	24.878	24.255	24.553
Matrix_mult_1	7.464	7.634	7.579	7.642	7.908	7.645
Performance	3.347	3.272	3.123	3.255	3.067	3.212

P=3.212

1025:

-	1	2	3	4	5	mean
Matrix_mult_0	10.494	10.474	10.480	10.515	10.456	10.483
Matrix_mult_1	5.546	5.443	5.519	5.479	5.549	5.507
Performance	1.892	1.9243	1.890	1.919	1.884	1.904

P=1.904

۲)اثر استفاده از کلمه کلیدی register در تابع دوم را با حذف آن بررسی کنید

999:

-	1	2	3	mean
Matrix_mult_0 Register=1	11.854	12.041	11.722	11.872
Matrix_mult_1 Register=1	5.645	5.691	5.715	5.683
Performance	2.100	2.116	2.051	2.089
Matrix_mult_0 Register=0	11.803	11.747	11.621	11.723
Matrix_mult_1 Register=0	8.033	7.986	7.930	7.983
Performance	1.469	1.471	1.465	1.468

P=1.468

1000:

-	1	2	3	mean
Matrix_mult_0 Register=1	7.893	7.790	7.812	7.832
Matrix_mult_1 Register=1	3.386	3.372	3.358	3.372
Performance	2.331	2.310	2.326	2.322
Matrix_mult_0 Register=0	7.736	7.750	7.901	7.795
Matrix_mult_1 Register=0	4.533	4.606	4.405	4.515
Performance	1.707	1.683	1.794	1.726

P=1.726

1023:

-	1	2	3	mean
Matrix_mult_0 Register=1	10.342	10.280	10.334	10.319
Matrix_mult_1 Register=1	5.481	5.533	5.505	5.506
Performance	1.887	1.858	1.877	1.874
Matrix_mult_0 Register=0	10.276	10.203	10.323	10.267
Matrix_mult_1 Register=0	7.685	7.570	7.589	7.615
Performance	1.337	1.348	1.360	1.348

P=1.348

1024:

-	1	2	3	mean
Matrix_mult_0 Register=1	25.122	24.912	24.692	24.909
Matrix_mult_1 Register=1	7.552	7.581	7.483	7.539
Performance	3.327	3.286	3.300	3.304
Matrix_mult_0 Register=0	24.873	24.345	24.694	24.637
Matrix_mult_1 Register=0	13.984	13.641	13.802	13.809
Performance	1.779	1.785	1.789	1.784

P=1.784

میبینیم که استفاده از رجیستر سرعت بیشتری به کد ما میدهد به خاطر این که متغیر را کنار دست قرار می دهد.

عدم استفاده از رجیستر میبینیم که پرفورمنس استفاه از تابع دوم برای ۳ داده اول نزدیک ۲۵ تا ۳۰ درصد افت میکند:

999: 0.70

1000:0.74

1023:0.72

اما برای عدد ۱۰۲۴ این میزان کاهش به ۴۷ درصد میرسد و نسبت پرفورمنس ها به ۰/۵۳ میرسد.

۳)ابتدا برای ۱۰۲۴ را بررسی میکنیم که چرا زمانش زیاد میشود:

در این کد ما از روش loop tiling استفاده کردیم:

Loop tiling, also known as *loop blocking*, is a loop transformation that exploits spatial and temporal locality of data accesses in loop nests. This transformation allows data to be accessed in blocks (tiles), with the block size defined as a parameter of this transformation. Each loop is transformed in two loops: one iterating inside each block (intratile) and the other one iterating over the blocks (intertile).

حالا مشکل این الگوریتم چیست:

Problems tiling on the CPU

Tiling is not cache-agnostic. This means that if we pick a tile size that's optimal for a CPU with a given cache size and then end up running the same code on a CPU with a smaller cache size (assuming the same instruction set), the performance on the CPU with the smaller cache might end up abysmal. Note that we might essentially end up with the case of the ΘΘ(N3$\blacksquare$3) cache misses. The other headache is that the number of unique implementations would need to be dependent on all cache hierarchies. The tiling discussion above is based on one level of caching, but modern CPUs have 3 levels (L1, L2, L3). This means we might need to add extra loops to cover tiles for L2, not

just L1, and do the same for L3. Too many loops end up having control flow costs, which may outweigh the benefits of 3 levels of cache-aware tiling.

Another major concern with cache size-tuned algorithms is that they make the assumption that nothing else can run on the CPU at any given time. This might be an appropriate assumption to make if we were using a microcontroller, but on any platform with a running operating system, it's unrealistic. Even if the CPU does not need to run any heavy processes in the background, the kernel itself needs to context switch between OS housekeeping tasks (say interrupt servicing) and running the application. The context switch between the kernel and the application can result in purging caches to make room for the task being done by the kernel. The problem is obviously worse if the kernel needs to context switch not just between itself and our matrix multiplying application, but other userspace applications. This for instance is a serious concern if we try to load up the cache with a lot of data at once, since it may be evicted due to a context switch.

Overall, the fact that tiling isn't all that great, because it's not cache-agnostic, it assumes that there are no context switches, and the performance seems to reflect those shortcomings.

در n=1024 کامپایلر بجای اینکه از کش بخواند از ممووری میخواند برا همین زمان افزایش پیدا میکند اما در کد دوم یک بخش خیلی زیادی از داده ها را در رجیستر میگذاریم(رجیستر ها هم اندازه محدودی دارند به همین دلیل خور به خود بعضی رجیستر ها پاک می شوند و دوباره پر میشوند و در تابع دوم هم سرعت کم میشود) به این دلیل در تابع دوم سرعت کمتر کم میشود و بازده استفاده از تابع دوم بیشتر میشود.

در n=1000 هم کامپیلر با استفاده از تکنیک های شیفتینگ زمان خواندن را کم میکند. همینطور در n=1001 که برا خواندن از همان تکنیک استفاده میشود و در اخر یک خانه جا بجا می شود برا همین سرعتش از n=999 بیشتر است.