دانشگاه صنعتی شریف

دانشکده مهندسی برق

درس طراحی سیستمهای میکرو پروسسوری (۲۵۷۷۱)

مجموعه آزمایش‌های بهره گیری از قابلیت‌های پردازنده‌ها

مرحله‌های اول و دوم: پوینتر و رجیستر در زبان C و استفاده‌ی حداکثری از Cache

تهیه کنندگان:

محمّدرضا موحّدین

علیرضا عباسیان

به نـــــام خـــــــدا

مقدمه

هدف از این مجموعه آزمایش‌ها که در چند فاز ارائه می‌شود، فعال‌سازی و بهره‌گیری از قابلیت‌های گوناگون پردازنده‌ها است که از مجموعه بخش‌های Going Faster کتاب Computer Organization & Design: The HW/SW Interface, 6th Edition الهام گرفته شده اند. این بخش‌های کتاب فوق در یک فایل جداگانه در اختیار شما قرار می‌گیرد.

در این آزمایش‌ها، عملیات ضرب دو ماتریس مربعی در نظر گرفته شده و در مراحل گوناگون تلاش می‌شود زمان اجرای این ضرب بهبود یابد. این مراحل شامل استفاده از پوینتر و رجیستر در زبان C، بکارگیری حافظه‌ی Cache، بکارگیری قابلیت‌های اجرای چندگانه‌ی پردازنده با Loop Unrolling، استفاده از دستورات برداری از قبیل SSE و AVX و نهایتاً استفاده از چند هسته‌ی پردازنده می‌باشند.

مرحله‌ی صفر: کد مرجع ضرب

کد مقابل به عنوان کد مرجع و نقطه‌ی شروع آزمایش‌ها مورد استفاده قرار می‌گیرد که فرایند ضرب ماتریس را در سه حلقه‌ی ساده و سر راست اجرا می‌کند. زمان اجرای این کد برای سایز مشخصی از ماتریس (متغیر n) به عنوان مبدأ مقایسه‌ی بهبود کیفیت کدهای آتی است. همچنین این کد برای صحت سنجی کدهای آینده مورد استفاده قرار خواهد گرفت. البته این کار باید با مقدار کوچک n (مثلاً حدود ۲۰ تا ۱۰۰) صورت گیرد تا زمان صحت سنجی بیش از اندازه نشود.

```
void matrix_mult_0
(int n, double* a, double* b, double* c){
  int i, j, k;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++){
      c[i*n+j] = 0.0;
      for(k = 0; k < n; k++)
        c[i*n+j] += a[i*n+k] * b[k*n+j];
    }
}
```

مرحله‌ی اول: استفاده از پوینتر و رجیستر در زبان C

در اولین قدم جهت بهبود سرعت اجرای ضرب ماتریس‌ها، کد مقابل را در نظر بگیرید. در این تابع، دیگر از آدرس‌دهی آرایه‌ها توسط اندیس استفاده نشده بلکه پوینترها به جای آنها به کار رفته‌اند. همچنین تلاش شده است که کامپایلر متغیرهای پر استفاده را به رجیسترهای داخلی پردازنده اختصاص دهد.

```
void matrix_mult_1
(int n, double* a, double* b, double* c){
  register double cij;
  register double *at, *bt;
  register int i, j, k;
    for (i = 0; i < n; i++, a+ = n)
      for (j = 0; j < n; j++, c++) {
        cij = 0;
        for(k = 0, at = a, bt = &b[j];
            k < n; k++, at++, bt += n)
              cij += *at * *bt;
        *c = cij;
      }
}
```

۱-۱- دو تابع فوق را با مقادیر مختلف n اجرا و زمان اجرا را مقایسه کنید. قطعه کدی که همراه دستورکار است، می‌تواند برای این منظور مورد استفاده قرار گیرد.

۱-۲- اثر استفاده از کلمه کلیدی register در تابع دوم را با حذف همه‌ی آنها بررسی کنید.

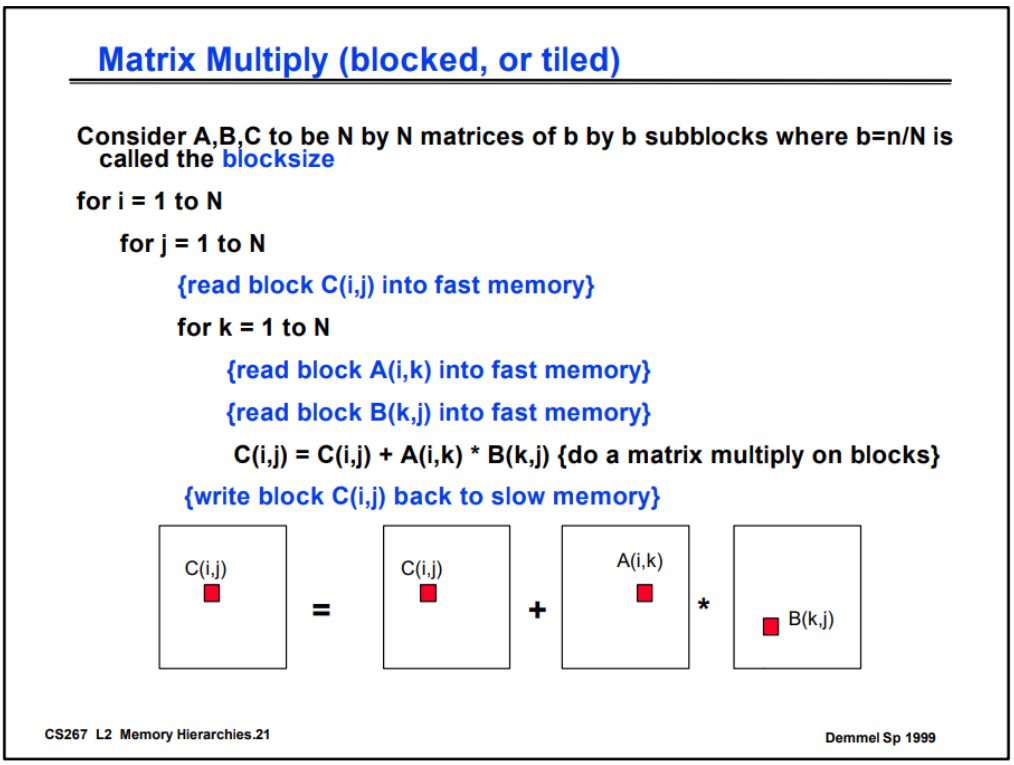
۱-۳- تابع دوم را به ازای اندازه‌های گوناگون ماتریس یا همان متغیر n اجرا کنید. در غالب پردازنده‌ها، حوالی n = 1024 زمان اجرا یک افزایش چشمگیر دارد. علت چیست؟

مرحله‌ی دوم: استفاده‌ی حداکثری از حافظه‌ی Cache

برای به حداقل رساندن دسترسی‌های مستقیم به حافظه‌ی اصلی و بکارگیری Cache بجای آن، دو روش زیر را به کار گرفته و نتایج زمان اجرا را برای سایز نسبتاً بزرگ ماتریس‌ها (حداقل ۲۰۴۸ و ترجیحاً ۴۰۹۶ و بیشتر) استخراج و با بهترین نتایج مرحله‌ی اول مقایسه کنید.

۲-۱- مانریس دوم را ترانهاده (Transpose) کرده و عملیات ضرب را انجام دهید. برای ماتریس ترانهاده، هم می‌توانید حافظه‌ی جدیدی اختصاص (_aligned_malloc) داده و یا از همان فضای ماتریس دوم استفاده نمایید. در این حالت دوم، باید ماتریس مجدداً ترانهاده شده تا به حالت قبلی خود بازگردد. به عبارت دیگر، در این فرایند نباید محتوای ماتریس دوم مخدوش گردد. برای یک مقایسه‌ی منصفانه، لازم است زمان ترانهاده کردن را نیز به عنوان بخشی از زمان ضرب ماتریس در نظر بگیرید.

۲-۲- روش ضرب بلوکی ماتریس‌ها را پیاده‌سازی کنید. برای این منظور می‌توانید به روش ارائه شده در شکل زیر مراجعه نمایید. برای سادگی، سایز ماتریس اصلی را توانی از دو انتخاب کنید و زمان ضرب را به ازای سایزهای گوناگون بلوک (مثلاً ۸، ۱۶، ۳۲، ۶۴ و حتی ۱۲۸ و ۲۵۶) استخراج، مقایسه و تحلیل کنید.



نکات مهم در نوشتن کد و بررسی زمان اجرای آن

۱- کلیه‌ی بهینه‌سازی‌های کامپایلر (Compiler Optimizations) را قطع کنید تا فقط نتیجه‌ی اقدامات شما در نتیجه‌ی خروجی ظاهر شود.

۲- کلیه‌ی برنامه‌های جانبی از قبیل آنتی ویروس، به روز رسانی‌ها، Search Indexing و دیگر برنامه‌ها را هنگام استخراج زمان اجرای توابع متوقف کنید تا زمان اجرای تابع هر چه بیشتر دقیق باشد.

۳- زمان را در چندین اجرا استخراج و میانگین آنها را به دست آورید. زمان‌های طولانی غیر متعارف را اولاً از میانگین حذف و ثانیاً علت‌یابی نمایید.

۴- برای تضمین آنکه حافظه‌های تخصیص یافته نسبت به سایز متغیر aligned است از تابع aligned_malloc_ بجای malloc استفاده کنید. بدین ترتیب اطمینان حاصل می‌شود از بابت دسترسی‌های misaligned زمان پردازنده تلف نخواهد شد. در انتها نیز فضای حافظه توسط تابع aligned_free_ آزاد می‌گردد.

۵- اطمینان حاصل کنید کد C فوق در مود ۶۴ بیتی (x64) و نه ۳۲ بیتی (x86)، کامپایل می‌شود. برای این منظور به راهنمای کامپایلر مورد استفاده مراجعه کنید.
نکته: اگر در ضمن مقایسه‌ی کد اول و دوم و در حال استفاده از register در تعریف double cij ملاحظه کردید که مقادیر با هم برابر نیستند (مثلاً در بخش مقایسه‌ی کدی که همراه دستور کار ارسال شده است) در این صورت کد در مود ۳۲ بیتی کامپایل شده است. ریشه‌ی این اختلاف نیز انجام عملیات در رجیسترهای ۸۰ بیتی x87 است که در SSE و بعد از آن اصلاح شده است.

۶- برای اختصاص یک هسته به اجرای برنامه و عدم چرخش آن بر روی هسته‌های گوناگون که موجب غیر واقعی شدن زمان اجرا می‌شود می‌توانید از دستور start /affinity 0xPP در cmd ویندوز (و نه power-shell) استفاده کنید که در آن 0xPP پترن هسته‌هایی است که می‌خواهید به یک برنامه اختصاص دهید. برای ساده‌ترین حالت و اختصاص یک هسته از 0x4 استفاده کنید. برای جزئیات بیشتر به راهنمای دستور start مراجعه کنید.

۷- برای بررسی کد اسمبلی معادل کد C می‌توانید از دستور objdump -D -S FILE.o > FILE.asm استفاده کنید. FILE.o کد Object تولید شده توسط کامپایلر قبل از لینک نهایی است و معمولاً در پوشه‌ی obj نگهداری می‌شود.
به عنوان نمونه، تفاوت کد تولید شده توسط کامپایلر را زمانیکه cij بصورت ساده (یعنی double cij) و یا رجیستر (یعنی register double cij) تعریف شده باشد را در زیر ملاحظه کنید:

cij declaration is regular, i.e. non register	cij declaration is register

`cij = 0;`
`  pxor  %xmm0,%xmm0`
`  **movsd  %xmm0,-0x78(%rbp)**`
 	
`.....`
 	
`cij += *at * *bt;`
`  movsd  (%rbx),%xmm1`
`  movsd  (%rsi),%xmm0`
`  mulsd  %xmm1,%xmm0`
`  **movsd  -0x78(%rbp),%xmm1**`
`  addsd  %xmm1,%xmm0`
`  **movsd  %xmm0,-0x78(%rbp)**`
 	
`.....`
 	
`*c = cij;`
`  mov  -0x18(%rbp),%rax`
`  **movsd  -0x78(%rbp),%xmm0**`
`  movsd  %xmm0,(%rax)`
	`cij = 0;`
	`  pxor  %xmm6,%xmm6`
	
	`.....`
	
	`cij += *at * *bt;`
	`  movsd  (%rbx),%xmm1`
	`  movsd  (%rsi),%xmm0`
	`  mulsd  %xmm1,%xmm0`
	`  addsd  %xmm0,%xmm6`
	
	`.....`
	
	`*c = cij;`
	`  mov  -0x18(%rbp),%rax`
	`  movsd  %xmm6,(%rax)`