CAP-387(2016) – Tópicos Especiais em Computação Aplicada: Construção de Aplicações Massivamente Paralelas

Aula 16: Vetorização em Compiladores Atuais

Celso L. Mendes, Stephan Stephany

LAC / INPE

Emails: celso.mendes@inpe.br, stephan.stephany@inpe.br

Programação com Vetorização

Três Formas Possíveis:

a) Codificação em Assembly
   - Máximo desempenho; grande esforço; baixa portabilidade

b) Codificação com funções intrínsecas
   - Esforço ainda significativo; pouca portabilidade;

c) Uso de compiladores vetorizadores
   - Alta portabilidade do programa; desempenho depende da qualidade do compilador; forma preferencial atualmente

Problemas para Vetorização

- Principal alvo da vetorização de programas
  - Loops – na verdade, iterações de um loop
- Objetivo primário de um compilador vetorizador:
  - Vetorizar loops, de modo a explorar o hardware vetorial
- Problema Fundamental: Dependências
  - Compilador deve garantir que código vetorizado não viola dependências entre iterações do loop
  - Código vetorizado deve produzir o mesmo resultado que o código serial original

Problemas para Vetorização

- Dependências entre iterações
  - Como já visto, existe dependência entre dois commandos quando ambos acessam a mesma posição de memória, e um deles escreve naquela posição
- Linguagens com ponteiros:
  - Como garantir se um ponteiro acessa a mesma região que um outro ponteiro ou array?
  - Quando isto ocorre, é comum dizer que há um *alias* entre os ponteiros (isto é, eles acessam uma região comum de memória)

Desafio para o Compilador

- Exemplo: duas funções semelhantes
  - void sum (double *total, double *a, int n) {
    int i; for (i=0; i<n; i++) *total += a[i]; }
  - void sum2 (double *total, double *a, int n) {
    int i; double s=*total; for (i=0; i<n; i++) s += a[i]; *total=s; }

Pergunta: as duas funções produzem o mesmo resultado sempre ?

Sugestão: Considere as chamadas sum(&a[2],a,3) e sum2(&a[2],a,3) onde a[] é um array de tamanho igual ou maior que 3, e veja se os resultados de *sum* e *sum2* são os mesmos

Desafio para o Compilador

- Exemplo (cont.): duas funções semelhantes
  - void sum (double *total, double *a, int n) {
    int i; for (i=0; i<n; i++) *total += a[i]; }
  - void sum2 (double *total, double *a, int n) {
    int i; double s=*total; for (i=0; i<n; i++) s += a[i]; *total=s; }

Supondo a={20,21,22}, ou seja a[0]=20, a[1]=21, a[2]=22

sum(&a[2],a,3): *total=a[2]=22;

i=0: *total=*total+a[0]: 22+a[0]=22+20 a[0]=20,a[1]=21,a[2]=22+20

i=1: *total=*total+a[1]: 22+20+a[1]=22+20+21 a[0]=20,a[1]=21,a[2]=22+20+21

i=2: *total=*total+a[2]: 22+20+21+a[2]=22+20+21+(22+20+21)

sum2(&a[2],a,3): *total=a[2]=22; s=*total=a[2]=22;

i=0: s=s+a[0]=22+20

i=1: s=s+a[1]=(22+20)+21

i=2: s=s+a[2]=(22+20+21)+22 ; *total=22+20+21+22

Desafio para o Compilador

- Exemplo (cont.): Na verdade, os resultados são estes:
  - sum: a[2]+a[0]+a[1]+a[2]+a[0]+a[1]
  - sum2: a[2]+a[0]+a[1]+a[2]
  - Logo, os resultados são diferentes!
    - Razão: Primeiro argumento aponta para região do segundo
      → Há um alias entre "&a[2]" e "a"
- Problema concreto para o compilador:
  - Como decidir se há dependências entre iterações, caso as iterações contenham um alias ?
  - Decisão precisa ser tomada em tempo de compilação!
  - Na dúvida, compilador deve ser conservador e não vetorizar

Auxílio ao Compilador

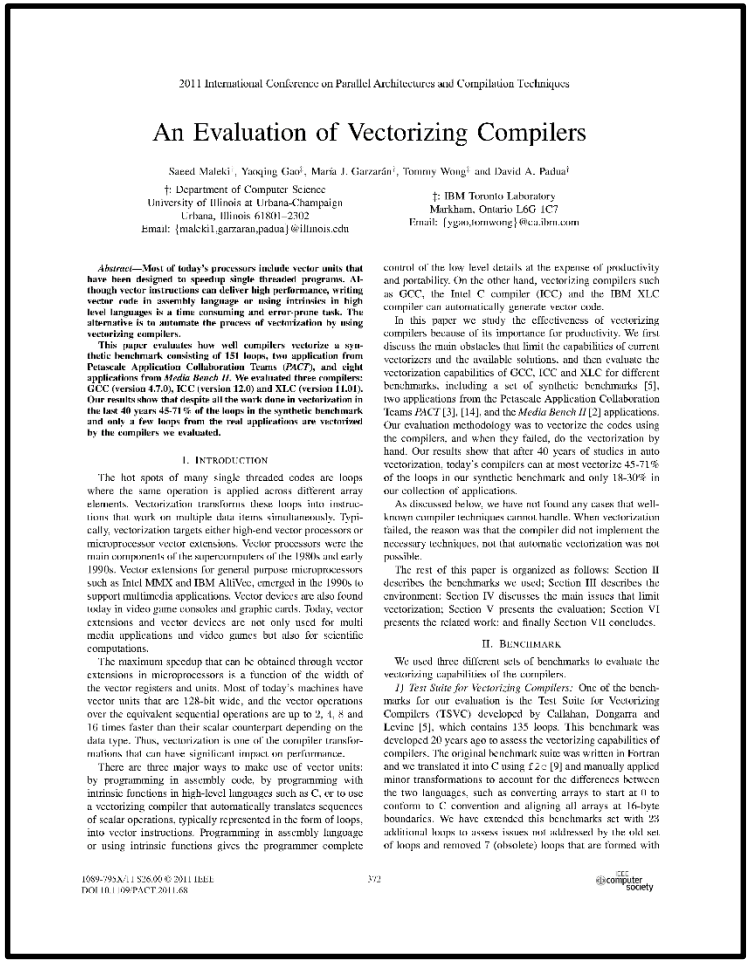
- Atributo *__restrict__*
  - Inserido pelo programador, para programas em C
  - Informa ao compilador que um ponteiro com este atributo não tem um *alias* com nenhum outro ponteiro
    - Compilador tem máxima liberdade para vetorizar
    - Óbvio que cabe ao programador garantir que um *alias* de fato não exista
    - Se existir, resultado da execução poderá ser incorreto

Vetorização em Compiladores Atuais

- Avaliação em 2011:

  S. Maleki et al: *An Evaluation of Vectorizing Compilers* – PACT'2011

  - URL: http://polaris.cs.uiuc.edu/~garzaran/doc/pact11.pdf
  - 3 compiladores avaliados:
    - gcc-4.7.0, icc-12.0, xlc-11.1
  - 2 plataformas de hardware usadas
    - INTEL i7 @ 2,66 MHz
    - IBM Power7 @ 3,86 MHz



2011 International Conference on Parallel Architectures and Compilation Techniques

An Evaluation of Vectorizing Compilers

Saeed Maleki†, Yaoqing Gao‡, María J. Garzarán†, Tommy Wong‡ and David A. Padua†

†: Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801–2302
Email: {maleki1,garzaran,padua}@illinois.edu

‡: IBM Toronto Laboratory
Markham, Ontario L6G 1C7
Email: {ygao,tomwong}@ca.ibm.com

***Abstract*—Most of today's processors include vector units that have been designed to speedup single threaded programs. Although vector instructions can deliver high performance, writing vector code in assembly language or using intrinsics in high level languages is a time consuming and error-prone task. The alternative is to automate the process of vectorization by using vectorizing compilers.**

This paper evaluates how well compilers vectorize a synthetic benchmark consisting of 151 loops, two application from Petascale Application Collaboration Teams (*PACT*), and eight applications from *Media Bench II*. We evaluated three compilers: GCC (version 4.7.0), ICC (version 12.0) and XLC (version 11.01). Our results show that despite all the work done in vectorization in the last 40 years 45-71% of the loops in the synthetic benchmark and only a few loops from the real applications are vectorized by the compilers we evaluated.

I. Introduction

The hot spots of many single threaded codes are loops where the same operation is applied across different array elements. Vectorization transforms these loops into instructions that work on multiple data items simultaneously. Typically, vectorization targets either high-end vector processors or microprocessor vector extensions. Vector processors were the main components of the supercomputers of the 1980s and early 1990s. Vector extensions for general purpose microprocessors such as Intel MMX and IBM AltiVec, emerged in the 1990s to support multimedia applications. Vector devices are also found today in video game consoles and graphic cards. Today, vector extensions and vector devices are not only used for multi media applications and video games but also for scientific computations.

The maximum speedup that can be obtained through vector extensions in microprocessors is a function of the width of the vector registers and units. Most of today's machines have vector units that are 128-bit wide, and the vector operations over the equivalent sequential operations are up to 2, 4, 8 and 16 times faster than their scalar counterpart depending on the data type. Thus, vectorization is one of the compiler transformations that can have significant impact on performance.

There are three major ways to make use of vector units: by programming in assembly code, by programming with intrinsic functions in high-level languages such as C, or to use a vectorizing compiler that automatically translates sequences of scalar operations, typically represented in the form of loops, into vector instructions. Programming in assembly language or using intrinsic functions gives the programmer complete control of the low level details at the expense of productivity and portability. On the other hand, vectorizing compilers such as GCC, the Intel C compiler (ICC) and the IBM XLC compiler can automatically generate vector code.

In this paper we study the effectiveness of vectorizing compilers because of its importance for productivity. We first discuss the main obstacles that limit the capabilities of current vectorizers and the available solutions, and then evaluate the vectorization capabilities of GCC, ICC and XLC for different benchmarks, including a set of synthetic benchmarks [5], two applications from the Petascale Application Collaboration Teams *PACT* [3], [14], and the *Media Bench II* [2] applications. Our evaluation methodology was to vectorize the codes using the compilers, and when they failed, do the vectorization by hand. Our results show that after 40 years of studies in auto vectorization, today's compilers can at most vectorize 45-71% of the loops in our synthetic benchmark and only 18-30% in our collection of applications.

As discussed below, we have not found any cases that well-known compiler techniques cannot handle. When vectorization failed, the reason was that the compiler did not implement the necessary techniques, not that automatic vectorization was not possible.

The rest of this paper is organized as follows: Section II describes the benchmarks we used; Section III describes the environment; Section IV discusses the main issues that limit vectorization; Section V presents the evaluation; Section VI presents the related work; and finally Section VII concludes.

II. Benchmark

We used three different sets of benchmarks to evaluate the vectorizing capabilities of the compilers.

1) Test Suite for Vectorizing Compilers: One of the benchmarks for our evaluation is the Test Suite for Vectorizing Compilers (TSVC) developed by Callahan, Dongarra and Levine [5], which contains 135 loops. This benchmark was developed 20 years ago to assess the vectorizing capabilities of compilers. The original benchmark suite was written in Fortran and we translated it into C using *f2c* [9] and manually applied minor transformations to account for the differences between the two languages, such as converting arrays to start at 0 to conform to C convention and aligning all arrays at 16-byte boundaries. We have extended this benchmarks set with 23 additional loops to assess issues not addressed by the old set of loops and removed 7 (obsolete) loops that are formed with

1089-795X/11 $26.00 © 2011 IEEE
DOI 10.1109/PACT.2011.68

Códigos para Avaliação

a) Benchmark público com 151 loops
   - 128 loops de uma suite antiga de teste de sistemas vetoriais
   - 23 loops adicionais, com aspectos não cobertos ainda
     - TSVC: *Test Suite for Vectorizing Compilers*
     - Originalmente em Fortran, adaptada para C com *f2c*

b) Aplicações científicas com processamento numérico
   - MILC – quantum chromodynamics
   - DNS - turbulência

c) Aplicações de multimedia
   - Encoders/decoders de JPEG, H263, MPEG2, MPEG4

Desempenho com Vetorização

- Ganho esperado da vetorização
  - Avaliação com um código simples:

```
for (k=0; k<ntimes; k++) {
      for (i=0; i<N; i++) a[i]=b[i] + 1;
}
```

  - a[] e b[]: tamanho = 12.5 KB (cabem na cache L1)
  - ntimes = 400.000
  - Compilações com e sem vetorização
    - Mas com outras otimizações em ambos os casos

Desempenho com Vetorização

- Ganho estimado por Maleki et al:
  - Tempos de execução em *segundos*
  - IBM Power7: AltiVec, 128 bits ; INTEL i7: SSE, 128 bits
  - Compiladores: IBM: xlc; INTEL: icc ou gcc ?

Machine		double	float	int	short	char
Power 7	Vec	1.18	1.10	1.11	1.16	1.14
	NoVec	1.59	2.91	2.64	5.46	10.49
	Speedup	1.35	2.64	2.37	4.71	9.20
Intel i7	Vec	1.67	1.61	1.67	1.65	1.68
	NoVec	2.37	4.81	4.83	9.61	19.22
	Speedup	1.42	2.99	2.89	5.82	11.44

TABLE II
RUNNING TIMES AND SPEEDUPS FOR THE OPERATION `A[I]=B[I]+1` WITH FIXED ARRAY SIZE 12.5KB.

Compilação no Santos Dumont

- Opções de compilação utilizadas
  - -O3 utilizado em todos os casos
  - Flags específicos de vetorização e não-vetorização
  - Flags específicos para relatório da vetorização

Compilador	Não-Vetorizar	Vetorizar	Relatório
GCC	-fno-tree-vectorize	Default (-ftree-vectorize)	-ftree-vectorizer-report=2
ICC	-no-vec	Default (-vec)	-qopt-report
PGCC	-Mnovect	-Mvect=sse –Mcache_align	-Minfo=vect

Desempenho no Santos Dumont

		Double	Float	Int	Short	Char
GCC 4.4.7	Não-Vetoriz.	0,538 s	1,073 s	1,078 s	2,147 s	4,278 s
	Vetorizado	0,278 s	0,278 s	0,273 s	2,146 s	4,277 s
	Speedup	1,94	3,86	3,95	1,00	1,00
GCC 5.3.0	Não-Vetoriz.	0,580 s	1,078 s	1,078 s	2,147 s	4,282 s
	Vetorizado	0,203 s	0,206 s	0,278 s	0,278 s	0,274 s
	Speedup	**2,86**	5,23	3,88	**7,72**	**15,63**
ICC 16.0.2	Não-Vetoriz.	0.405 s	0.811 s	0.805 s	1.606 s	4.277 s
	Vetorizado	0.141 s	0.144 s	0.169 s	0.169 s	0.169 s
	Speedup	**2,87**	5,63	4,76	**9,50**	**25,31**
PGCC 16.5	Não-Vetoriz.	0.288 s	1.090 s	0.569 s	1.176 s	2.678 s
	Vetorizado	0.173 s	1.341 s	0.276 s	1.182 s	2.675 s
	Speedup	1,66	0.81	2,06	0,99	1,00