

Processamento vetorial - cada instrução processa N elementos de dados. O Texec CPI * #IP. #I indica CPI reduzindo seja em paralelo, caso seja sequencial o CPI aumenta.

Taxonomia de Flynn diz: SISD, SIMD, MISD, MIMD.

Instruções AVX [V] ADD [SIP] escalarelacionalmulticore [SID]

y.xmmm? → 128 ← m128

Sem V: será SSE, com y.xmmm? / m128 yxmmm?

Vom V: será AVX, com y.ymmm? / y.ymmm? / m256, y.ymmm? y.ymmm? → 256 ← m256 memoria

[SIP] → Escalar ou vetorial / Single - 32 bits logo 128 dà 4 e 256 dà 8 registros
[SID] → Single ou Double Double - 64 bits logo 128 dà 2 e 256 dà 4)

[V] MOV [AIU] P [SID], em que [AIU] indica se o endereço está alinhado ou não.

Para desenvolver Processamento Vetorial, pode ser em Assembly, Compiler Intrinsics ou Auto-vetorial

Compiler Intrinsics -- m256 _mm_mm-add_ps (- mm256, - m256) etc

↑ ↑ Temos ainda: -m256 - m256_load_ps float*)
tipo de dado vetor **VADDPS** broadcast
-mm256_store_ps float*, -m256)

Auto-vetorialização. Pode existir aliasing, logo o compilador faz versioning com o modo escalar em modo vetorial, escolhendo que tem melhor runtime. Existe o qualificador __restrict__ para indicar que daquele apontador não existe qualquer outra referência para a zona memória.

Bloqueadores de N: Dados não contíguos (AoS, SoA) / Stride != 1, acessos não contíguos massacrando

loops não contíguos / estruturas condicionais (podendo-se usar uma máscara invés) / funções - localidade espacial

Dependências Raw em que $d = C^W \cdot C^R$ se $d < 0$ não há Raw, logo é vetorialável / built-in

Recomenda: Usei ciclos contíguos / $d = -C^W \cdot C^R$. Caso seja seu máquina AVX com W elementos se

existirem condicionais usar máscaras, entao $d > W$, ainda é vetorialável.

Eficiente com ciclos stride=1 + endereços alinhados Raw, muitos apontadores e prevenir aliasing, usar memória

- Multicore - B

Termos ILP com pipelining e superscalariade / P=C v2 f, logo a potência dissipada aumenta com a

para isso criou-se TLP, um paralelismo mais grosso, sendo partes de um programa tendo o seu espaço

para aumentar o efêlio em compiladores com vários programas diminuindo o tempo de execução.

Termos o SMT - Simultaneous Multithreading. As threads podem usar os recursos de cada core.

Em que em Intel HT são 2 threads em cada núcleo físico. Threads com o mesmo propósito terão de recorrer aos recursos de outro core. Daí criou-se multi-core para que as threads tenham livre

sendo possível existir Multicore + SMT. Diz-se que com Intel HT tem dois núcleos/lógoicos lógicos.

Ao aumentar os cores, o SO vê como se fossem independentes, mas depois começa a guerra por

recursos tais como BUS e memória. Para isso gerir a memória e os programas é fundamental.

- OpenMP -

Criam-se blocos paralelos de código. No final todas as threads sincronizam e deixam de existir exceto

os dados que são partilhados por defeito exceto variáveis privadas como quando são declaradas por um bloco privado, marcadas como privadas ou índices de ciclos for.

#pragma omp enqueued [clausulas]. Algunhas funções são int omp_get_thread_num,

omp_get_num_threads, omp_set_num_threads, double omp_get_wtime, etc.

Directiva parallel #pragma omp parallel cria-se um nº threads, execute-se depois sincroniza no final

Data scope: #pragma omp parallel private (x), ou os índices do for. Normalmente há threads = processos

Directiva for - o índice é privado. Caso sejam vários for, temos o opção collapse.

Directiva parallel-if, em que se é paralelo se a condição for 0.

Directiva single - Apesar a 1ª thread a chegar, faz o trabalho. Existe um sync no final, logo todas as

threads esperam o final.

Directiva critical / atomic - apenas uma thread executa naquele tempo, mas todas vão executar.

Reduction: Dedica-se a partes do código que usam longos dados e dão um único valor. Sirva op.

associativas como +, -, *, /, %, !, &, &, |, &&, ||, etc. VR nunca é associativo.

Speedup = T_1 / T_p , tempo sequencial sobre tempo paralelo com p processadores

Eficiência = $(Sp / p) \times 100$. O speed up pode ser inferior ao linear devido aos overheads, como gestão

do paralelismo, replicação do trabalho, distribuição da carga, sync no final de bloco

- GRUs -

~~Instruções AVX~~ Processamento Vetorial

SISD → Escalar

SIMD → Vetorial

MISD → \propto

MIMD → Multicore

Instruções AVX

[V] ADD [SIP] [SID]

Sem V : será $x_{mmmm?}/m128$, $x_{mmmm?}$

Com V : será $y_{ymmm?}, y_{ymmm?}/m256, y_{ymmm?}$

$x_{mmmm?} \rightarrow 128$ bits $\leftarrow m128$

$y_{ymmm?} \rightarrow 256$ bits $\leftarrow m256$

registos Memória

[SIP] → ~~Escalar ou~~ Vetorial ?

[SID] → Single ou Double Precision Floating Point

Double → 64-bits, logo 128 díz 2. e 256 díz 4.

Single → 32-bits, logo 128 díz 4. e 256 díz 8.

instruções possíveis:

[V] MOV [AIU] P [SID]

O mov ainda pode ter [AIU] que significa que o endereço está alinhado ou não. (Tem de ser ~~vetorial~~ já agora)

• Assembly

• Compiler Intrinsics

• Auto-Vetorializaçāo



Auto-velorizada.

Pode existir aliasing, logo o compilador cria versioning com o modo escalar e modo vetorial, escolhendo o com melhor runtime.

Bloqueadores de AV. (ciclos)

- Dados não contíguos
- Stride $\neq 1$, ou seja está ordenado mas não contíguo (junto)
- Uncountable loops
- Condicionais
- funções não embutidas
- Dependências
 - Raw impossível
 - WAR possível

- Resumindo:

- Usar ciclos contínuos com pontos únicos de entrada saída
- Evitar condicionais; no entanto, máscaras são velorizáveis
- " dependências Raw
- " apontadores e prevenir aliasing
- Usar acessos à memória eficientes:
 - Ciclo mais aninhado, com stride 1 (dados consecutivos)
 - Alinhar os dados a múltiplos de 32 (Intel AVX)

Speed

ultituners MultiCore

LP - Várias instruções em paralelo

- Pipelining (até 32 estágios)

- ~ 7 estágios é ótimo em termos de potência e performance

- Superescalabilidade

- Múltiplos pipelines

• Como Moore dizia o desempenho aumenta a cada 18 meses.

$$P = CV^2 f$$

P - potência

C - capacidade

V - tensão

f - frequência

Logo o hardware adicional para ILP é complexo resultando num elevado consumo.

• O grau de ILP é 3 nos programas em média

Dai sera power wall. A barreira da potência

Pela isso a Intel seguiu o modelo (hyper) threading

• Threading é um paralelismo mais grosso que as instruções

• Pode ser partes de um programa ou programas/processos diferentes.

• Cada thread tem o seu estado (instruções, dados, contexto, registo, IP, etc).

ILP \rightarrow paralelismo implícito numa única sequência de instruções

TLP \rightarrow " expícito entre múltiplas sequências de instruções

Isto diminui o tempo de execução de programas paralelos
(multi-threaded)

Termos o Simultaneous Multi threading.

As threads podem usar os recursos de cada core. (2 max thread em cada processador)

Por ex: Caso uma thread queira somar inteiros e outra somar flutuantes, ambas são corridas ao mesmo tempo e utilizam os recursos apropriados. Threads com o mesmo propósito não podem usar o mesmo recurso no core. Para isso replicam-se múltiplos cores num único chip de modo a conseguir que as threads tenham o recurso livre e com SMT.

Logo os chips podem ser Multi Core com SMT também.

- Multi Core

O SO vê cada core como um processador independente

Ex: 4 cores (8 lógicos com Intel Hyper Threading)

dado que 2 threads em paralelo por core/processador

OMP - Open MultiProcessing

⇒ para expressar o paralelismo multi-threading e de memória partilhada

Modelo de execução:

- Criamos explícita de blocos paralelos de código
- No final de cada bloco
 - todas as threads sincronizam (barreira implícita)
 - todas as "n" excepto a principal deixam de existir.

N = 1000

```
#pragma omp parallel for  
for (i=0; i<N; i++)  
    A[i] = B[i] + C[i];
```

printf("programa done\n");

sequencial

paralela

Os dados podem ser partilhados por todas as threads ou privados acessíveis apenas a uma thread.

• Por de resto é partilhado.

As variáveis privadas são declaradas dentro dum bloco privado, explicitamente marcadas como privadas. Ex: índice de alguns ciclos for

- #pragma omp <nome diretivas [clausulas]
cada diretiva aplica-se ao bloco de instruções que lhe segue.

~~• compatibility~~

Algumas funções:

```
int omp_get_thread_num (void) ← Devolve o ID da thread  
int omp_get_num_threads (void) ← Devolve nº threads existentes no bloco  
void omp_set_num_threads (int) ← Estabelece nº threads a ser criadas no próximo bloco  
int omp_get_num_procs (void) ← nº de processadores disponíveis  
double omp_get_wtime (void) ← timestamp
```

#pragma omp parallel

{

... // bloco paralelo

}

- Cria um grupo de N threads
- Cada um executa independentemente o bloco
- No fim do bloco existe uma barreira (sync):
A thread principal só continua depois de todas as outras terem chegado ao fim do bloco

Data scope da diretiva parallel

• Variáveis globais ao bloco paralelo são partilhadas

• São variáveis privadas:

- locais ao bloco

pragma omp parallel

{ int i;

... }

- explicitamente declaradas privadas com a cláusula private (-) →
pragma omp parallel private(x)

- os índice dos ciclos abrangidos pela diretiva for

int i; ← privado porque é índice de

pragma omp parallel for ↴

for (i=0; i < N; i++)

A[i] = B[i] + C[i]

Por defeito, o nº de threads é igual ao nº de processadores (lógicos)

- diretiva for

- distribui as iterações do ciclo for pelas threads

pragma omp parallel

{ pri - - -

pragma omp for ↴

for - - -

}

A distribuição é feita, por defeito, estaticamente pelas threads

- caso seja for dentro de for

pragma omp parallel for collapse(2) private(x)

for (z

+2 { for (c

for (x -

Dessa forma os índices ficam privados e abrangidos.

collapse(3)

- Directiva parallel - if

pragma omp parallel if(i) ← i != 0 cria-se o bloco
i == 0 sequencial.

- Diretiva single

Apenas a primeira thread a atingir o executa
Todas as threads sync no final do bloco

pragma omp parallel

{
|
| ← paralelo

pragma omp single

{
| ...
| ← sequencial (sóma 1º thread que chega)

}

printf ... } ← paralelo
|
| ← sync

- Diretiva critical e atomic -

critical: apenas uma thread executa naquele instante (mas quando o instante acaba as outras threads sequenciam)

atomic - versão light das mesmas

- Reduzão é uma operação que processa um conjunto de dados para gerar um único valor (ex: soma/max/produto de array)

pragma omp parallel for reduction (+:sum)

{ for
| sum += .. }

Só funciona para operações associativas e nunca de float, double

Speed Up = $\frac{T_1}{T_p}$, tempo sequencial (1 processador) sobre o tempo paralelo com p processadores.

Eficiência $\frac{S_p}{P} \times 100$,