NACAD
Núcleo Avançado de Computação de Alto Desempenho

AMT
High Performance Cloud Computing

Modernização de Códigos

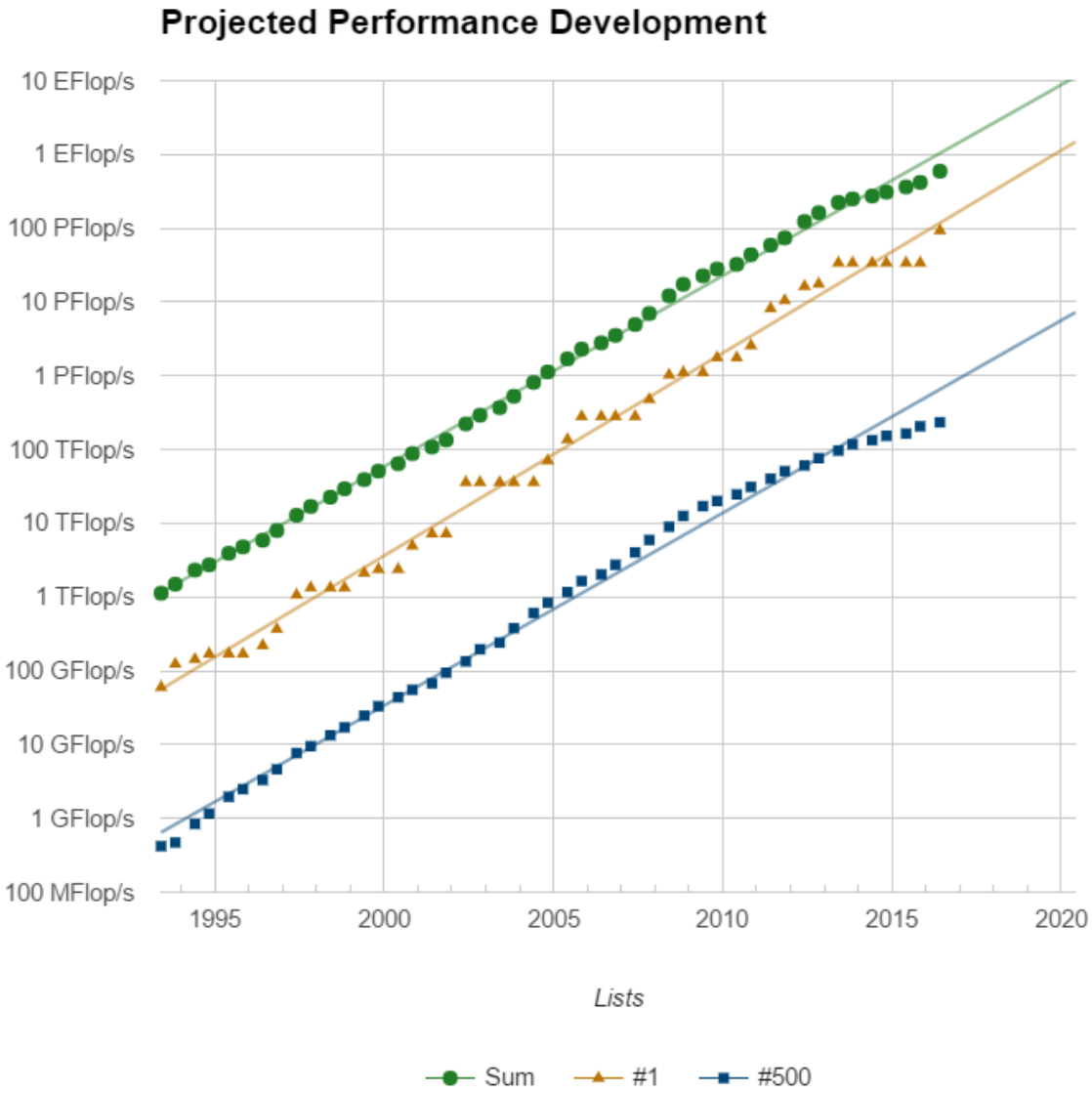
Dr. José Camata
camata@nacad.ufrj.br

NACAD/COPPE-UFRJ
Núcleo Atendimento em Computação de Alto Desempenho
(Parceiro AMT – Centro de Inovação da Intel)

Agenda

- Modernização de Códigos
  - Arquitetura Intel
  - Processo de otimização
  - Métricas de desempenho e peerfilagem
  - Técnicas de otimização:
    - manual e automática
  - Otimização Vetorial
  - Identificando Paralelismo Multi-Threading
  - Hands-on.

Computação na Era Exascala



Desafios da Computação Exascala

- A arquitetura do processador em aberto.
- Consumo energético
  - Projeção simplista exigiria 200 MW em 2020, o que é insustentável. A meta é de 20-40 MW para 1 exaflop.
- Largura de banda de memória e capacidade não estão mantendo o mesmo ritmo do crescimento dos FLOPs.
- Diminuição da frequências do *clock* e o número de unidades de processamento num único chip devem aumentar.
  - Modelo de programação será revisto: compiladores não serão capazes de esconder o nível de concorrência das aplicações
- Não é esperado melhora significativa no custo de movimentação de dados tanto quanto a de flops.
  - Algoritmos necessitam minimizar o movimento de dados, e não flops.
- Aumento de complexidade do gerenciamento do sistema de I/O
- Confiabilidade e resiliência serão pontos fundamentais.

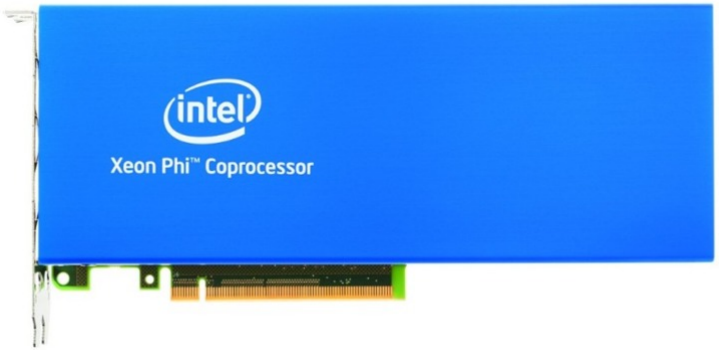
Fonte: Departamento de Energia Norte-Americano

Arquitetura Intel

Intel Xeon Processor

Current: Broadwell
Upcoming: Skylake

Intel Xeon Phi Coprocessor, 1st generation



Knights Corner (KNC)

Intel Xeon Phi Processor, 2nd generation*

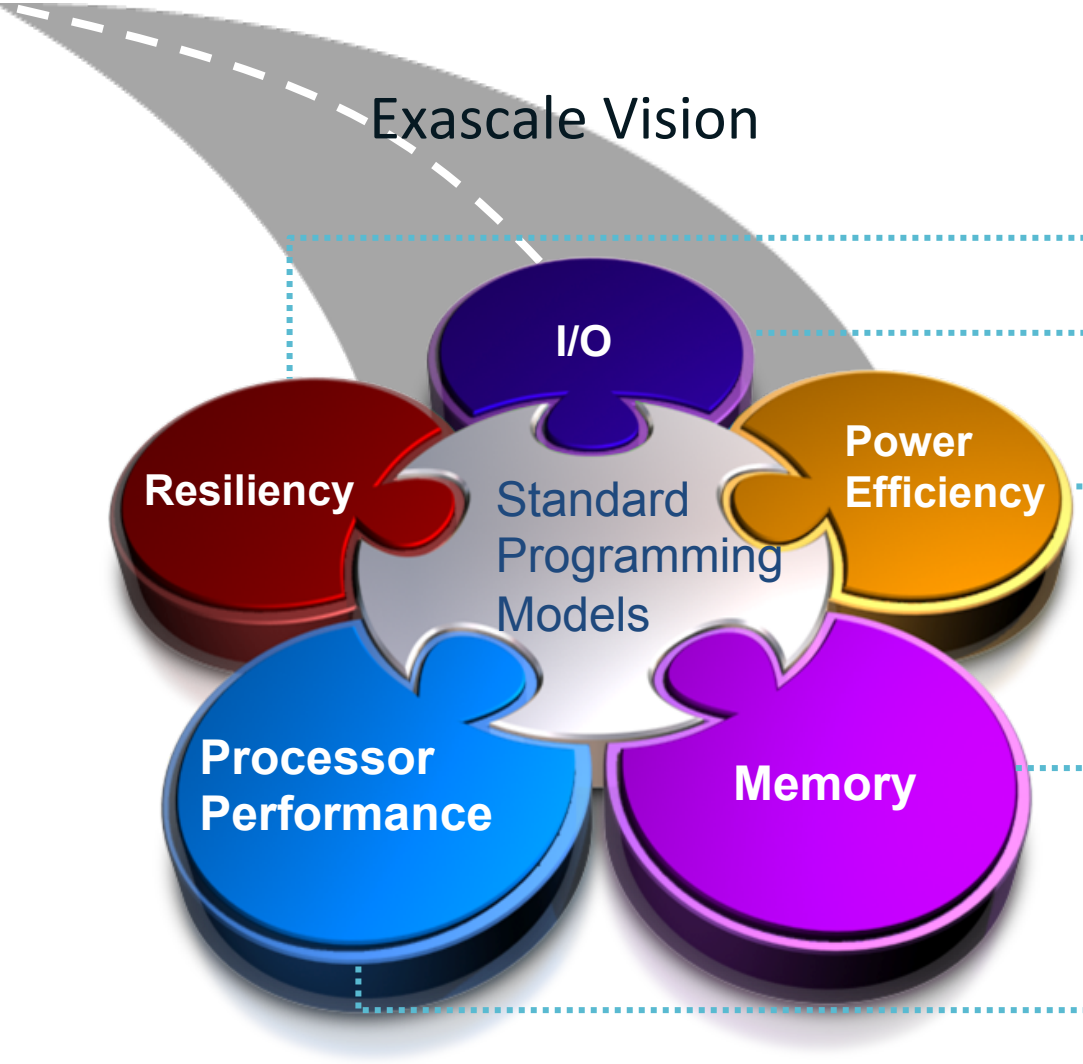
* socket and coprocessor versions

Knights Landing (KNL)

Multi-Core Architecture

Intel Many Integrated Core (MIC) Architecture

Next Step on Intel's Path to Exascale Computing

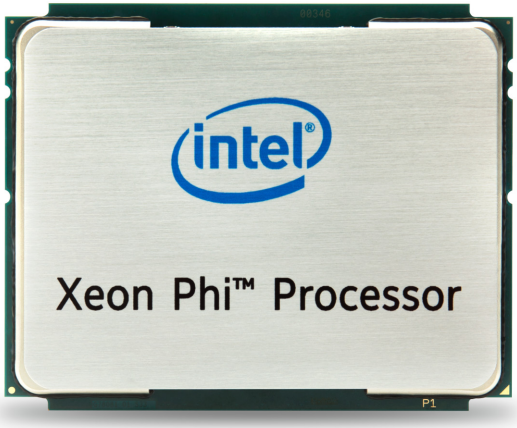


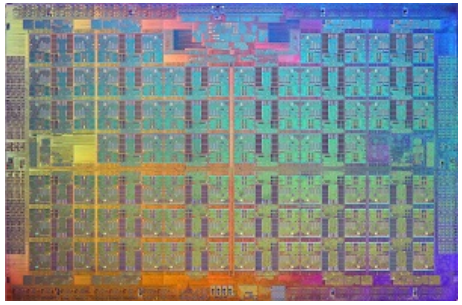
Next Step: KNL

- *Systems scalable to >100 PFlop/s*
- *Up to 100 Gb/s with Storm Lake integrated fabric*
- *Over 15 GF/Watt[1]*
- *~500 GB/s sustained memory bandwidth with integrated on-package memory*
- *~3X Flops and ~3X single-thread theoretical peak performance over Knights Corner[1]*

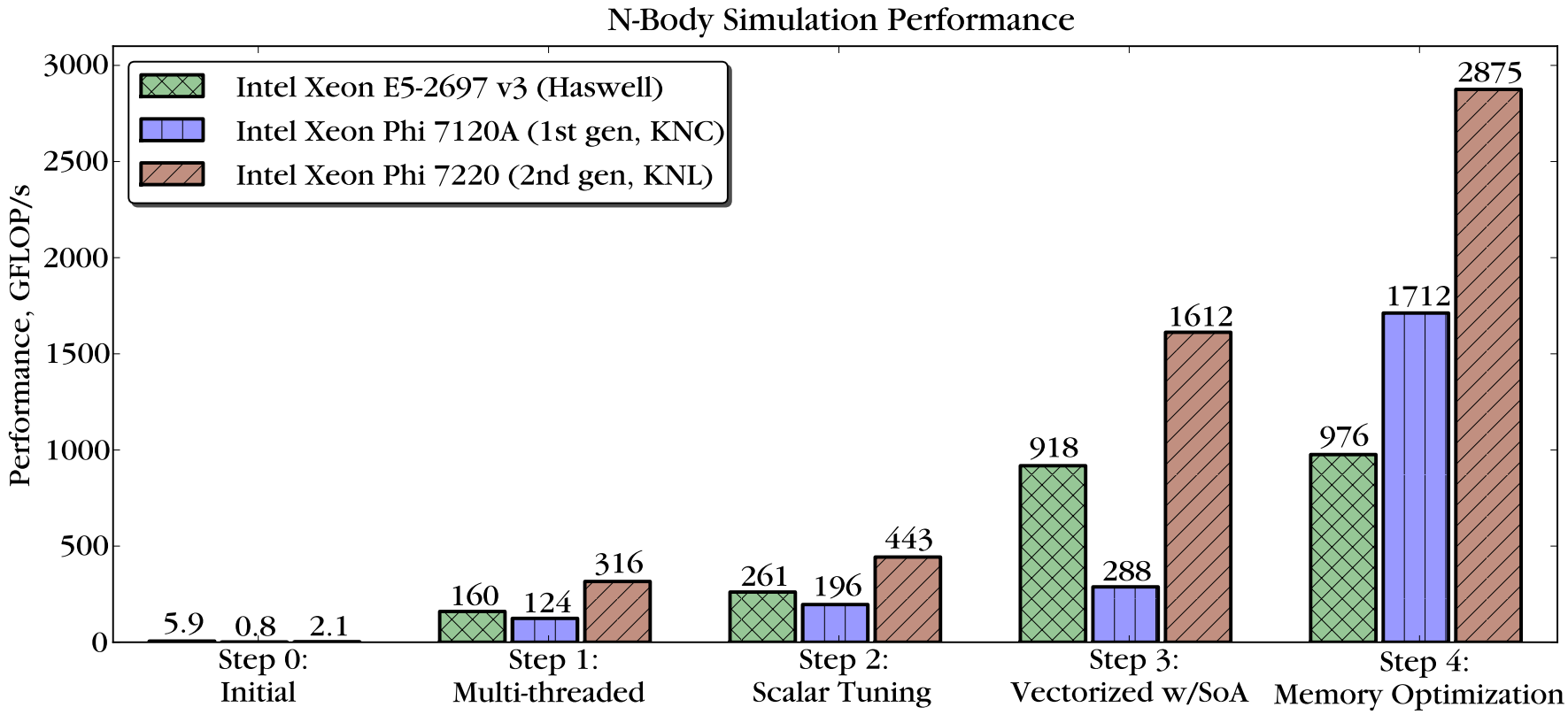
Intel Xeon PHI (2nd Gen)

- Plataforma especializada para aplicações que demandam alta computação
  - Versão em *socket* ou como coprocessador
  - 64-72 cores x 4 HT em 1.3-1.5GHz
  - +3 TFLOP/s in PD (FMA)
  - +6 TFLOP/s in PS (FMA)
  - > 380 GiB DDR4 (> 90 GB/s)
  - 16GiB HBM(MCDRAM, > 400 GB/s)
  - Binário compatível com Xeon





Por que modernização de código?



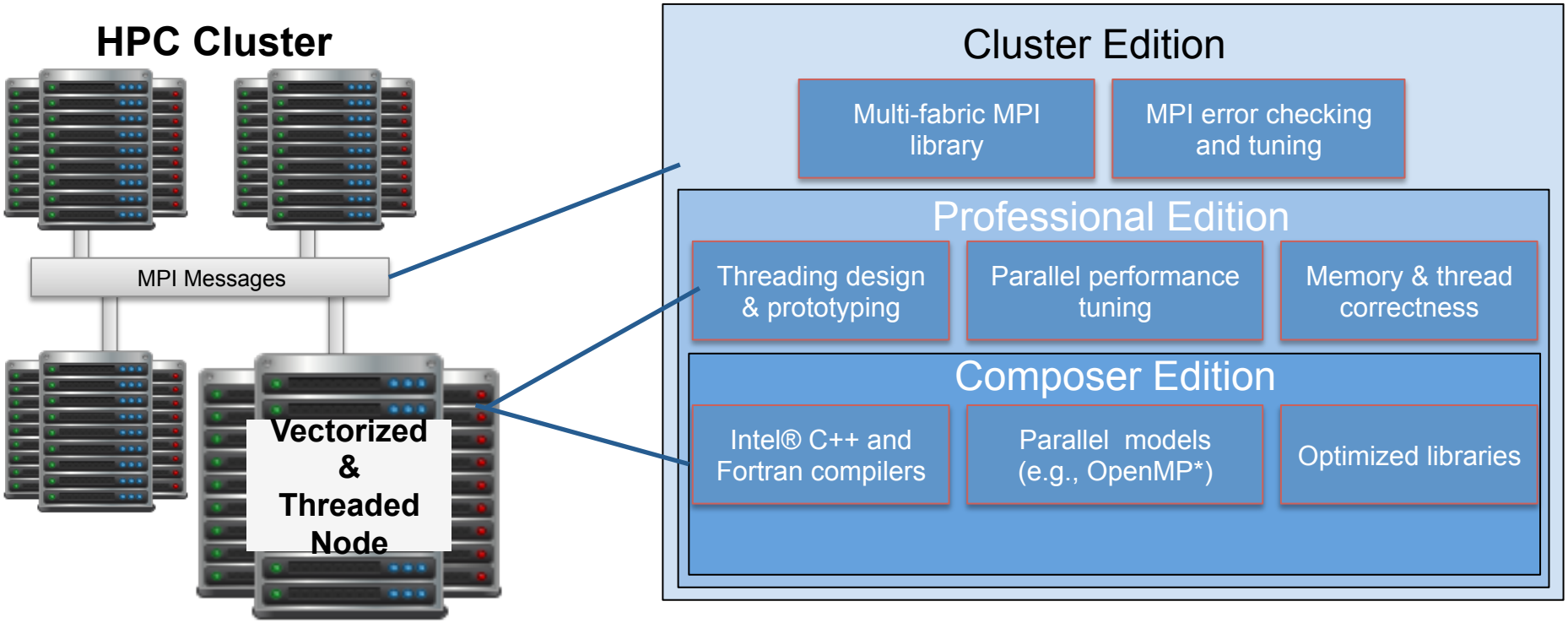
Detalhes sobre a simulação pode ser vista em: lotsofcores.com/KNLbook

IDENTIFICANDO OPORTUNIDADES DE OTIMIZAÇÃO

Identificando oportunidades de otimização

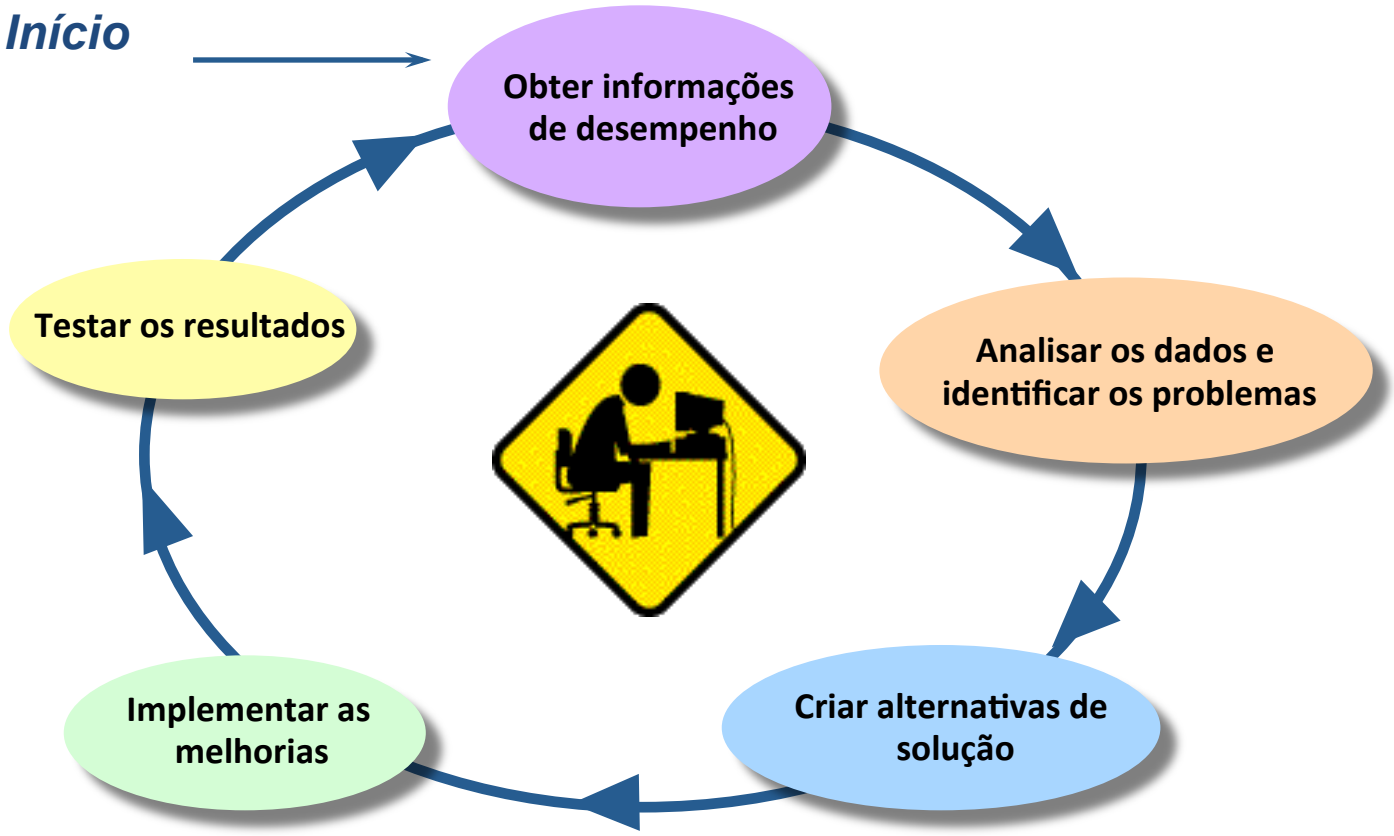
Foco deste seminário: Modernização de código

Otimização em um único "nó" de processamento



Áreas de Otimização

Ciclo de Otimização



Obter Informações de Desempenho

- **Perfilador**: programa capaz de reunir informações sobre o desempenho através de instrumentação de funções, métodos, blocos básicos e declarações.
  - Intel Vtune
  - PAPI
  - HPCToolkit, etc,...

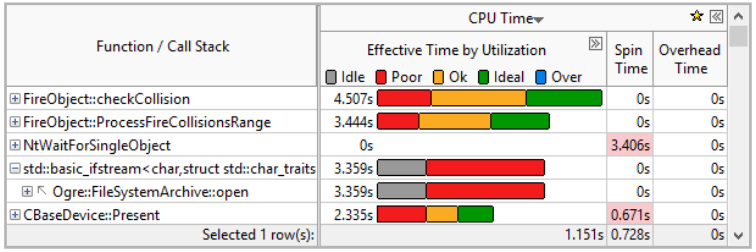
- **Principais métricas de desempenho**:
  - CPU Time
  - Operações de Ponto-Flutuante por segundo FLOPS
  - Contadores de Hardwares: CPI, L1/L2 Cache Misses, etc

Intel® VTune™ Amplifier

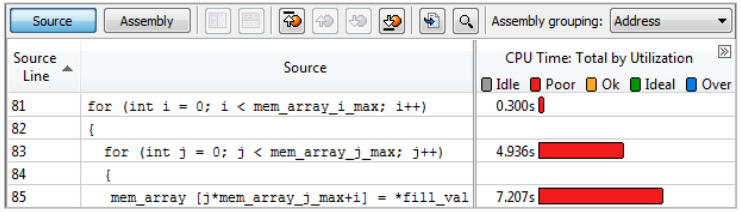
- Get the data you need
  - Hotspot (Statistical call tree), call counts (statistical)
  - Thread profiling – concurrency and locks and waits analysis
  - Cache miss, bandwidth analysis…[1]
  - GPU offload and OpenCL* kernel tracing
- Find answers fast
  - View results on the source/assembly
  - OpenMP* scalability analysis, graphical frame analysis
  - Filter out extraneous data – organize data with viewpoints
  - Visualize thread and task activity on the timeline
- Easy to use
  - No special compiles – C, C++, C#, Fortran*, Java*, ASM*
  - Visual Studio* integration or stand-alone
  - Graphical interface and command line
  - Local and remote data collection
  - Analyze Windows* and Linux* data on macOS*[2]

[1] Events vary by processor. [2] No data collection on macOS*

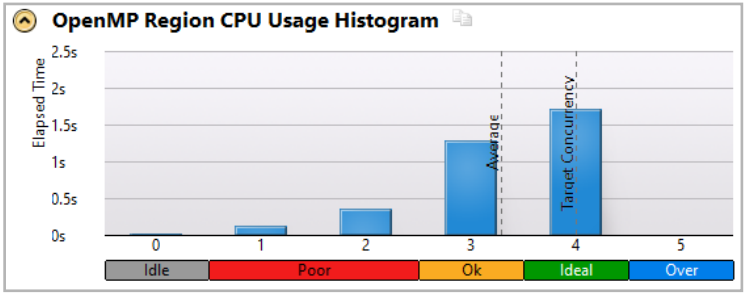
Quickly Find Tuning Opportunities



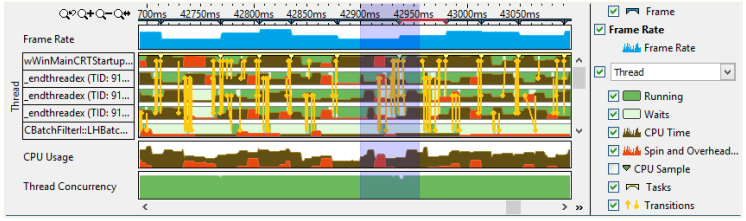
See Results On The Source Code



Tune OpenMP Scalability



Visualize & Filter Data



Técnicas de otimização

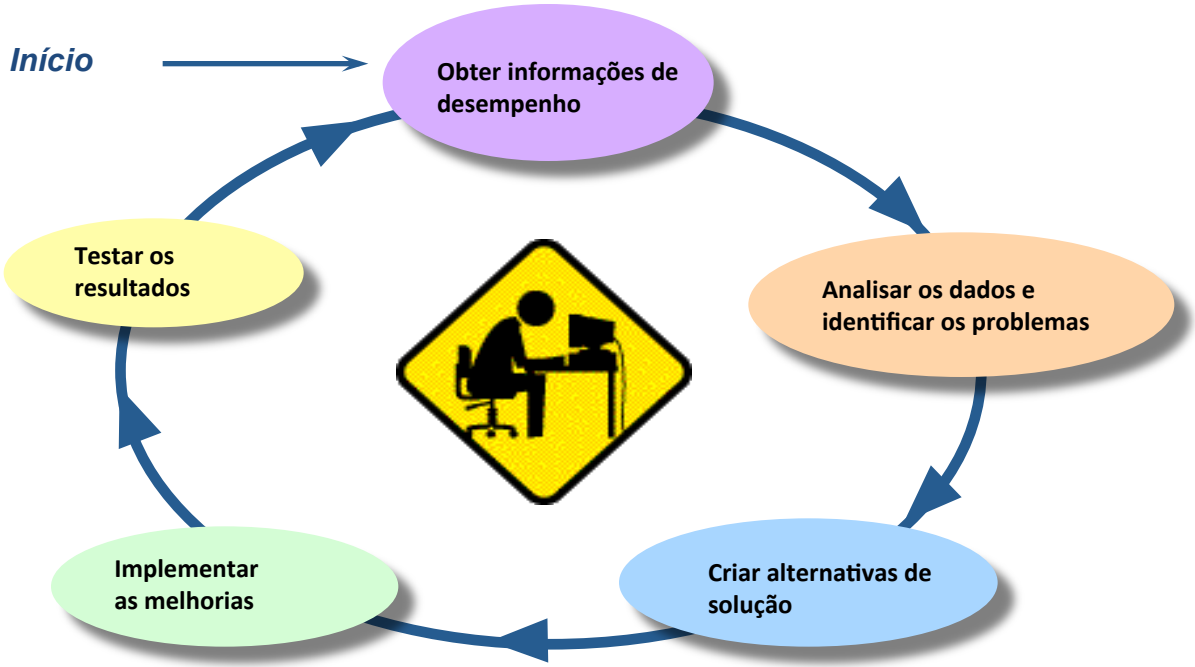
- Regras gerais:
  - Seleção dos melhores algoritmos;
  - Uso de bibliotecas eficientes (não esquecendo da portabilidade);
    - Exemplo: Álgebra Linear (Intel MKL)
  - Escolha do melhor layout de dados;
  - Programar com boas práticas de programação (otimizações manuais);
  - Usar otimizações do compilador **SEMPRE**!!!;

Técnicas de Otimização (cont)

- Otimização manual:
  - O usuário tem maior controle a respeito das dependências de dados;
  - Pode reduzir a legibilidade do programa;
  - Algumas otimizações podem ser benéficas em algumas arquiteturas e maléficas para outras;
- Otimização automática:
  - O compilador possui um conhecimento limitado das dependências de dados;
  - Praticamente impossível em C/C++ se forem utilizados ponteiros indiscriminadamente;
  - Não altera a legibilidade do código;
  - Resultados ótimos de acordo com a arquitetura e a habilidade do compilador;
  - Pode aumentar consideravelmente o tempo de compilação
- Otimização na prática
  - Realizar otimizações manuais básicas e deixar outras otimização por conta do compilador
  - Utilizar otimizações avançadas somente para as arquiteturas predominantes

Otimização Manual

- Forma mais trabalhosa porém a mais efetiva;
  - O programador deve ter controle sobre o que deve ser otimizado
- Baseia-se em:
  - Utilizar boas práticas de programação durante a fase de concepção do software;
  - Por em prática o ciclo de desempenho:



TÉCNICAS BÁSICAS DE OTIMIZAÇÃO

Reduções

Original

```
2*i
```

onde i é um valor inteiro

Melhorado

```
i + i
```

Original

```
2*f
```

onde f é um valor de ponto flutuante

Alternativa

```
f + f
```

essa modificação não traz benefícios pois a adição e a multiplicação de ponto flutuante consomem o mesmo número de ciclos

Original

```
x**2.d0
```

Melhorado

No mínimo, substitua por:

```
x**2
```

ou melhor:

```
x*x
```

Eliminação de sub-expressões

Original

```
s1 = a + c + b
s2 = a + b - c
```

ordem das operações inibindo as otimizações por parte do compilador

Melhorado

No mínimo, substitua por:

```
s1 = (a + b) + c
s2 = (a + b) - c
```

ou melhor:

```
t  = a + b
s1 = t + c
s2 = t - c
```

Original

```
r  = 2.*x(i)
s  = x(i)-1.
```

Melhorado

```
t1 = x(i)
r  = 2.*t1
s  = t1-1.
```

Original

```
x  = acos(-1.) + acos(-1.)**2
```

Melhorado

```
tt = acos(-1.)
x  = tt + tt**2
```

ou melhor:

```
tt = acos(-1.)
x  = tt + tt*tt
```

Muitas dessas otimizações podem ser automaticamente feitas pelo compilador

Expressões constantes em laços

Original

```
do i=1,n
   a(i) = r*a(i)*s
enddo
```

ordem das operações inibindo as otimizações por parte do compilador

Melhorado

No mínimo, substitua por:

```
do i=1,n
   a(i) = r*s*a(i)
enddo
```

ou melhor:

```
t1 = r*s
do i=1,n
   a(i) = t1*a(i)
enddo
```

Conversão de sinais

- A conversão de sinais consome alguns ciclos de processamento;
- A omissão das conversões, em operações de ponto flutuante, podem alterar o resultado das operações, ou seja, o compilador não tenta fazer tais transformações em níveis baixos de otimização;

Original

```
a(i) = -(1.d0-.5d0*b(i))
```

Melhorado

```
a(i) = -1.d0+.5d0*b(i)
```

Propagação e avaliação de constantes e simplificação no cálculo de endereços

Original

```
two = 2.d0
x = 3.d0*two*y
```

Melhorado

```
x = 6.d0*y
```

Original

```
dimension a(4,100)
do i=1,100
   x = x+a(1,i)
enddo
```

o endereço de `a(1,i)` em relação ao elemento `a(1,1)` é `4*i-4`

Melhorado

```
dimension a(4*100)
iadd = 1
do i=1,100
   x = x+a(iadd)
   iadd = iadd+4
enddo
```

aqui o endereço é calculado por uma simples adição

In-lining

- Função sendo chamada dentro de um laço:

```
...
do i=1,max
   r(i) = dist(x(i),y(i))
enddo
...
function dist(x,y)
real*8 :: x, y, dist
dist = dsqrt(x*x + y*y)
end function
```

- A técnica de "*in-lining*" consiste em substituir as chamadas de rotinas escrevendo-as explicitamente onde elas são chamadas:

```
...
do i=1,max
   r(i) = dsqrt(x(i)*x(i)+y(i)*y(i))
enddo
...
```

Os compiladores possuem flags específicos para realizar esse tipo de substituição baseado em alguns critérios, porém, quando efetuamos essa substituição explicitamente facilitamos o trabalho do compilador.

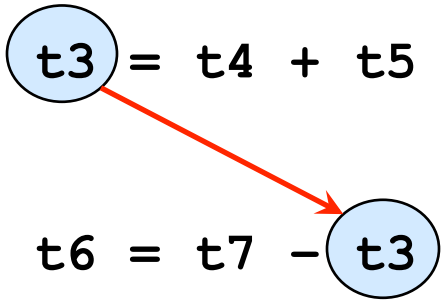
Precisão: Constantes

```
// bad: 2 is "int"
long b = a*2;

// bad: overflow
long n = 100000*100000;

// bad: excessive
float p=6.283185307179586;

// Bad: 2 is "int"
float q = 2*p;

// Bad: 1e9 is "double"
float r=1e9*p

// bad: 1 is "int"
double t=s+1
```

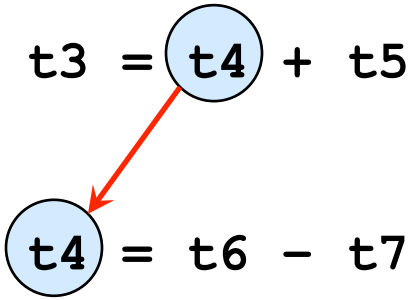
```
// Good: 2 is "long"
long b = a*2L;

// Good: correct
long n = 100000L*100000L;

// Good: accurate
float p=6.283185f;

// Good: 2 is "float"
float q = 2.0f*p;

// Goof: 1e9 is "float"
float r=1e9f*p

// Good: 1.0 is "double"
double t=s+1.0
```

Tipos de dependências e instruções FMA

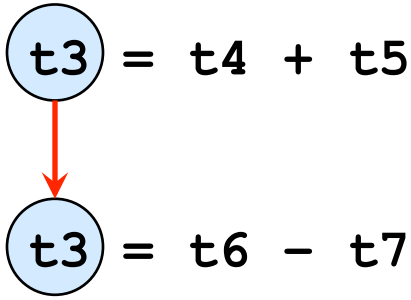
Dependência verdadeira



Anti-dependência



Dependência de saída



Fused multiply-add (FMA)

Original

```
x = 2.*y+1.
s = s+x
z = 3.*r+2.
t = t+z
```

Melhorado

```
x = 2.*y+1.
z = 3.*r+2.
s = s+x
t = t+z
```

Evite sempre!

- Testes no interior de laços intensivos

```
do i=1,1000
  if (i.lt.50) ! do something
enddo
```

- Chamadas de rotinas no interior de laços intensivos

```
do i=1,1000
   call something
enddo
```

- Operações de divisão *"...custam muito mais ciclos de processamento do que as operações de multiplicação..."*

- Operações de IO no interior de laços intensos
- Variáveis não iniciadas;

- *Arranjos* com dimensões superiores a 2;

- Aritmética de arranjos do Fortran90 *"...facilita a vida do programador mas deixa o desempenho inteiramente sob responsabilidade do compilador."*

- Sobrecarga de operadores (C++ e Fortran90)

- "Aliasing": 2 arranjos compartilhando a mesma área de memória
  - `call sub(n,a(1),a(1),c(1),sum)`

Otimização automática

- É a forma mais simples e rápida de se obter ganhos de desempenho, portanto, UTILIZE SEMPRE independentemente de ter feito otimização manual ou não;

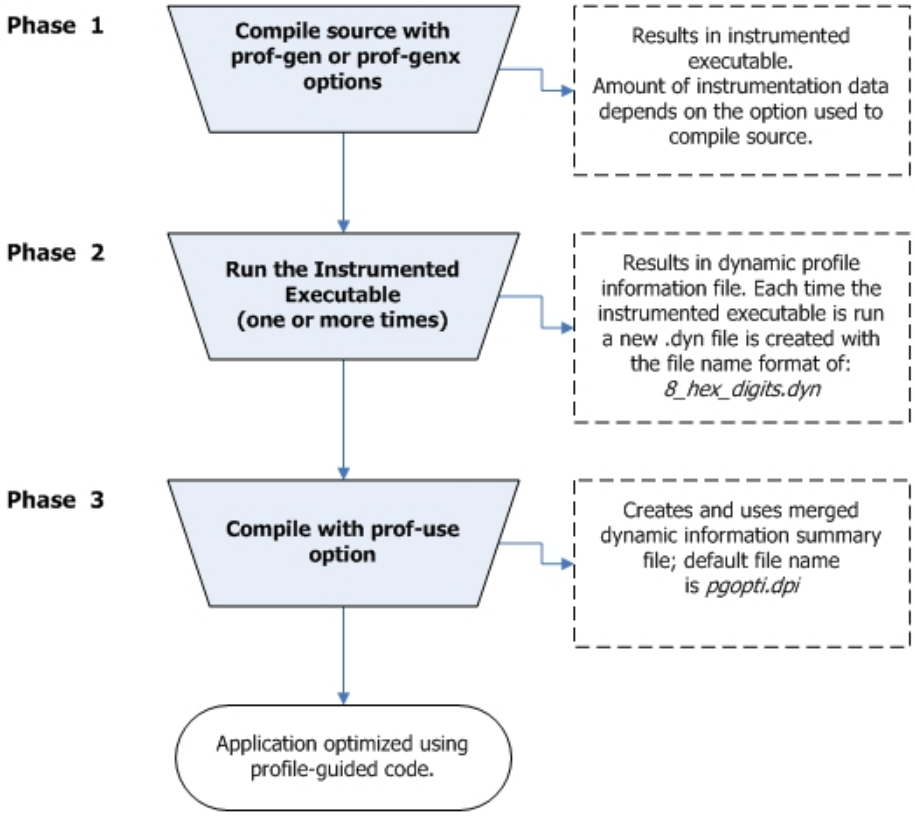
- Para gerar um código otimizado o compilador realiza algumas operações bastante similares a algumas técnicas de otimização manual, tais como: alteração da ordem de algumas instruções, desenrolamento de laços, *inlining*, etc...

Otimização automática (cont.)

- Armadilhas dos flags de otimização:
  - Um compilador normalmente possui dezenas de opções de otimização que podem ser usadas individualmente ou agrupadas entre si. Nem sempre a associação dos flags de otimização produzem o melhor resultado. É importante ressaltar que em alguns casos uma associação de flags mal-feita pode acarretar em efeito contrário ("*pessimização*").
- Escolhendo os flags de otimização que devem ser usados
  - Teste os flags recomendados pelo fabricante do compilador;
  - Utilize os flags mais conhecidos e avalie as diferenças;
- OBSERVAÇÃO CRUCIAL:
  - Os flags de otimização podem influenciar severamente no resultado de alguns cálculos, portanto, certifique-se que o programa otimizado está produzindo o resultado correto.

Otimização assistida

- Alguns compiladores possuem opções específicas para a geração de relatórios de avaliação de desempenho;
  - Compila-se o programa com essas opções ativadas;
  - Executa-se uma rodada normal do software. Será produzido o relatório de otimização;
  - Recompile o software fornecendo o relatório de otimização para o compilador;
  - Baseado no relatório de otimização o compilador tentará produzir uma versão de código binário mais eficiente que o primeiro.
- Enquanto houver algum ganho esse ciclo pode ser repetido



Alguns fabricantes de compiladores chamam este procedimento de PGO Profile Guided Optimization

Otimização assistida (cont.)

- Fase 1:

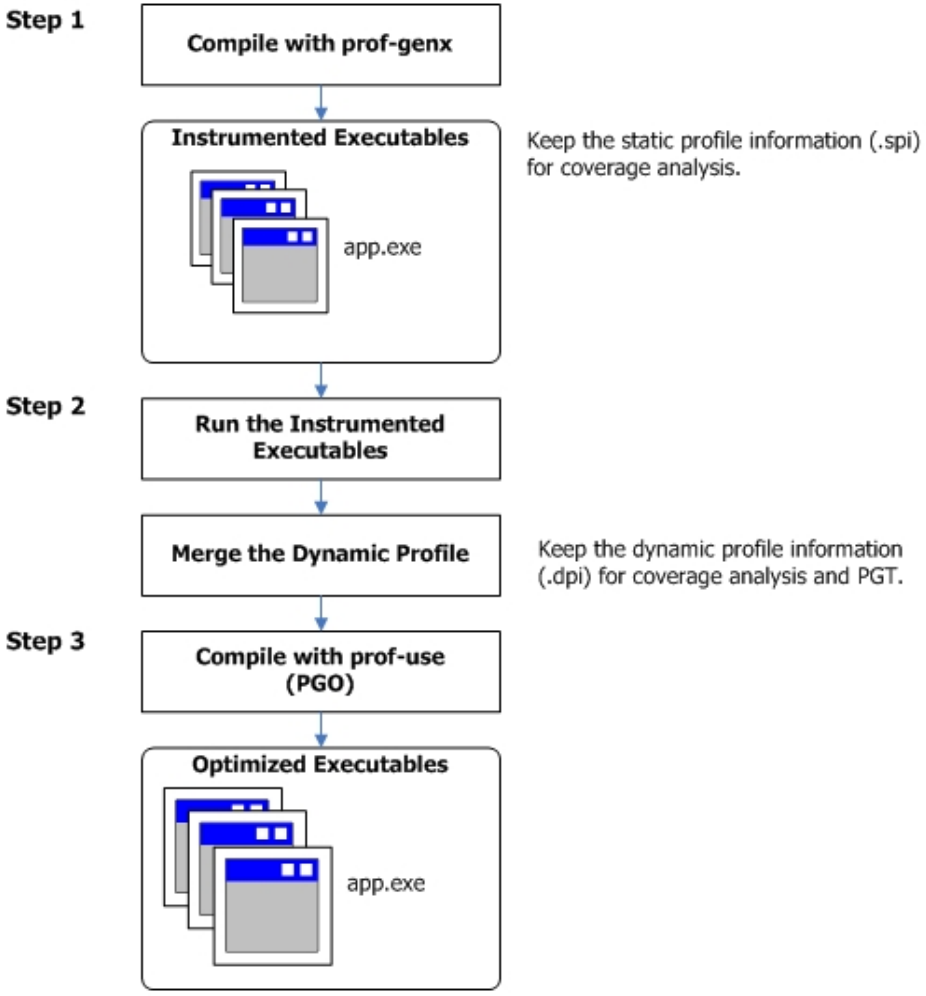
```
ifort -prof-gen -c foo.f
ifort -o foo foo.o
```

- Fase 2:

```
./foo
```

- Fase 3:

```
ifort -prof-use -c foo.f
ifort -o foo foo.o
```



Flags Compilador

Funcionalidade	Linux
Desabilita otimizações	-O0
Otimização (não aumenta tamanho do código)	-O1
Otimização (habilita vetorização, otimizaçoes de laços)	-02
Otimização alta-ordem (loop unrol)	-03
Vectorization	<várias Opções>
Otimização Agressiva (-ipo, -03, -no-prec-div, -xHost)	-fast
Criação de informações para depuração	-g
Gera arquivo Assembly	-S
Geração do relatório de otimização	-qopt-report=<n>
Suporte OpenMP	-qopenmp

LAB01- Otimização Automática

- Objetivos:
  - Explorar opções de otimização do compilador a partir de uma implementação de uma algoritmo de atualização vetorial do tipo: SAXPY
  - Demonstrar o ganho de desempenho com a utilização de bibliotecas especializadas.

Lab 01 – SETUP UP

- Obter dados do *handson* no site do workshop.
- Copiar os arquivos do hands-on para seu diretório home.
  - cp handson.tar.gz ${HOME}/.
- Descompacta-lo:
  - tar xzvf handson.tar.gz

LAB1: SAXPY

- Localização dos arquivos:
  - `~/handson/lab01/saxpy`
- Para compilar sem otimização: -O0
  - `icc -g -O0 saxpy1.cpp -o saxpy.exe`
- Compile com as opcões -O1, -O2 e -O3 -xHost
- Veja o relatório de otimização:
  - `acrescente a flag -qopt-report`

LAB01: Multiplicação Matricial

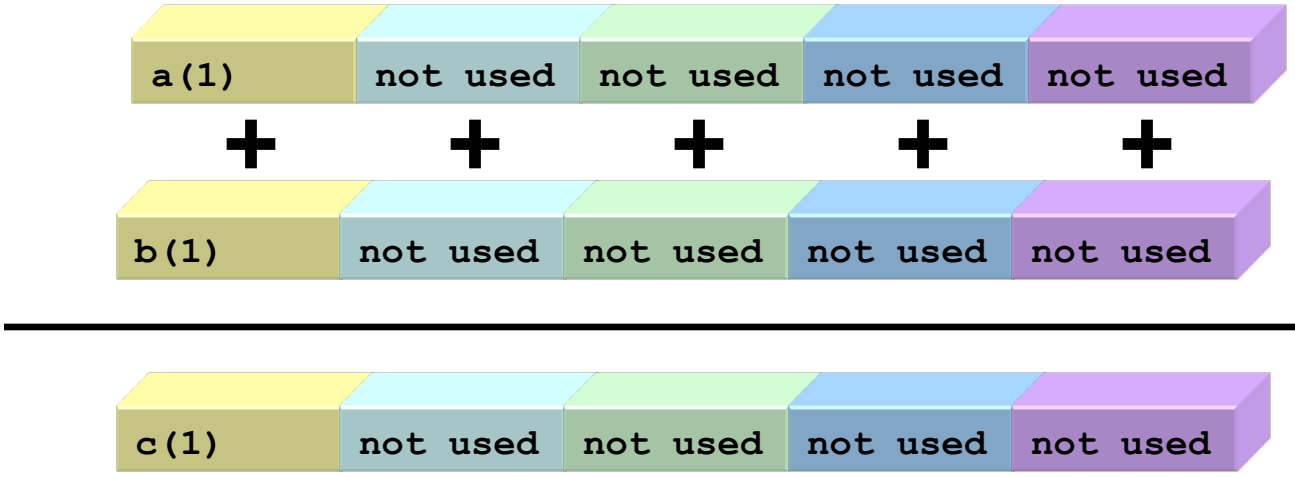
- Objetivos: Demostrar o uso de bibliotecas eficientes
- Localização do código-fonte:
  - `~/handson/lab01`
- Para compilar:
  - Versão ingênua: `$make step1`
  - Versão usando MKL Blas: `$make step2`
- Compare os tempos de processamento.

VETORIZAÇÃO

Vetorização: Como funciona?

- Uma operação ESCALAR pode ser ilustrada por:

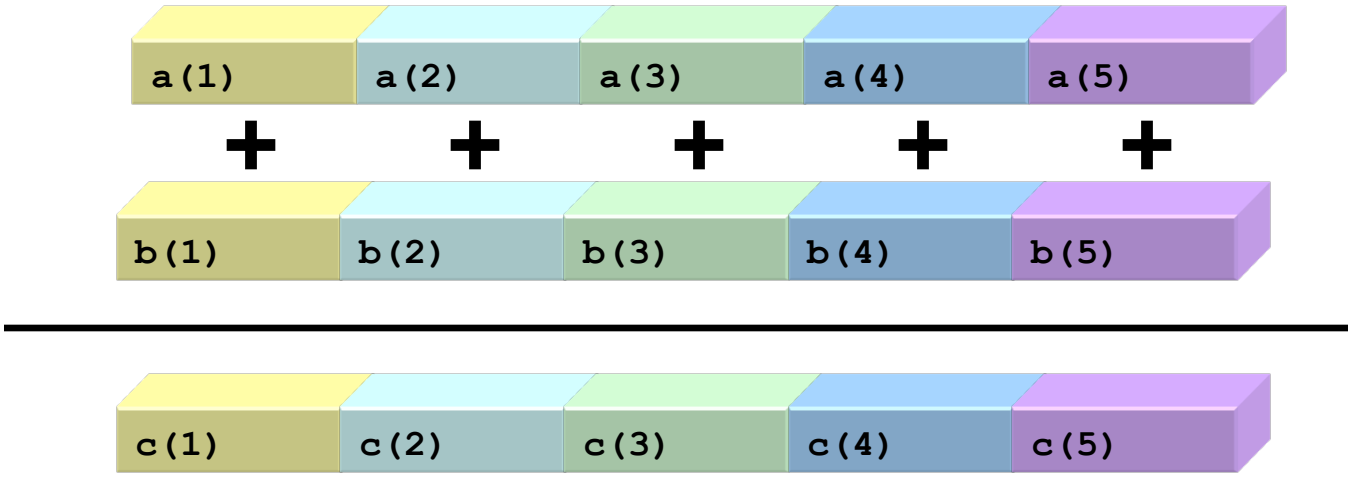
```
do i=1,n
   c(i) = a(i) + b(i)
enddo
```



Vetorização: Como funciona? (cont.)

- Executando-se a mesma operação em um processador VETORIAL teríamos:

```
do i=1,n
   c(i) = a(i) + b(i)
enddo
```



ou seja, poderíamos operar em 5 posições dos vetores simultaneamente a cada ciclo do processador (5 x mais rápido!).

Vetorização - Perspectivas

- Perspectiva para o hardware: Instruções, registradores ou unidades funcionais especializadas que permitem o paralelismo das operações no interior do núcleo em arranjos de dados (vetores)

- Perspectiva do Compilador: determinar como e quando é possível expressar computações em termos de instruções vetoriais

- Perspectiva do usuário: determinar como escrever códigos de uma maneira que permita ao compilador deduzir que a vetorização é possível

Vetorização: Hardware

- Objetivo: paralelizar computação sobre arranjos vetoriais
- Duas principais técnicas: *pipelining* e SIMD (*Single Data Multiple Data)*
- *Pipelining*: Diversas tarefas distintas executando simultaneamente
- SIMD: Muitas instâncias de uma simples tarefa executando simultaneamente
  - Vetores menores, poucos ciclos por instrução
  - Novos CPUs podem usar o *pipelining* em algumas instruções SIMD

Pipelining: como funciona?

Clock Cycle	R1	R2	R3	R4
1	X_1			
2	X_2	X_1^2		
3	X_3	X_2^2	X_1^2+8	
4	X_4	X_3^2	X_2^2+8	$(X_1^2+8)/2$
5	X_5	X_4^2	X_3^2+8	$(X_2^2+8)/2$
6		X_5^2	X_4^2+8	$(X_3^2+8)/2$
7			X_5^2+8	$(X_4^2+8)/2$
8				$(X_5^2+8)/2$

Load

Square

Add

Divide

Vetorização e Pipelining

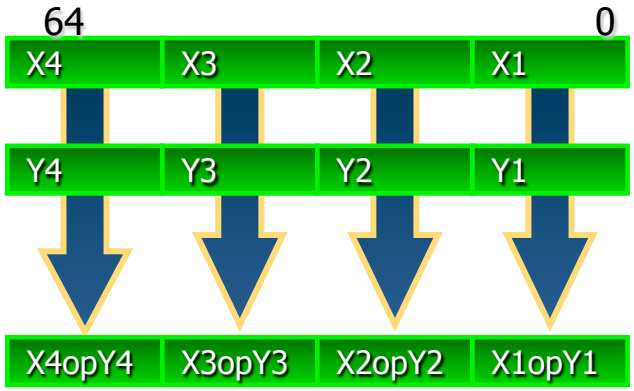
Clock	R1	R2	R3	R4
1	$[X_1, X_2 \ldots X_5]$			
2		$[X_1^2, X_2^2 \ldots X_5^2]$		
3			$[X_1^2+8, X_2^2+8 \ldots X_5^2+8]$	
4	Load	Square	Add	$[(X_1^2+8)/2,$ $(X_2^2+8)/2$ … $(X_5^2+8)/2]$
				Divide

Vetorização: Motivação

- Velocidade dos CPUs alcançou um patamar
  - Limitações de consumo
  - Muitos transistores “lentos” são mais eficiente que poucos transistores “rápidos”

- Melhorias de processo ocasionou barateamento do espaço físico
  - Uma solução: Adicionar mais cores
    - Primeiro Intel dual cores CPUs aparece em 2005.
    - Número cresce rapidamente (+60 no MIC)
  - Outra solução: + FPU por core – Operações vetoriais
    - Surgimento no Pentium com MMX em 1996
    - Rápido aumento da largura do vetor (512-bit [8 doubles]) no MIC

Evolução do processamento vetorial dos processadores Intel®



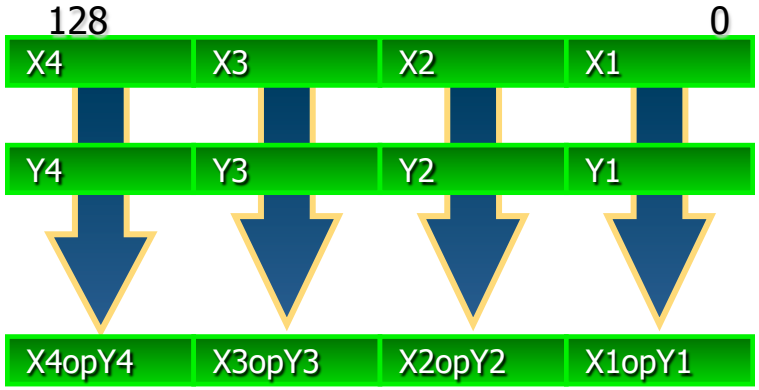
MMX™

Vector size: 64bit

Data types: 8, 16 and 32 bit integers

VL: 2,4,8

For sample on the left: Xi, Yi 16 bit integers



Intel® SSE

Vector size: 128bit

Data types:

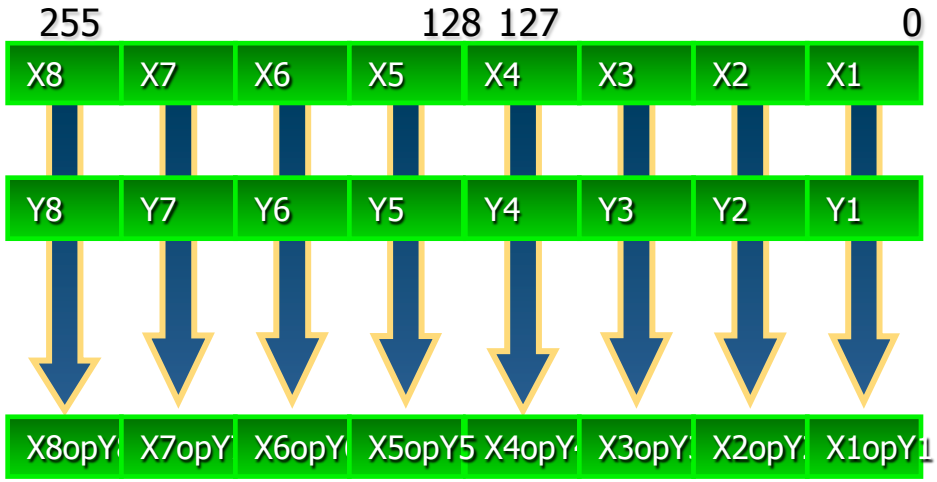
8,16,32,64 bit integers

32 and 64bit floats

VL: 2,4,8,16

Sample: Xi, Yi bit 32 int / float

Evolução do processamento vetorial dos processadores Intel®



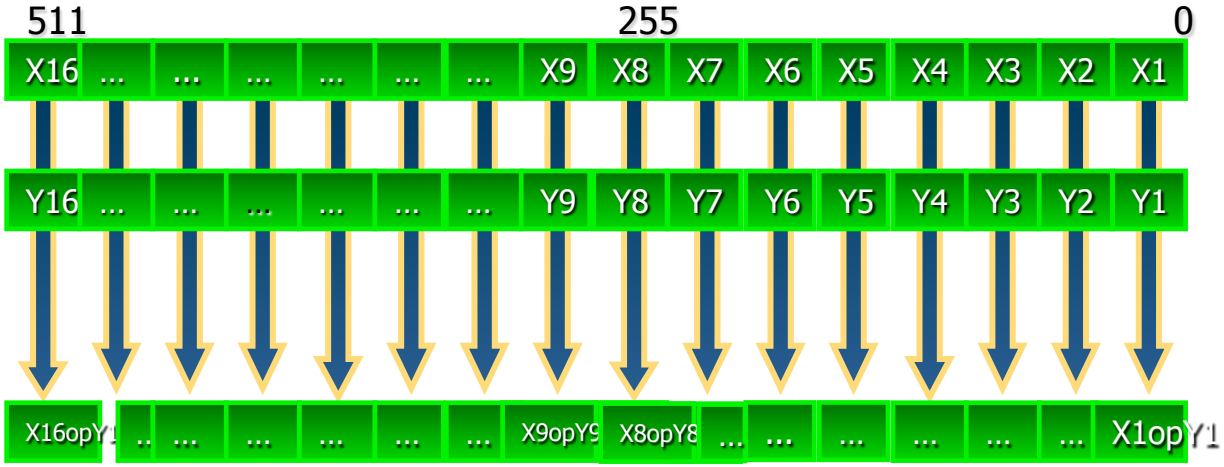
Intel® AVX / AVX2

Vector size: 256bit

Data types: 32 and 64 bit floats

VL: 4, 8, 16

Sample: Xi, Yi 32 bit int or float



Intel® MIC / AVX-512

Vector size: 512bit

Data types:

32 and 64 bit integers

32 and 64bit floats

(some support for 16 bits floats)

VL: 8,16

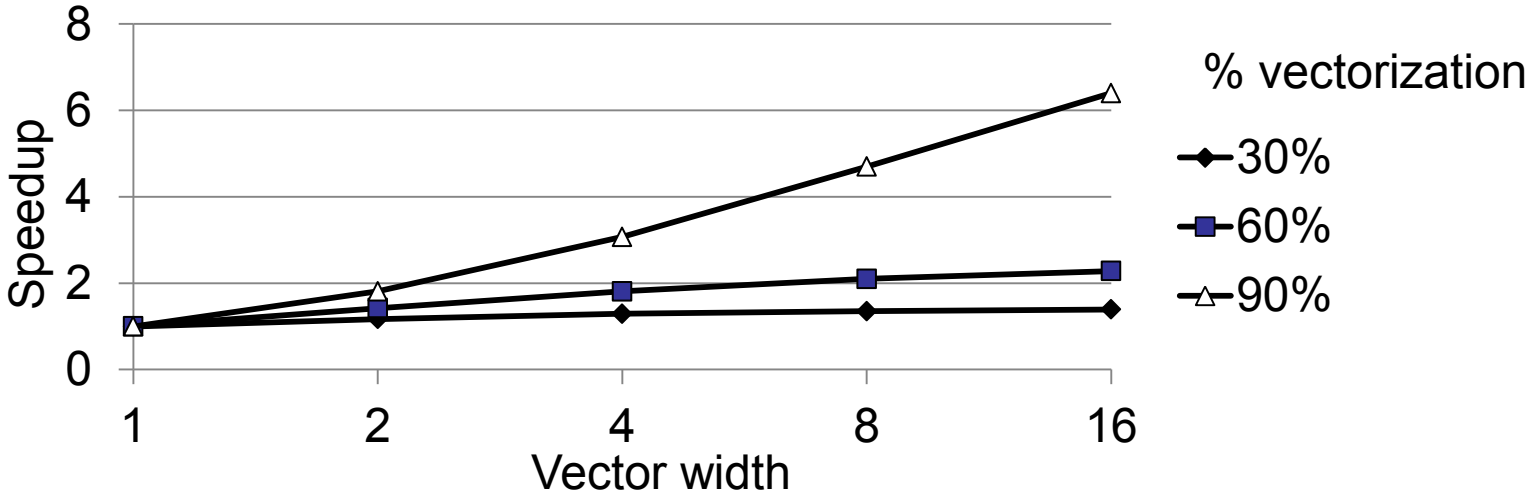
Sample: 32 bit float

Speedup

- Paralelismo SIMD – tipicamente 1 ciclo por operação com ponto-flutuante
  - Exceções: divisões, sqrt, sin
- Speedup (comparado com código sem vetorização)
  - 128-bit SSE – 2x double, 4x single
  - 256-bit AVX – 4x double, 8x single
  - 512-bit AVX2 – 8x double, 16x single
- Exemplo AVX hipotético: 8 cores/CPU*4 double/vetor * 2.0 GHz = 64 Gflops/CPU
  - Pipelining pode melhorar ainda mais!

Speedup

- Largura de banda da memória pode ser um problema, vamos explorar isso mais tarde
  - Utilização ineficiente da cache, alinhamento, latência
- SIMD é paralelo → lei de Amdahls é efetivo!
  - Porções seriais/escalares do código ou CPU são limitadores
  - *Speedup* teórico é somente um limitante superior.

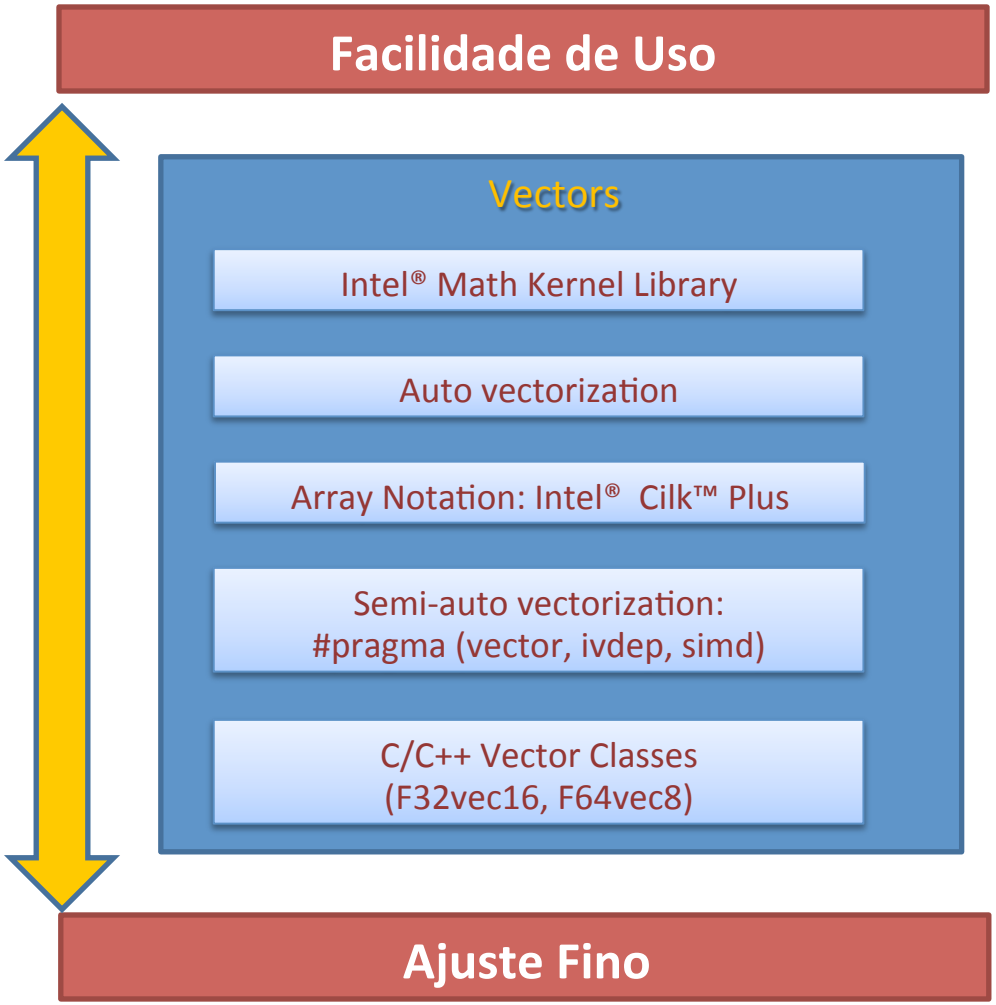


Perspectiva do Usuário

- Como podemos aproveitar esse poder?
  - Codificação *Assembly*
    - Potencial ganho de desempenho final mas somente para os bravos
  - Funções Intrínsecas
    - Passo anterior ao *assembly*. Útil mas arriscado
  - Deixe o compilador descobrir como… (auto-vetorização)
    - Relativamente simples, desafiador para o compilador
    - O compilador pode necessitar de uma ajuda
  - Link com um biblioteca otimizada que faça o trabalho real
    - Exemplo: Intel MKL, escrito por pessoas que sabem todos os truques
    - Obtém os benefícios quando executados em plataformas suportadas

Identificando oportunidades de otimização

Maneiras de vetorizar o código



- Devemos avaliar três fatores:
  - Necessidade de performance
  - Disponibilidade de recursos para otimizar o código
  - Portabilidade do código

O Que eu preciso fazer?

- Conhecer o que faz ser “vetorizável”
  - Laços “for” em C/C++ ou “do” em Fortran que atendam a certas restrições
- Conhecer onde a vetorização irá ajudar
- Avaliar a saída do compilador
  - Ele está realmente vetorizado onde eu acho que deveria estar?
- Avaliar o desempenho da execução
  - Comparar com o *speedup* teórico
- Conhecer o padrão de acesso ao dados para maximizar a eficiência
- Implementar correções: diretivas, *flags* compilador e alteração do código
  - Remover construções que fazem vetorização impossível
  - Forçar vetorização quando compilador não faz, mas que deveria
  - Melhorar o padrão de acesso a memória.

Escrevendo laços “vetorizáveis”

- Requerimentos básicos:
  - Contável em tempo de execução
    - Número de iterações conhecido antes de sua execução
    - Sem terminação condicional (sem breaks)
  - Ter um fluxo de execução simples
    - Sem saltos condicionais
    - If são permitidos quando eles podem ser implementados com máscaras
  - Deve ser o laço mais interno se houver laços aninhados
    - Compilador pode reverter a ordem
  - Evitar chamadas de função
    - Funções básicas matemáticas são permitidas: *pow, sqrt, sin*
    - Algumas funções inline são permitidas

Auto-vetorização

- Suporte compilador Intel:
  - Vetorização é realizado a partir do nível de otimização –O2
  - Default para instruções SSE
  - Pode incorporar instruções SSE e AVX em um mesmo binário com a flag `-axAVX`
    - Rodará AVX em CPUs com AVX suporte, SSE caso contrário.
  - `-qopt-report=<n>` para gerar um relatório de vetorização.
  - AVX512: Suporte para versões > 15.0.
- GCC (>4.9):
  - Vetorização é desabilitado por default
  - Use a flag `-ftree-vectorize` combinado com nível otimização > -O2
  - Padrão instruções SSE, `-mavx -march=corei7-avx` para AVX
  - `-ftree-vectorizer-verbose` para gerar o relatório.

Desafio
Dependência de dados

- Vetorização pode alterar a ordem da computação em comparação de um caso escalar
- Compilador deve assegurar que a vetorização irá produzir resultados corretos
- Necessidade de considerar operações independentes em laços desenrolados – depende do tamanho do vetor
- Compilador executa uma análise de dependência

Dependências: leitura após escrita

Considere o laço:

a = {0,1,2,3,4}

b = {5,6,7,8,9}

```
for(i=1;i < N; i++)
    a[i] = a[i-1] + b[i]
```

Aplicando cada operação sequencialmente:

a[1] = a[0] + b[1] → a[1] = 0 + 6 → a[1] = 6

a[2] = a[1] + b[2] → a[2] = 6 + 7 → a[2] = 13

a[3] = a[2] + b[3] → a[3] = 13+8 → a[3] = 21

a[4] = a[3] + b[4] → a[4] = 21+9 → a[4] = 30

a={0,6,13,21,30}

Dependências: Leitura após escrita

Vamos tentar vetorizar:
a = {0,1,2,3,4}
b = {5,6,7,8,9}

```
for(i=1;i < N; i++)
    a[i] = a[i-1] + b[i]
```

Aplicando operação vetorial, i={1,2,3,4}:
a[i-1] = {0,1,2,3} (load)
b[i] = {6,7,8,9} (load)
{0,1,2,3} + {6,7,8,9} = {6,8,10,12} (operate)
a[i] = {6,8,10,12} (store)

a = {0,6,8,10,12} != {0,6,13,21,30} **NÃO VETORIZADO**

Dependências: escrita após leitura

Vamos tentar vetorizar:
a = {0,1,2,3,4}
b = {5,6,7,8,9}

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

Aplicando cada operação sequencialmente:

A[0] = a[1] + b[0] → a[0] = 1 + 5 → a[1] = 6
A[1] = a[2] + b[1] → a[1] = 2 + 6 → a[2] = 8
A[2] = a[3] + b[2] → a[2] = 3 + 7 → a[3] = 10
A[3] = a[4] + b[3] → a[3] = 4 + 8 → a[4] = 12
a={6,8,10,12,4}

Dependências: Leitura após escrita

Vamos tentar vetorizar:
a = {0,1,2,3,4}
b = {5,6,7,8,9}

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

Aplicando operação vetorial, i={1,2,3,4}:
a[i-1] = {1,2,3,4} (load)
b[i] = {5,6,7,8} (load)
{1,2,3,4} + {5,6,7,8} = {6,8,10,12} (operate)
a[i] = {6,8,10,12} (store)

q = {0,6,8,10,12} == {0,6,8,10,12} **VETORIZADO**

Tipos de Dependências

- Leitura após escrita:
  - Também conhecida como dependência de fluxo
  - Variável escrita primeiro e depois lida
  - Não vetorizável

```
for(i=1;i < N; i++)
   a[i] = a[i-1] + b[i]
```

- Escrita seguida por leitura
  - Conhecida como anti dependência
  - Variável lida primeiro, depois escrita
  - Vetorizável

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

Tipos de Dependências

- Leitura após leitura:
  - Na verdade não é um dependência
  - vetorizável

```
for(i=0;i < N; i++)
   a[i] = b[i%2] + c[i]
```

- Escrita após escrita
  - Dependência de saída
  - Variável escrita e reescrita
  - Não vetorizável

```
for(i=0;i < N; i++)
   a[i%2] = b[i] + c[i]
```

Dependência em Laços: "Aliasing"

- Em C, ponteiros podem esconder dependências de dados
  - Pode haver sobreposição de memória
- O código abaixo é seguro?

```
void compute(double *a, double *b,
             double *c)
{
   for(i =1; i < N; i++) {
   a[i] = b[i] + c[i];
   }
}
```

Não se forem dados os argumentos: compute(a, a+1,c)

Dependência em Laços: Aliasing

- C99 introduz a palavra chave *restrict* na linguagem
  - Diz para o compilador assumir que os endereços não irão sobrepor.
  - Pode ser necessário usar as *flags*: -restrict, -std=c99

```
void compute(double * restrict a,
             double * restrict b,
             double * restrict c)
{
   for(i =1; i < N; i++) {
   a[i] = b[i] + c[i];
   }
}
```

Relatório de Vetorização

- Mostra quais laços foram ou não vetorizados, e porque
- Intel: -qopt-report=<n>
  - 0: nenhuma informação
  - 1: lista laços vetorizados
  - 2: lista laços não vetorizados, com explicação
  - 3: saída adicional com informações de dependência
  - 4: lista laços não vetorizados, sem explicação
  - 5: lista laços não vetorizados, com informação de dependência

- Relatórios são essenciais para determinar onde o compilador encontrou uma dependência

- Compilador é conservador. Você precisa ir ao código e verificar se realmente há uma dependência.

Sugestões para Vetorização

- Compilador deve comprovar que não há nenhuma dependência de dados que irá afetar a exatidão do resultado
- Entretanto, as vezes isso é impossível
  - Ex. Uso complicado dos ponteiros
- Solução dos compiladores Intel: diretiva IVDEP (Ignore Vector DEPendence)
  - Diz ao compilador: assume nenhuma dependência

```
subroutine
vec1(s1,M,N,x)
…
!DEC$ IVDEP
do i = 1,N
 x(i) = x(i+M) + s1
end do
```

```
void vec1(double s1,int M,
   int N,double *x) {
…
#pragma IVDEP
for(i=0;i<N;i++) x[i]=x[i+M]+s1;
```

Diretivas que auxiliam a vetorização

- Antes do laço: #pragma (C/C++) !DIR$ (para Fortran)
- `ivdep`
  - Instrui o compilador a ignorar as dependências assumidas
  - Compilador pode vetorizar laços que ele inicialmente considerou não vetorizável
- `vector always`
  - Sobrescreve as heurísticas do compilador que determinam se a vetorização é susceptível de produzir ganhos de desempenho;
- `novector`
  - Não vetorize
- `omp simd` (OpenMP 4.0)
  - Vetoriza laço (mais poderoso que `vector always`)

Exemplos (C/C++)

```
void ignore_vec_dep(int *a, int k, int c, int m) {
  #pragma ivdep
  for (int i = 0; i < m; i++)
    a[i] = a[i + k] * c;
}
```

```
void vec(int *a,int *b,int m){
  #pragma vector always
  for(int i = 0; i <= m; i++)
    a[32*i] = b[99*i];
}
```

```
   #pragma omp simd collapse(2)
   for(i=0; i<N; i++) { a[i] = b[i] * c[i];
      for(i=0; i<N; i++) { d[i] = e[i] * f[i]; }
}
```

Exemplos (Fortran 90)

```
!dir$ ivdep
do i = 1, m;
   a(i) = a(i + k) * c;
enddo
```

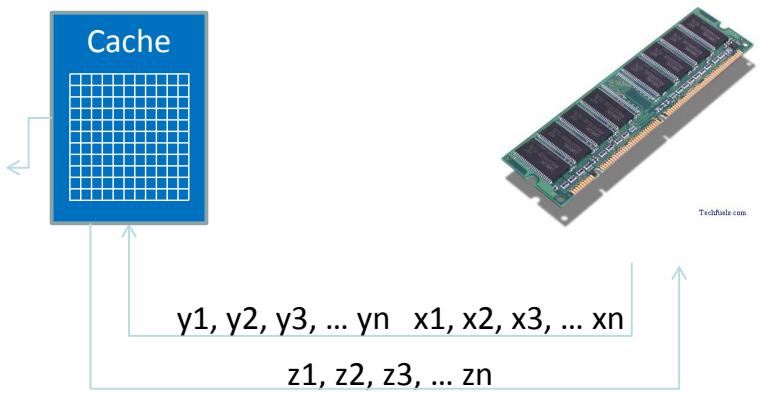
```
!dir$ vector always
do i = 1, m
   a(32*i) = b(99*i)
enddo
```

```
!$omp simd collapse(2)
do j=1, N
    do i=1, N
       a(i,j) = a(i,j) * b[i]
     enddo
enddo
```

CACHE e Alinhamento

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_n \end{bmatrix} = a * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}$$

ymm2 ymm0 ymm1



- Vetorização eficiente requer preocupações que vão além da unidade SIMD
  - Registradores: alinhamento de dados nos limites de 128, 256 e 512 bits
  - Cache: Cache é rápida, memória é lenta
- Memória: acesso sequencial muito mais rápido que acesso randômico ou espaçado.
- Se as unidades vetoriais ficam esperando dados, a eficiência é drasticamente prejudicada.

Acesso com passo largo

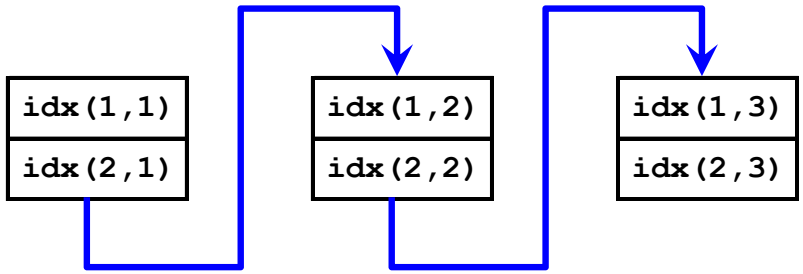
- Padrão de acesso mais rápido é passo unitário
- Melhor desempenho quando CPU pode carrega L1 Cache da memória em blocos de dados, sequencialmente.
- Construções Passo Unitário:
  - arranjos multidimensional
    - Fortran: passo unitário na dimensão mais interna
    - C/C++: passo unitário na dimensão mais externa

```
do j = 1,n; do i=1,n
   a(i,j)=b(i,j)*s
enddo; endo
```

```
for(j=0;j<n;j++)
for(i=0;i<n;i++)
   a[j][i]=b[j][i]*s;
```

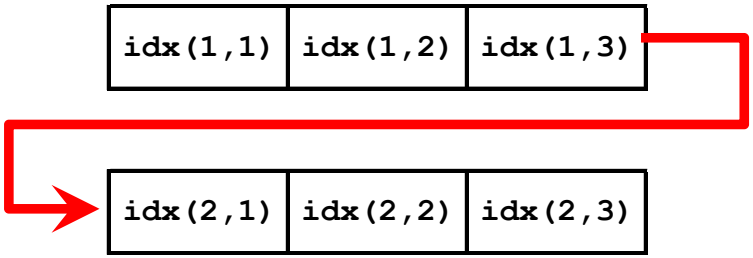
Ordem de armazenamento de dados em arranjos

Fortran (por colunas)




```
integer :: idx(2,3) = (/1,1,2,2,3,3/)
```

C/C++ (por linhas)

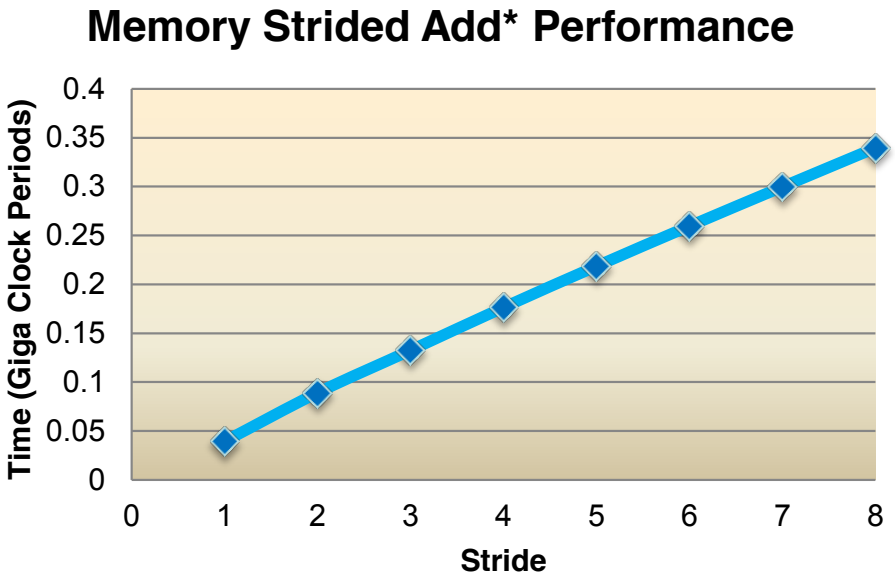



```
integer :: idx(2,3) = (/1,2,3,1,2,3/)
```

$$\mathbf{idx} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}$$

Acesso com passo largo

- Acesso espaçado na memória reduz a largura de banda efetiva da memória.
- Pior que acesso não alinhado. Muitas operações para povoar uma *cache line*, registradores, etc.




```
*do i = 1,4000000*istride, istride
      a(i) = b(i) + c(i) * sfactor
 enddo
```

Blocagem

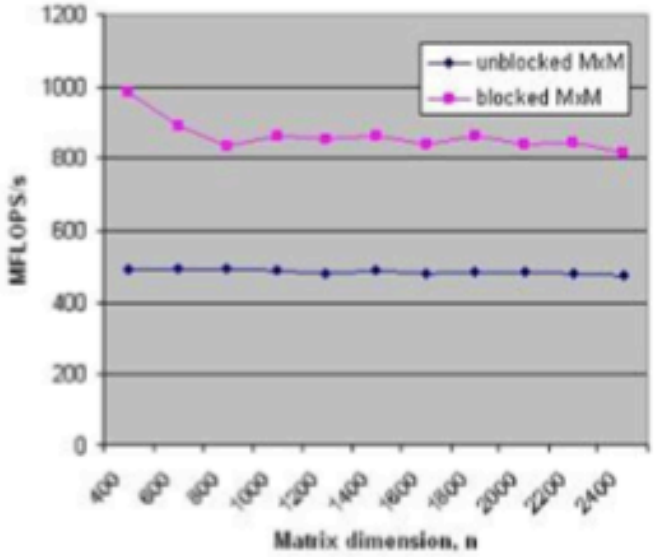
Funciona com pequenos blocos de matrizes quando a expressão contém acessos a memória com espaçamentos mistos.

```
do i=1,n
  do j=1,n
     A(j,i)=B(i,j)
  end do
end do
```

```
do i=1,n,2
  do j=1,n,2
    A(j  ,i  )=B(i  ,j  )
    A(j+1,i  )=B(i+1,j  )
    A(j  ,i+1)=B(i  ,j+1)
    A(j+1,i+1)=B(i+1,j+1)
  end do
end do
```

Blocagem

matrix multiplication example

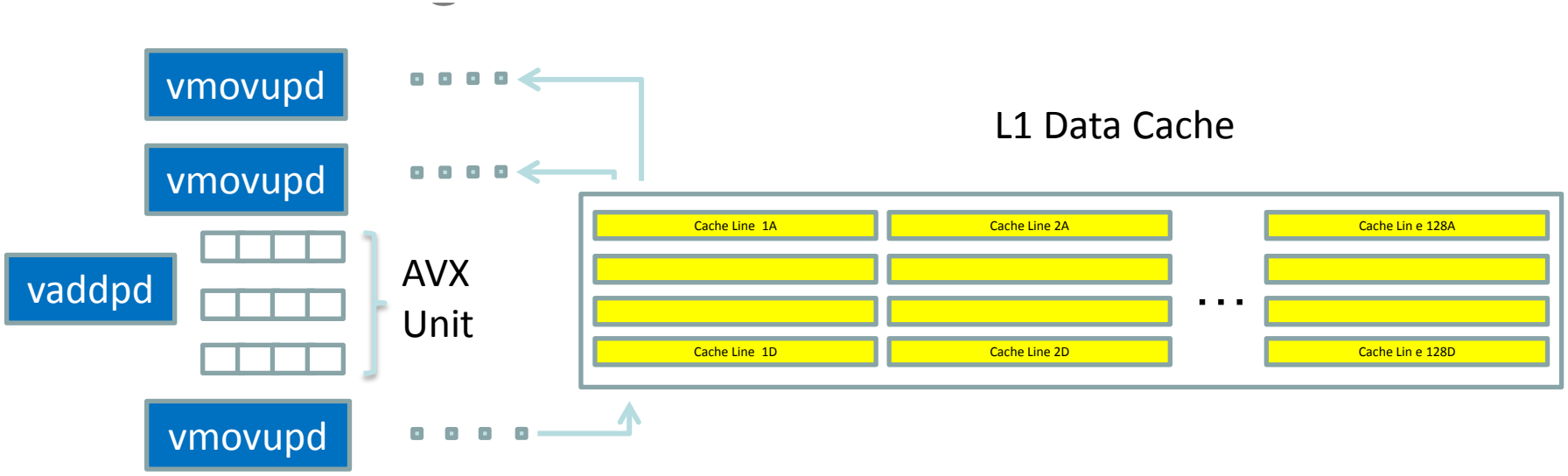



```
real*8 a(n,n), b(n,n), c(n,n)
do ii=1,n,nb
   do jj=1,n,nb
      do kk=1,n,nb
         do i=ii,min(n,ii+nb-1)
            do j=jj,min(n,jj+nb-1)
               do k=kk,min(n,kk+nb-1)

            c(i,j)=c(i,j)+a(j,k)*b(k,i)
```

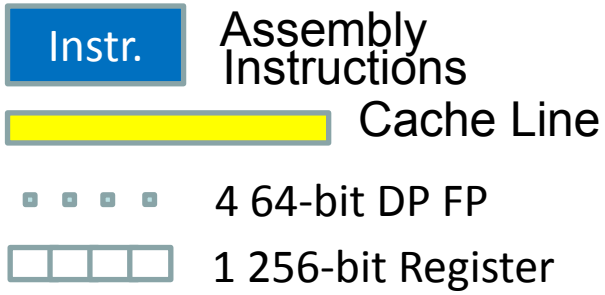

```
end do; end do; end do; end do; end do; end do
```

Much more efficient implementations exist, in HPC scientific libraries (ESSL, MKL, ACML,...).

Alinhamento dos dados



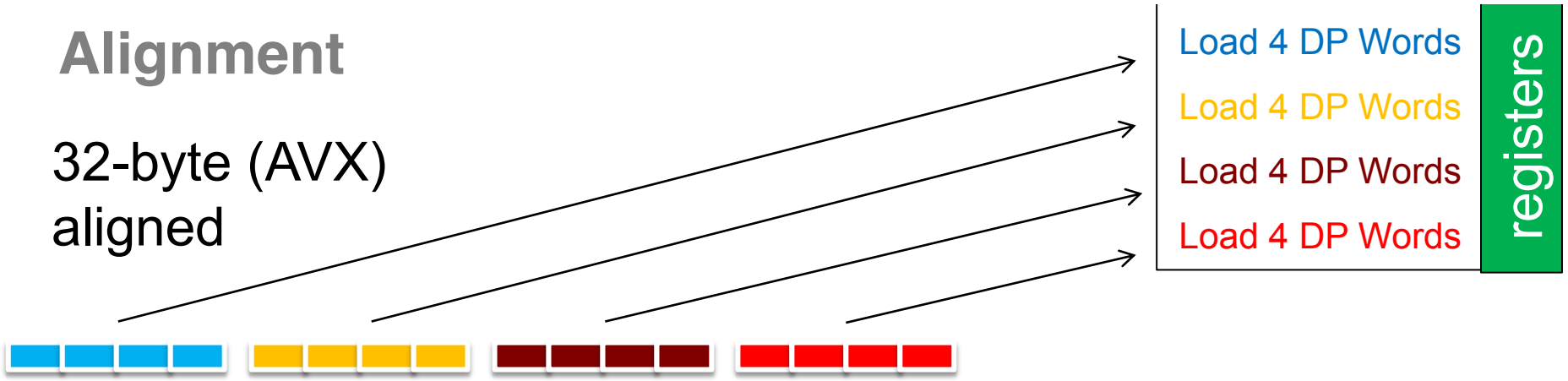
- Instruções de carregamento movem múltiplos valores da cache para os registradores simultaneamente.
- Mais rápido quando a *cache line* inteira é movida com um única unidade, i.e., alinhada



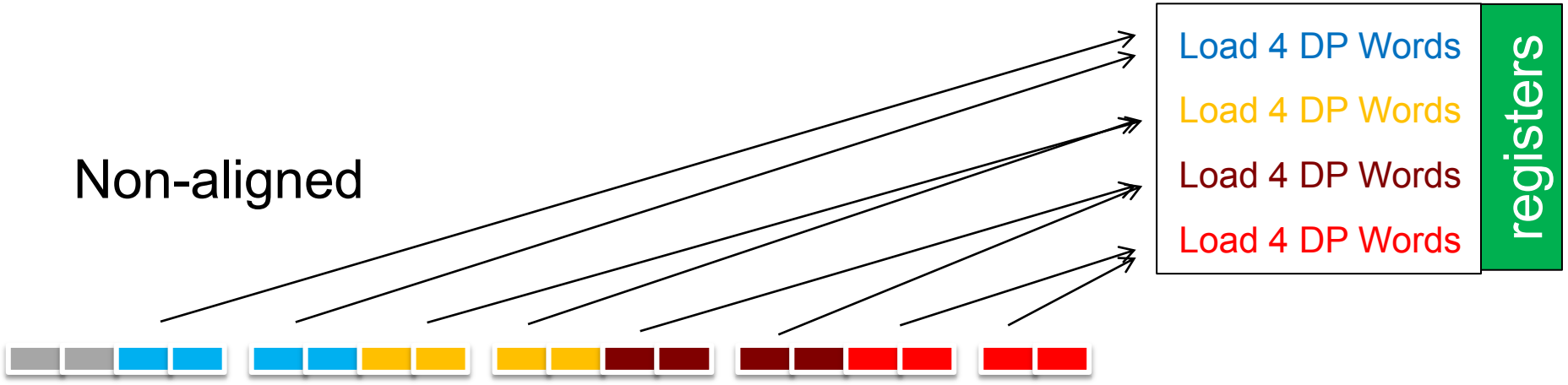
Alinhamento

Alignment

32-byte (AVX) aligned



Non-aligned



Alinhamento

- Aplicado especialmente a arranjos, estruturas
  - Iterar arranjos multidimensionais podem afetar o alinhamento se número de colunas/linhas não forem múltiplos do tamanho da *cache line*.
    - Solução: usar *padding* e adaptar seu algoritmo
- Alinhamento depende da arquitetura do processador:
  - Haswell: 64 bytes
  - MIC: 64 bytes

Alinhamento Manual

- Para Intel, diretivas podem forçar o compilador a assumir alinhamento
  - **`#pragma vector aligned (c/c++)`** afirma que dados em um laço estão alinhados para os limites apropriados.
  - Fortran 90: **`!dir$ vector aligned`**
  - Cuidado – *segfault* se você estiver errado

```
// C/C++
#pragma vector aligned
for(i=0;i<500;i++)
     B[i] = A(i,3)
```

```
! FORTRAN
!dir$ vector aligned
do i=1,500
     B(i) = A(3,i)
end do
```

- Pode forçar alocação dinâmica ser alinhada
  - Com compiladores Intel, use **`_mm_malloc`** ou **`_mm_free`**

Alinhamento

- Definindo alinhamento de arranjos estáticos:
  - Windows C/C++:
    - `__declspec(align(64)) float A[1000];`
  - Linux C/C++:
    - `float A[1000] __attribute((aligned(64)));`
  - Fortran:

```
Real :: A(1000)
!dir$ attributes align: 64::A
```

- Informando o compilador sobre o alinhamento:

```
// C/C++
void myfunc (double p[]){
  __assume_aligned(p, 64);
  for(int i=0; i < n; i++)
     p[i]++;
}
```

```
!Fortran
!dir$ assume_aligned A:64
DO I = 1, N
   A(i) = A(i) + 1
END DO
```

Diagnosticando deficiências da cache e memória

- Maus padrões de acesso pode impedir vetorização
  - No relatório de vetorização: "*vectorization possible but seems inefficient*"
- Caso contrário, pode ser de difícil detecção
  - Indicador: desempenho da vetorização longe do esperado
- Ferramentas de perfilagem podem ajudar:
  - Intel Vtune: Microarchitecture Analysys / Bandwith
  - Visualizar o número de ciclos do CPU gasto com acesso aos dados (L1 cache miss, TLB misses, etc)

Vetorização
Resumo

- Vetorização ocorre em laços e podem ser feitas “automaticamente” pelo compilador
- Precisa saber onde a vetorização pode ocorrer e verificar se o compilador está realmente fazendo.
- Precisa saber se a falha do compilador na vetorização é legítima
  - Se for, corrigir o código, se não, usar #pragma.
- Ter consciência dos efeitos relacionadas a forma como os dados são acessados e o alinhamento deles.
  - As unidades vetoriais precisam ser bem alimentadas!

LAB #2

- Objetivos: explorar as possibilidades da vetorização
  - Dois código-fonte:
    - Multiply: Multiplicação Matricial
      - Código base: multiply0.c
      - Arquivos: handson/lab02/multiply
    - Sten2D: Stencil de diferença finitas 2D
      - Código base: stenc2d_base.c
      - Arquivos: handson/lab02/stend2d
    - Exemplos em Fortran:
      - Handson/lab02/Fortran

LAB02: Multiply

- Step 1: Compile multiply0.c com -03 –qopt-report=3 e veja o relatório gerado.

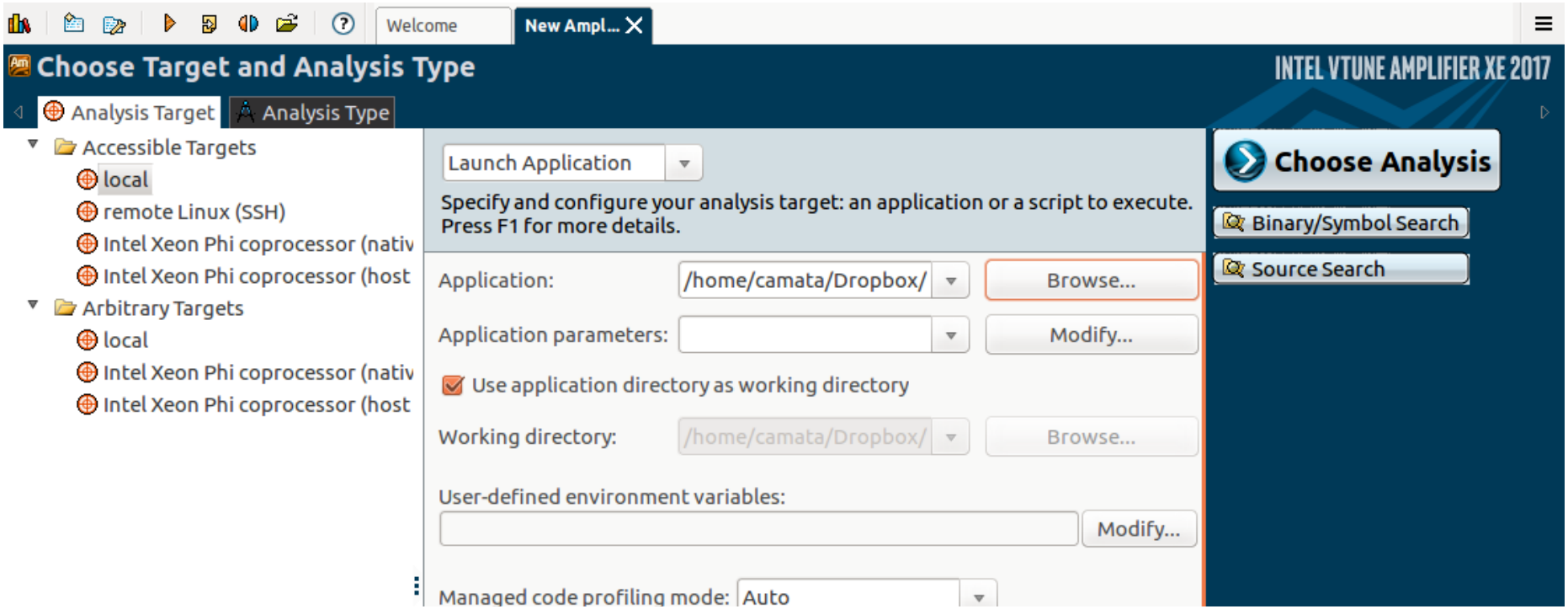
```
LOOP BEGIN at multiply0.c(47,5) inlined into multiply0.c(106,2)
   remark #25444: Loopnest Interchanged: ( 1 2 3 ) --> ( 1 3 2 )
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at multiply0.c(49,10) inlined into multiply0.c(106,2)
      remark #15542: loop was not vectorized: inner loop was already vectorized

      LOOP BEGIN at multiply0.c(48,9) inlined into multiply0.c(106,2)
      <Peeled loop for vectorization>
      LOOP END
    ....
      LOOP BEGIN at multiply0.c(48,9) inlined into multiply0.c(106,2)
      <Remainder loop for vectorization>
         remark #15335: remainder loop was not vectorized: vectorization possible but seems inefficient.
                Use vector always directive or -vec-threshold0 to override
```

Vtune: Coletando dados de desempenho

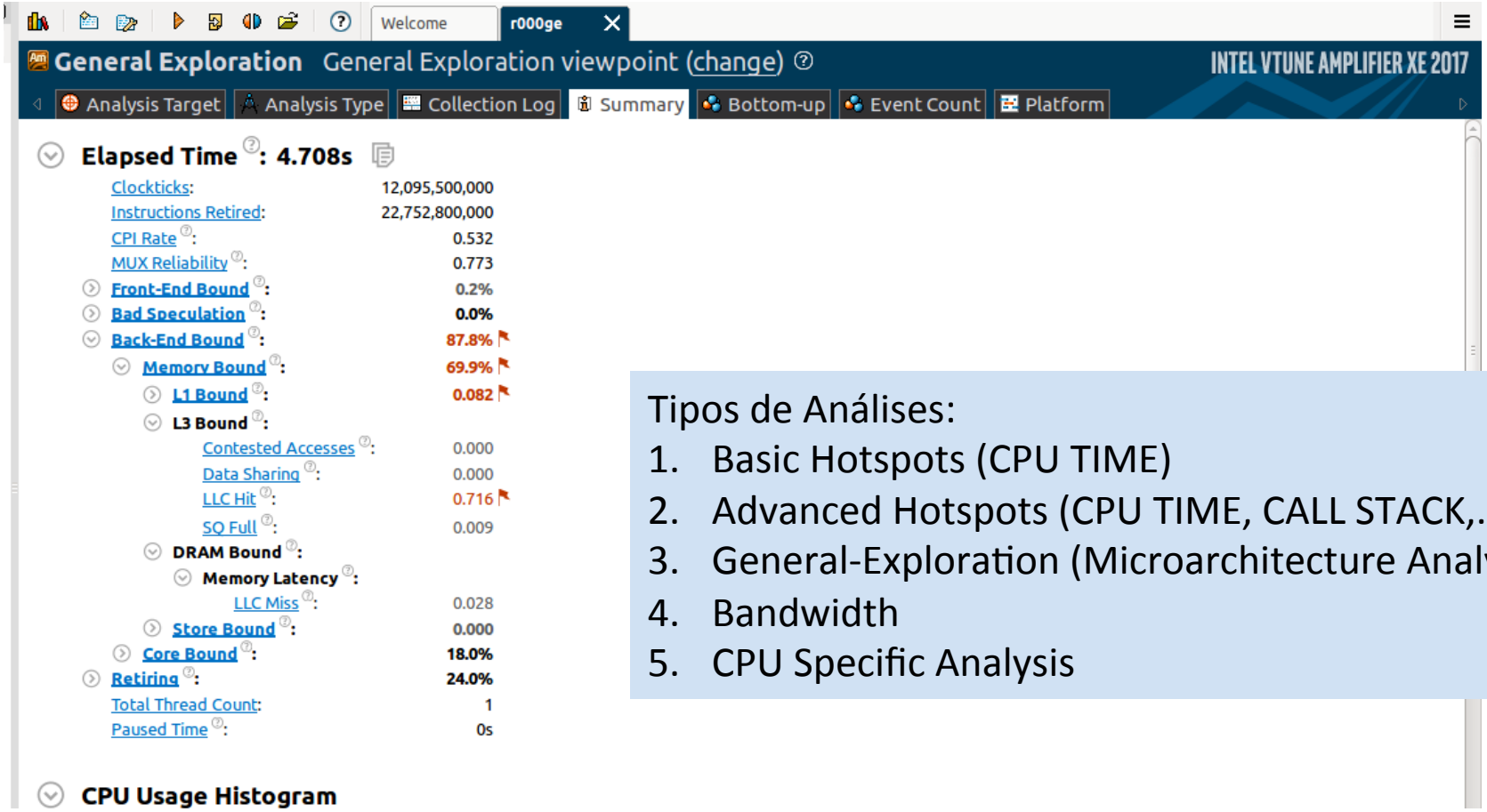
- Processo em três etapas:
  - Compile com a opção –g
  - Execute amplxe-cl para coletar os dados de perfilagem
  - Visualize os dados com amplxe-gui
- O que coletar:
  - `amplxe-cl -help collect`
- Identificando gargalos:
  - `ampxe-cl -collect hotspots ./myexe`
- Visualizando em TXT
  - `amplxe-cl -report summary -r ./r000hs`

Perfilando com o vtune

- Criando o projeto de perfilagem local



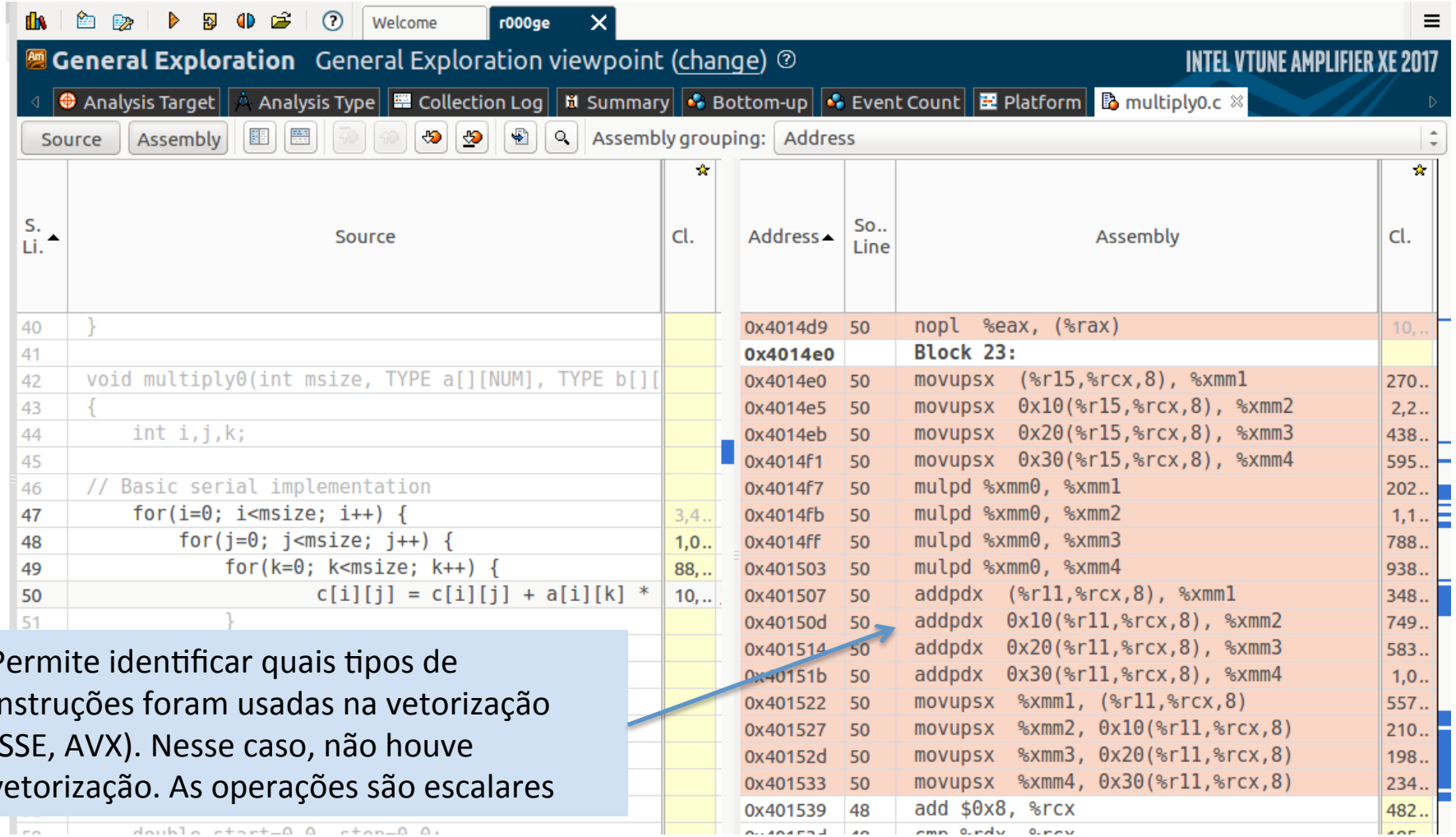
Vtune Amplifier XE 2017



Tipos de Análises:

1. Basic Hotspots (CPU TIME)
2. Advanced Hotspots (CPU TIME, CALL STACK,…)
3. General-Exploration (Microarchitecture Analysis)
4. Bandwidth
5. CPU Specific Analysis

Bottom-UP View



Permite identificar quais tipos de instruções foram usadas na vetorização (SSE, AVX). Nesse caso, não houve vetorização. As operações são escalares

LAB02: Multiply

- Step 2: Permute os loops e compile novamente com -03 –qopt-report=3 e veja o relatório gerado.

```
LOOP BEGIN at multiply0.c(48,9) inlined into multiply0.c(106,2)
    <Remainder loop for vectorization>
      remark #15335: remainder loop was not vectorized: vectorization possible but seems inefficient.
            Use vector always directive or -vec-threshold0 to override
```

- Step 3: Alocação dinâmica usando _mm_malloc() e #pragma ivdep no loop interno

LAB02: Multiply

- Step 4: Implemente blocagem para melhorar eficiência do uso da cache.

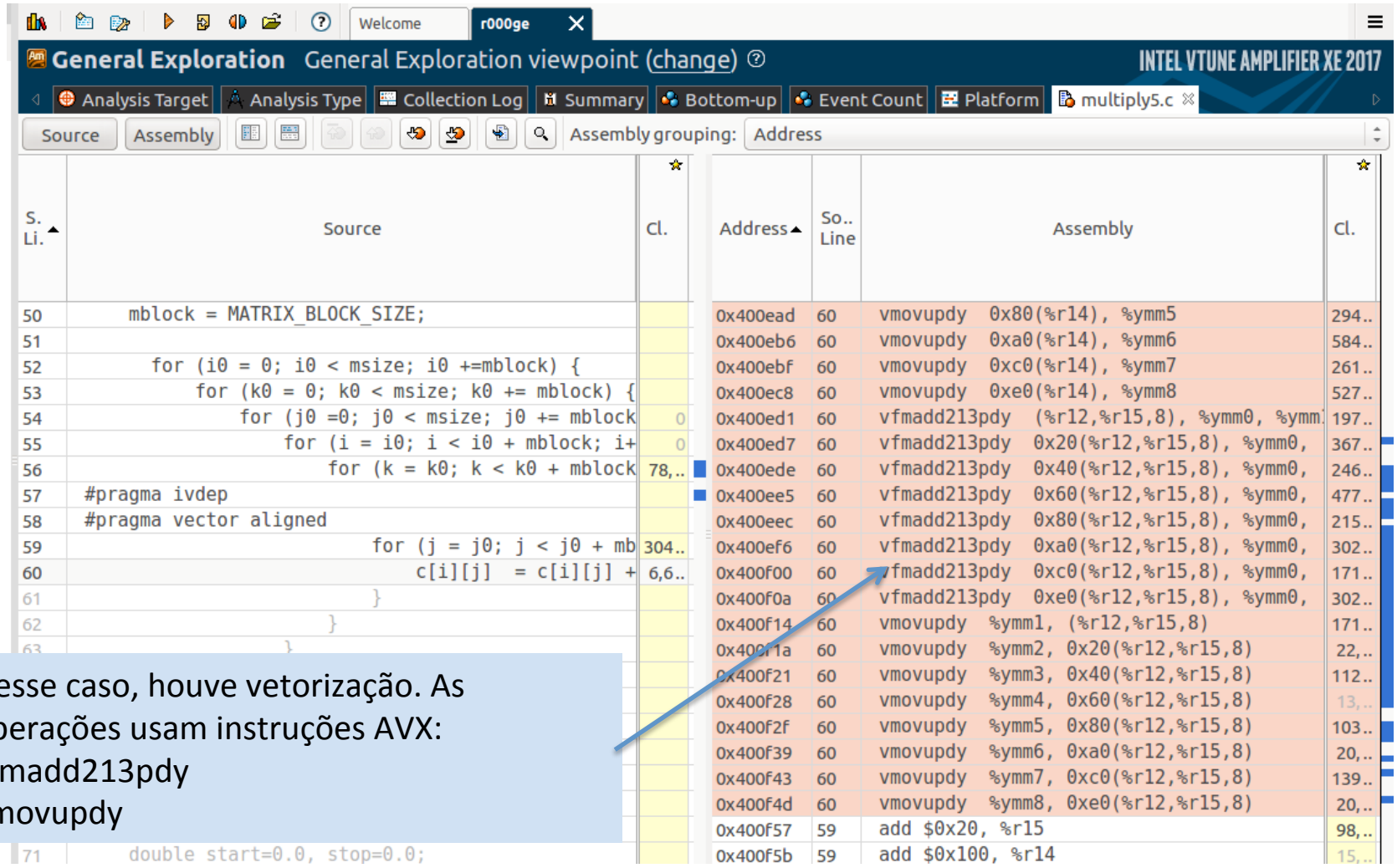
```
mblock = MATRIX_BLOCK_SIZE;

  for (i0 = 0; i0 < msize; i0 +=mblock) {
    for (k0 = 0; k0 < msize; k0 += mblock) {
      for (j0 =0; j0 < msize; j0 += mblock) {
        for (i = i0; i < i0 + mblock; i++) {
          for (k = k0; k < k0 + mblock; k++) {
#pragma ivdep
#pragma vector aligned
            for (j = j0; j < j0 + mblock; j++) {
              c[i][j] = c[i][j] + a[i][k] * b[k][j];
                                  }
                              }
                          }
                      }
                  }
              }
}
```

LAB02: Multiply

- Step 4: Implemente blocagem para melhorar eficiência do uso da cache.

```
LOOP BEGIN at multiply5.c(52,7)
  remark #15542: loop was not vectorized: inner loop was already vectorized
  LOOP BEGIN at multiply5.c(53,11)
    remark #15542: loop was not vectorized: inner loop was already vectorized
    LOOP BEGIN at multiply5.c(54,15)
      remark #15542: loop was not vectorized: inner loop was already vectorized
      LOOP BEGIN at multiply5.c(55,19)
        remark #15542: loop was not vectorized: inner loop was already vectorized
        LOOP BEGIN at multiply5.c(56,23)
          remark #15542: loop was not vectorized: inner loop was already vectorized
          LOOP BEGIN at multiply5.c(59,27)
            remark #15300: LOOP WAS VECTORIZED
          LOOP END
        LOOP END
      LOOP END
    LOOP END
  LOOP END
LOOP END
```

Vtune Bottom-up



Nesse caso, houve vetorização. As operações usam instruções AVX:
vfmadd213pdy
vmovupdy

LAB02: Sten2D

- Compilar com –O2 –qopt-report
- Observar o relatório de otimização.

```
LOOP BEGIN at sten2d9pt_base.c(53,3)
   remark #25096: Loop Interchange not done due to: Imperfect Loop Nest
   remark #25452: Original Order found to be proper, but by a close margin
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15346: vector dependence: assumed OUTPUT dependence between fout[c] (68:9) and fout[c] (68:9)
   remark #15346: vector dependence: assumed OUTPUT dependence between fout[c] (68:9) and fout[c] (68:9)
   LOOP BEGIN at sten2d9pt_base.c(54,5)
      remark #15344: loop was not vectorized: vector dependence prevents vectorization
      remark #15346: vector dependence: assumed FLOW dependence between fout[c] (68:9) and fin[c] (68:9)
      remark #15346: vector dependence: assumed ANTI dependence between fin[c] (68:9) and fout[c] (68:9)
      LOOP BEGIN at sten2d9pt_base.c(67,7)
         remark #15344: loop was not vectorized: vector dependence prevents vectorization
         remark #15346: vector dependence: assumed FLOW dependence between fout[c] (68:9) and fin[c] (68:9)
         remark #15346: vector dependence: assumed ANTI dependence between fin[c] (68:9) and fout[c] (68:9)
```

LAB02: Sten2D

- Verificar se há dependências.
- Usar `#pragma ivdep` no loop mais interno
- Recompilar e observar o relatório de otimização.

```
LOOP BEGIN at sten2d9pt_vect.c(53,3)
   remark #25096: Loop Interchange not done due to: Imperfect Loop Nest
remark #25452: Original Order found to be proper, but by a close margin
   remark #15542: loop was not vectorized: inner loop was already vectorized
LOOP BEGIN at sten2d9pt_vect.c(54,5)
      remark #15542: loop was not vectorized: inner loop was already vectorized
      LOOP BEGIN at sten2d9pt_vect.c(67,7)
      <Peeled loop for vectorization>
         remark #25015: Estimate of max trip count of loop=3
      LOOP END
      LOOP BEGIN at sten2d9pt_vect.c(67,7)
         remark #15388: vectorization support: reference fout[c] has aligned access   [ sten2d9pt_vect.c(68,9) ]
         remark #15389: vectorization support: reference fin[c] has unaligned access   [ sten2d9pt_vect.c(76,26) ]
         remark #15389: vectorization support: reference fin[n] has unaligned access   [ sten2d9pt_vect.c(74,26) ]
```

LAB02: Sten2D

- Alinhar os dados: `__mm_malloc/_mm_free`
- Usar `#pragma vector aligned` no loop mais interno
- Recompilar e observar o relatório de otimização.

```
OOP BEGIN at sten2d9pt_pad.c(83,7)
      remark #15388: vectorization support: reference fout[c] has aligned access   [ sten2d9pt_pad.c(84,9) ]
      remark #15388: vectorization support: reference fin[c] has aligned access   [ sten2d9pt_pad.c(92,26) ]
      remark #15388: vectorization support: reference fin[n] has aligned access   [ sten2d9pt_pad.c(90,26) ]
      remark #15388: vectorization support: reference fin[s] has aligned access   [ sten2d9pt_pad.c(91,26) ]
      remark #15388: vectorization support: reference fin[e] has aligned access   [ sten2d9pt_pad.c(89,26) ]
      remark #15388: vectorization support: reference fin[w] has aligned access   [ sten2d9pt_pad.c(88,26) ]
      remark #15388: vectorization support: reference fin[nw] has aligned access   [ sten2d9pt_pad.c(84,26) ]
      remark #15388: vectorization support: reference fin[ne] has aligned access   [ sten2d9pt_pad.c(85,26) ]
      remark #15388: vectorization support: reference fin[sw] has aligned access   [ sten2d9pt_pad.c(86,26) ]
      remark #15388: vectorization support: reference fin[se] has aligned access   [ sten2d9pt_pad.c(87,26) ]
```

IDENTIFICANDO OPORTUNIDADES DE DE VETORIZAÇÃO E PARALELISMO (MULTI-THREADING)

Identificando oportunidades de vetorização e paralelismo

Intel® Advisor fornece duas ferramentas que ajudam garantir que aplicações C/C ++ e Fortran alcancem o potencial pleno de desempenho em processadores modernos, como processadores Intel® Xeon Phi ™

Vectorization Advisor é uma ferramenta de otimização que permite identificar loops que mais beneficiariam da vetorização, identificar o que está bloqueando a vetorização, explorar o benefício da alternativas de reorganizações de dados, e verificar a corretudo do código.

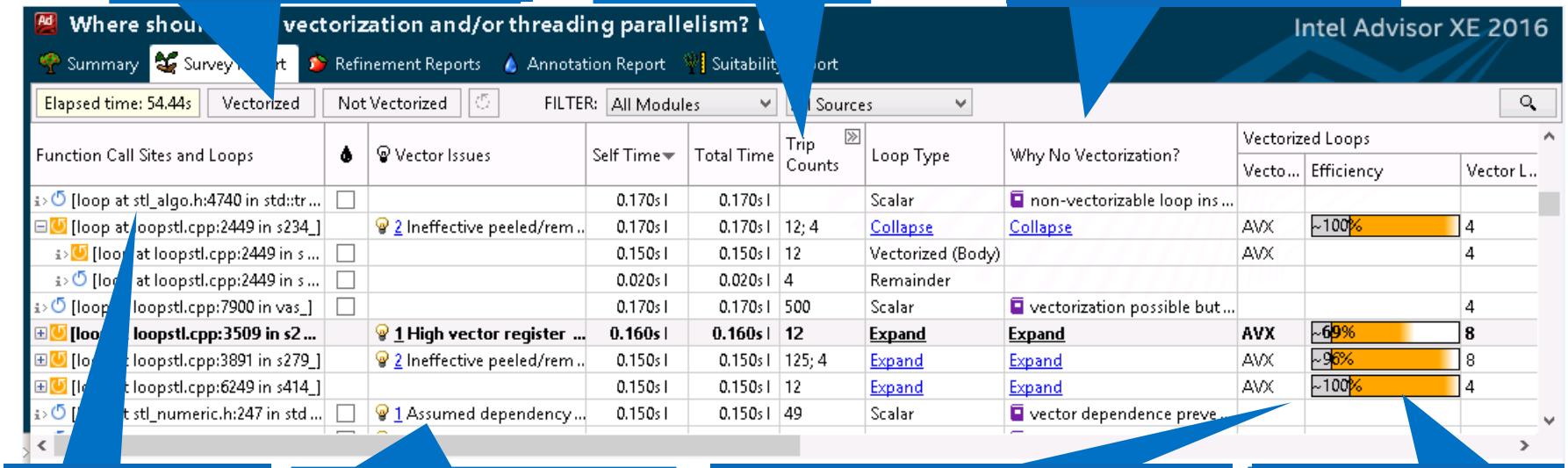
Threading Advisor é uma ferramenta de prototipagem Multi-threading que permite analisar, projetar, ajustar e verificar opções de paralelismo sem interromper o seu desenvolvimento normal.

Vectorization Advisor

Filtra quais loops foram vetorizados

Trip Counts

O que prejudica a vetorização?



Foco nos laços que demadam mais CPU

Quais problemas de vetorização?

Quais instruções vetoriais estão sendo usadas?

Como eficiente é o código?

Buscando Vetorização Eficiente

1. Diagnóstico do compilador + Dados de Desempenho + Informações SIMD

Function Call Sites and Loops	Self Time	Total Time	Compiler Vectorization: Loop Type	Compiler Vectorization: Why No Vectorization?
[loop in runCForallLambdaLoops]	0.094s	0.094s	Scalar	vector dependence prevents vector ...
[loop in runCForallLambdaLoops]	0.140s	3.744s	Scalar	inner loop was already vectorized
[loop in std::_Complex_base<double,struct _C_double_complex>::i...	0.031s	0.031s	Vectorized (Body)	
Vectorized SSE; SSE2 loop processing Float32; Float64 data type(s) having Divisions; Square Roots operations Peeled loop; loop stmts were reordered				
[loop in std::basic_string<char,struct std::char_traits<char>,class std::allo...	0.000s	544.0...	Scalar	nonstandard loop is not a vectoriza...
[loop in std::basic_string<char,struct std::char_traits<char>,class std::allo...	0.000s	544.0...	Scalar	nonstandard loop is not a vectoriza...
[loop in std::num_put<char,class std::ostreambuf_iterator<char,struct st...	0.000s	0.234s	Scalar	nonstandard loop is not a vectoriza...

2. Detecta problemas e fornece recomendações de como resolve-los.

2 8

Issue: Peeled/Remainder loop(s) present

All or some source loop iterations are not executing in the kernel loop. Improve performance by moving source loop iterations from peeled/remainder loops to the kernel loop. Read more at Vector Essentials, Utilizing Full Vectors...

Recommendation: Align memory access

Projected maximum performance gain: High
Projection confidence: Medium

The compiler created a peeled loop because one of the memory accesses in the source loop does not start at a data boundary. Align the memory access and tell the compiler your memory access is aligned. This example aligns memory using a 32-byte boundary:

```
float *array;
array = (float *)_mm_malloc(ARRAY_SIZE*sizeof(float), 32);

// Somewhere else
__assume_aligned(array, 32);
// Use array in loop
```

3. Análise de Dependência

Problems and Messages

ID	Type	Site Name	Sources	Modules	State
P1	Parallel site information	site2	dqtest2.cpp	dqtest2	✔ Not a problem
P2	Read after write dependency	site2	dqtest2.cpp	dqtest2	New
P3	Read after write dependency	site2	dqtest2.cpp	dqtest2	New
P4	Write after write dependency	site2	dqtest2.cpp	dqtest2	New
P5	Write after write dependency	site2	dqtest2.cpp	dqtest2	New
P6	Write after read dependency	site2	dqtest2.cpp	dqtest2	New
P7	Write after read dependency	site2	dqtest2.cpp; idle.h	dqtest2	New

4. Análise do padrão de acesso a memória

Site Name	Site Function	Site Info	Loop-Carried Dependencies	Strides Distribution	Access Pattern
loop_site_203	runCRawLoops	runCRawLoops.cxx:1063	RAW:1	No information available	No information available
loop_site_139	runCRawLoops	runCRawLoops.cxx:622	No information available	39% / 36% / 25%	Mixed strides
loop_site_160	runCRawLoops	runCRawLoops.cxx:925	No information available	100% / 0% / 0%	All unit strides

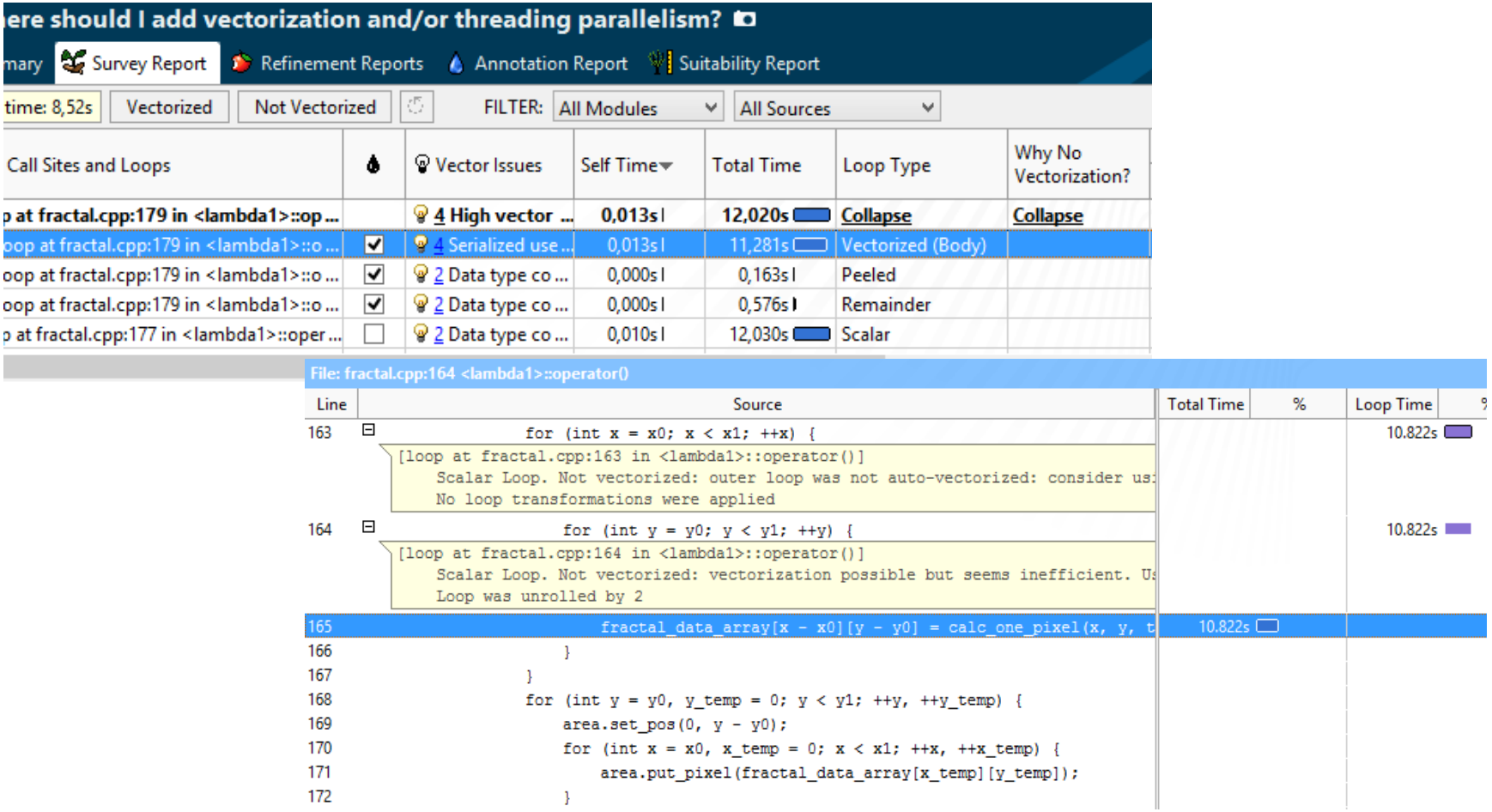
Memory Access Patterns | Correctness Report

ID	Stride	Type	Source	Modules	Alignment
P22	0; 0; 1	Unit stride	runCRawLoops.cxx:637	lcals.exe	
P23	0; 0	Unit stride	runCRawLoops.cxx:638	lcals.exe	
P30	-1575; -63; -26; -25; -1; 0; 1; 25; 26; 63; 2164801	Variable stride	runCRawLoops.cxx:628	lcals.exe	

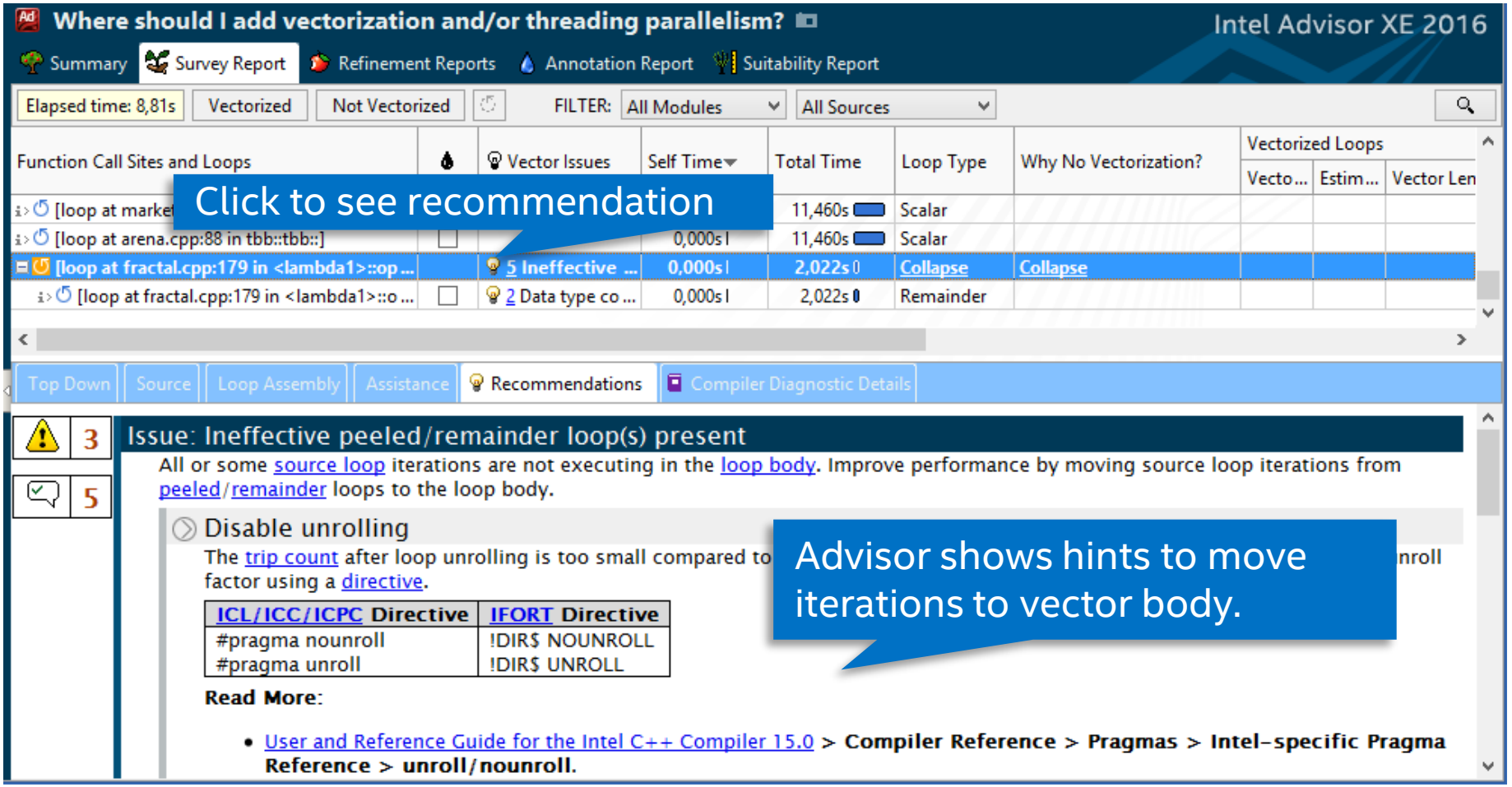
```
635        j2 = ( j2 & 64-1 ) ;
636        p[ip][0] += y[i2+32];
637        p[ip][1] += z[j2+32];
638        i2 += e[i2+32];
639        j2 += f[j2+32];
```

```
626        i1 &= 64-1;
627        j1 &= 64-1;
628        p[ip][2] += b[j1][i1];
```

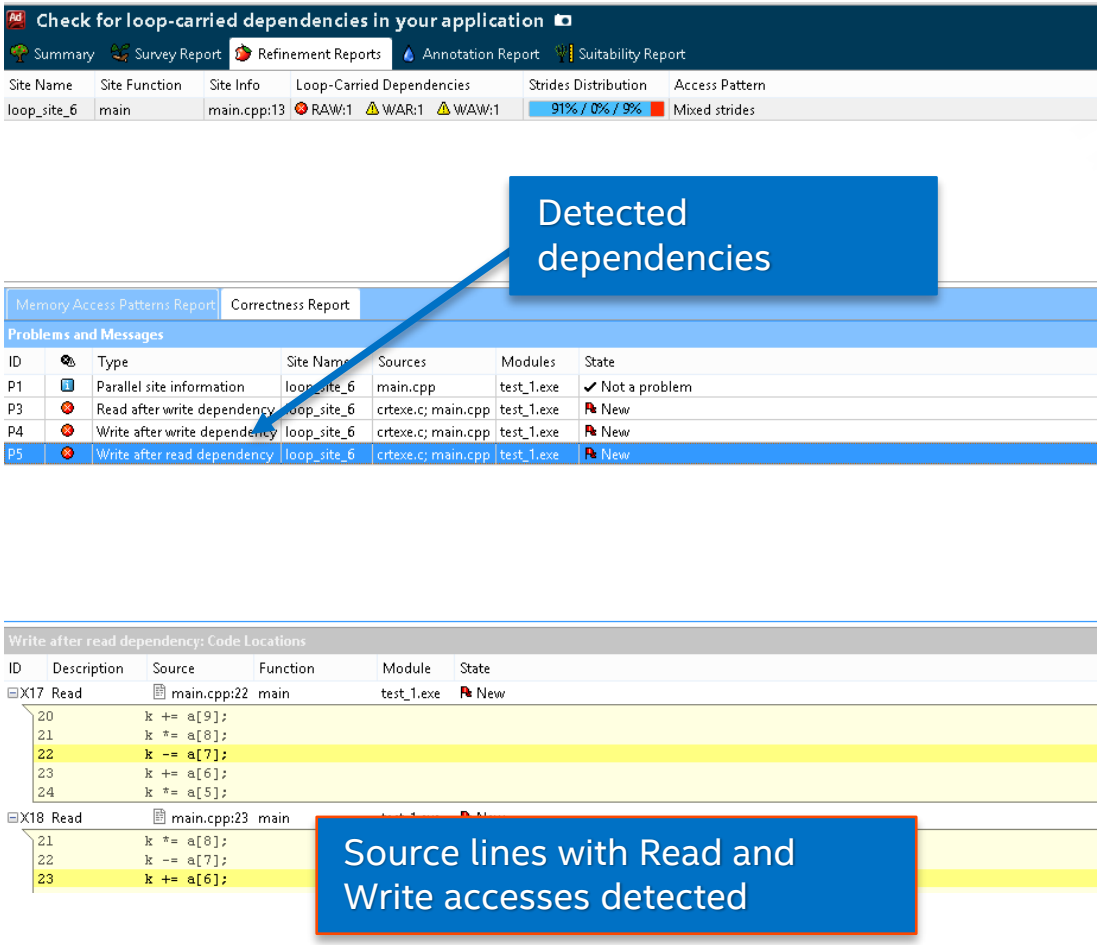
P1: Onde a vetorização será mais benéfica.



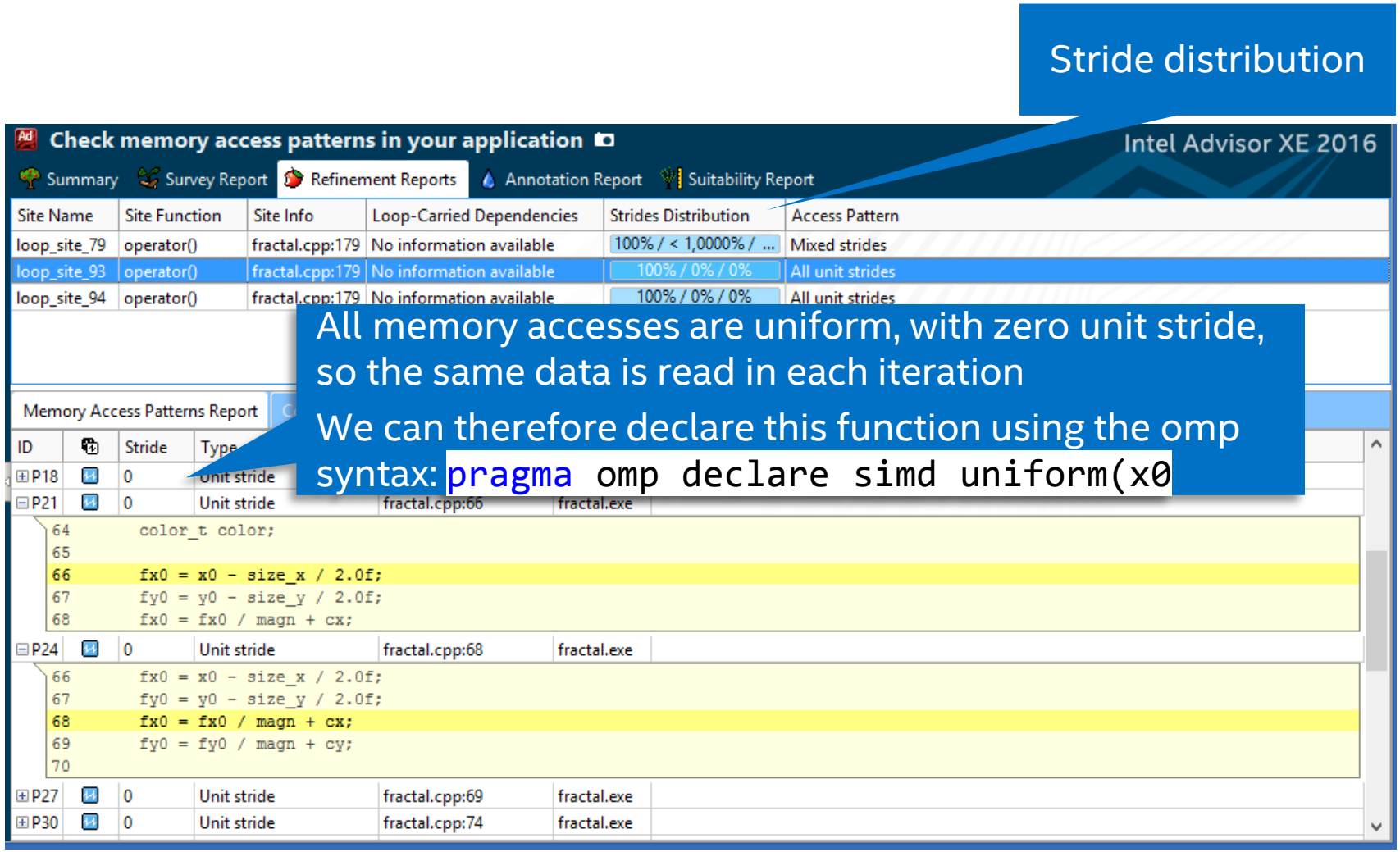
P2: Detecta problemas e fornece recomendações



P3: Verifica dependência de dados



P4: Analise do padrão de acesso a memória



LAB 03: Vectorization Advisor

- Objetivos:
  - Identificar loops que mais serão beneficiados pela vetorização.
  - Identificar o que está bloqueando vetorização.
  - Aumentar a confiança de que a vetorização é segura.
  - Explorar o benefício da reorganizações de dados.

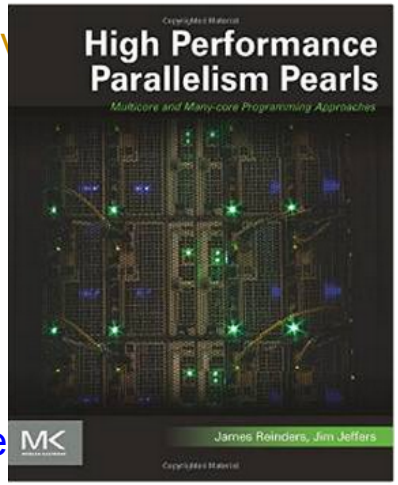
- Arquivos:
  - handson/lab03/pricing.tar.gz
  - Copie para sua area de trabalho e segue as instruções dada.

- Tutorial detalhando disponível em:
  - https://software.intel.com/en-us/intel-advisor-tutorial-vectorization-linux-cplusplus

LAB 03

Código de exemplo - Black-Scholes Pricing Code

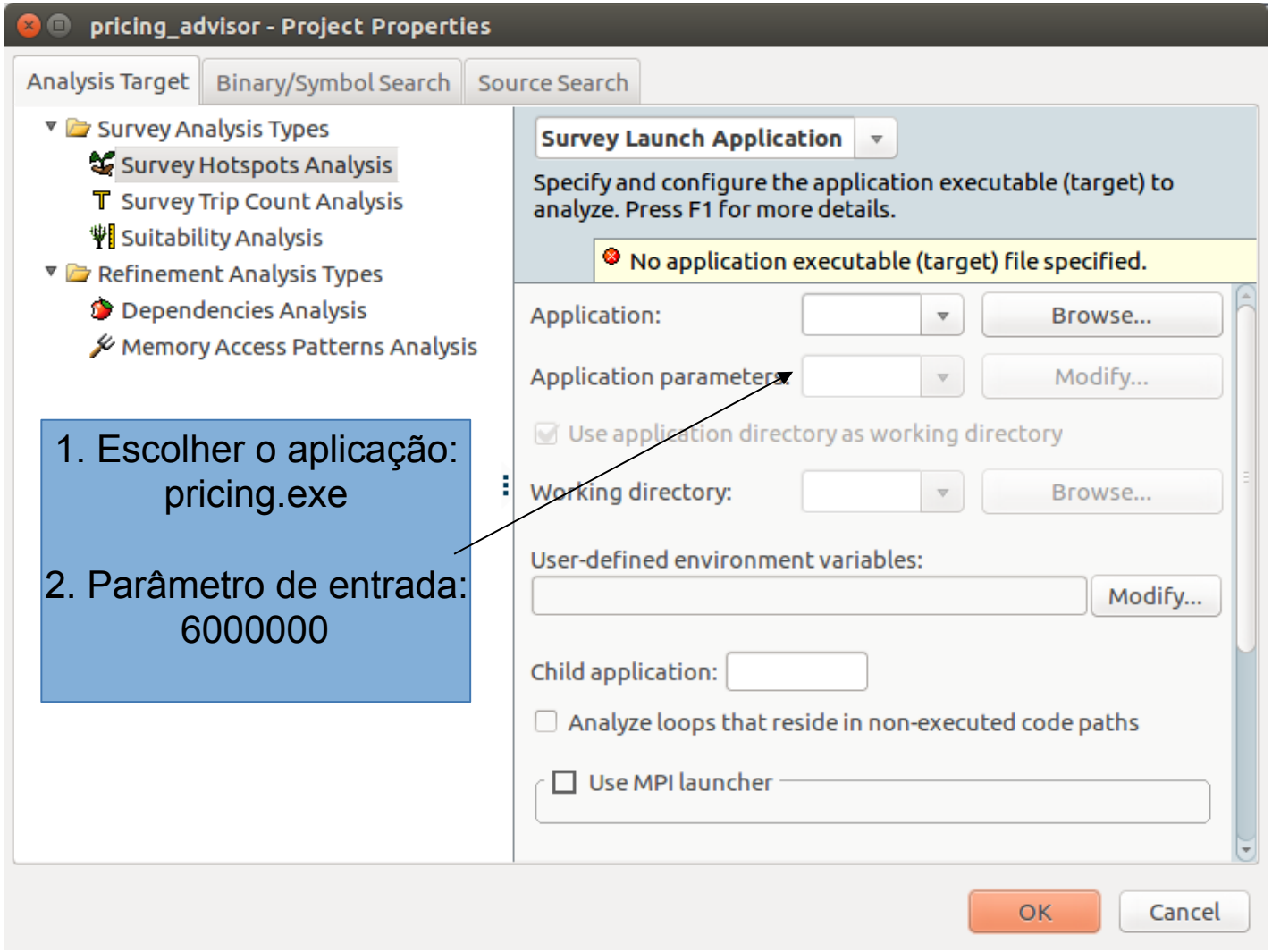
"a mathematical model of a financial market containing certain derivative inv
instruments. "



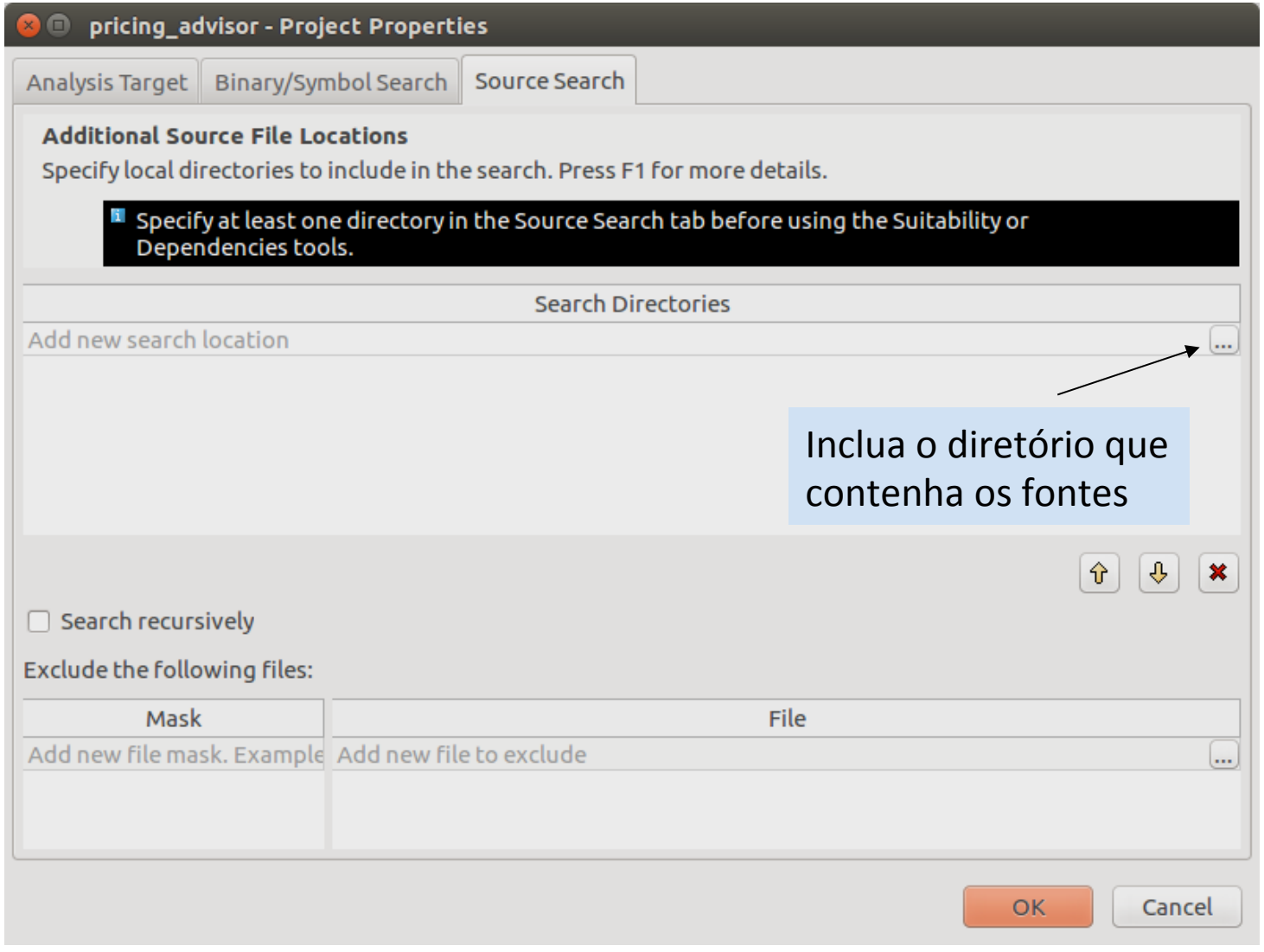
Exemplo retirado do livro "High Performance Parallelism Pearls"
Código fonte: http://lotsofcores.com/pearls.code

Artigo sobre otimização deste método
https://software.intel.com/en-us/articles/case-study-computing-black-schole
advanced-vector-extensions

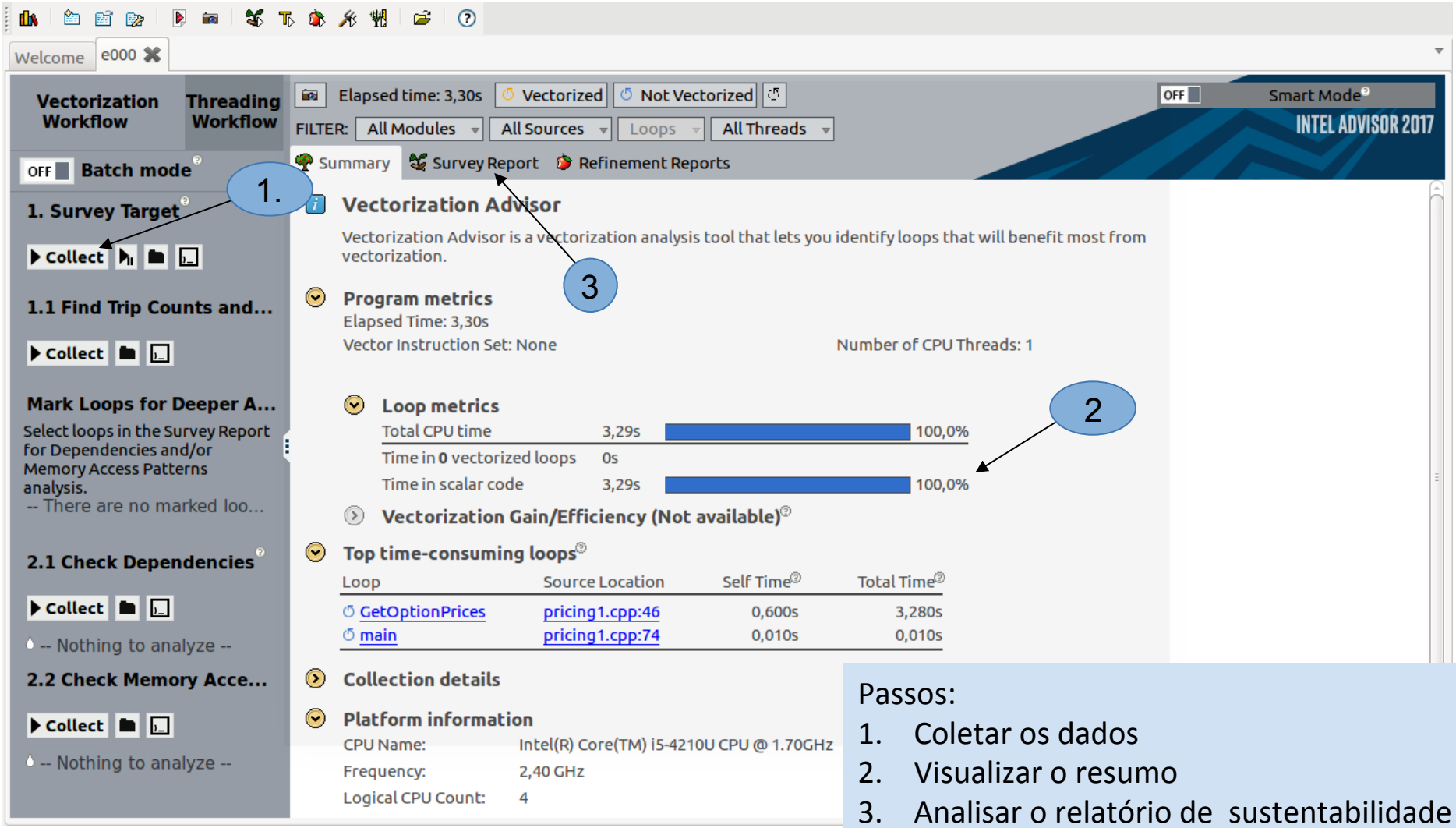
Criando Projeto



Indicar o diretório do código-fonte



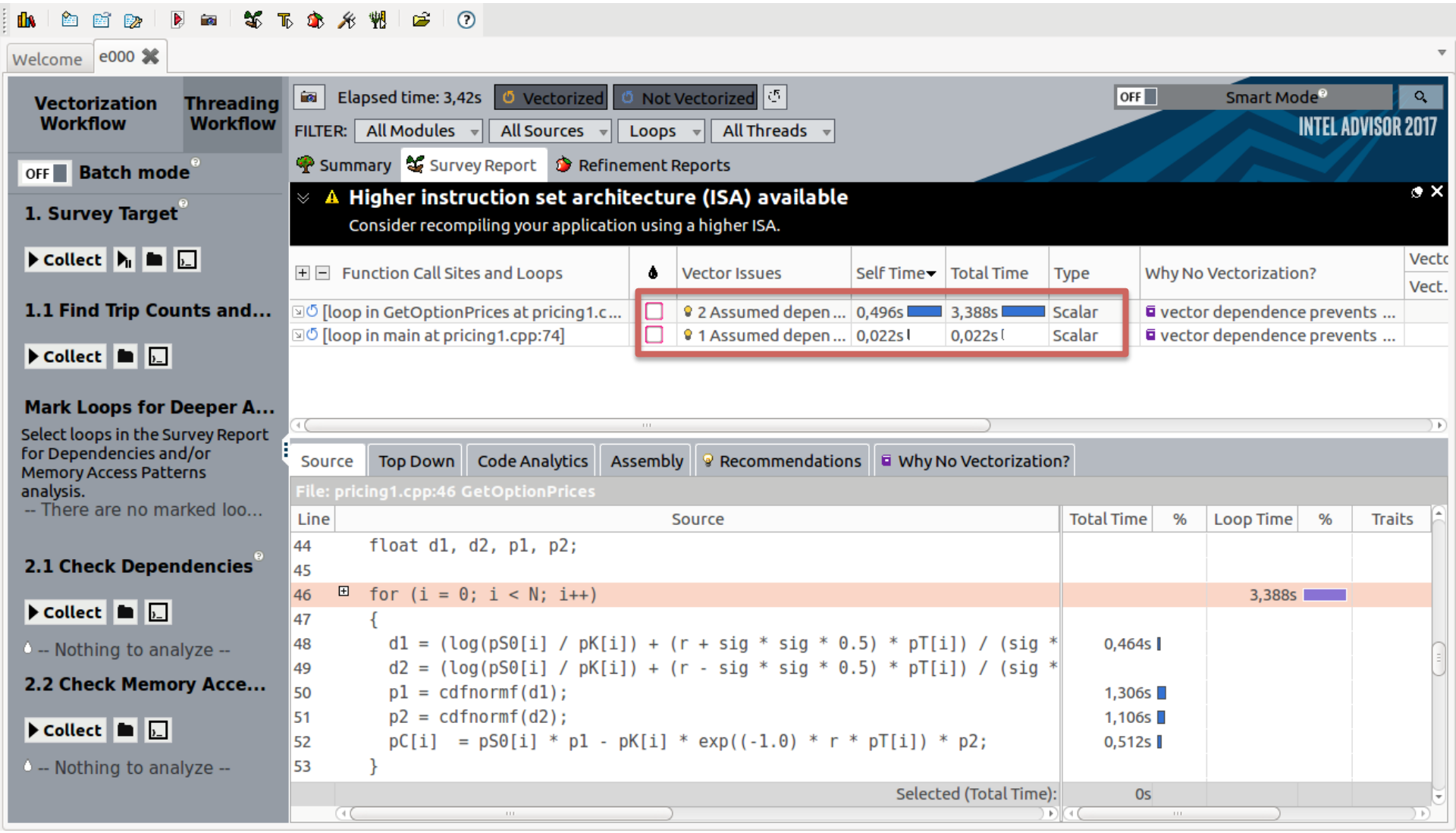
Survey Analysis



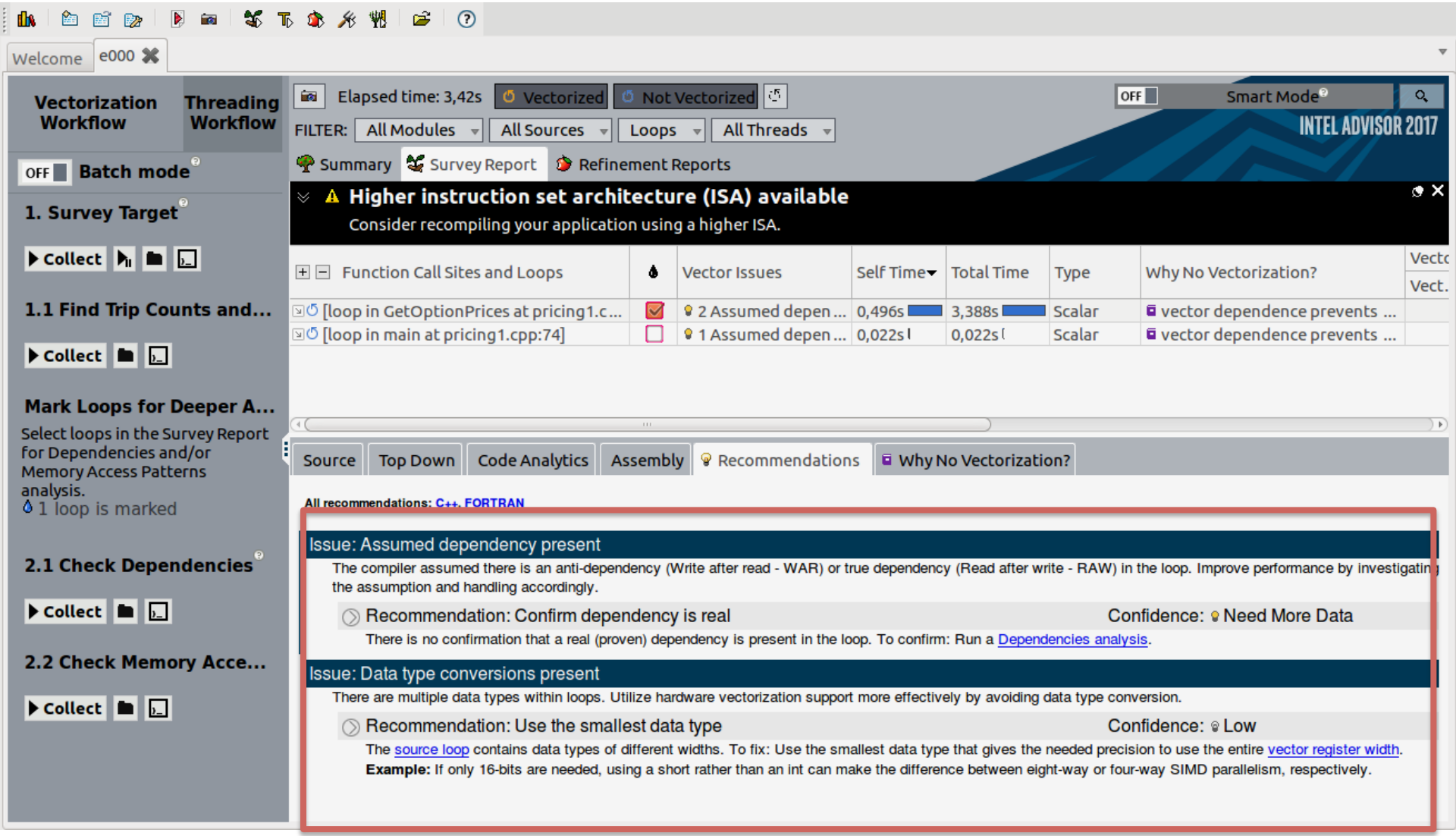
Passos:

1. Coletar os dados
2. Visualizar o resumo
3. Analisar o relatório de sustentabilidade

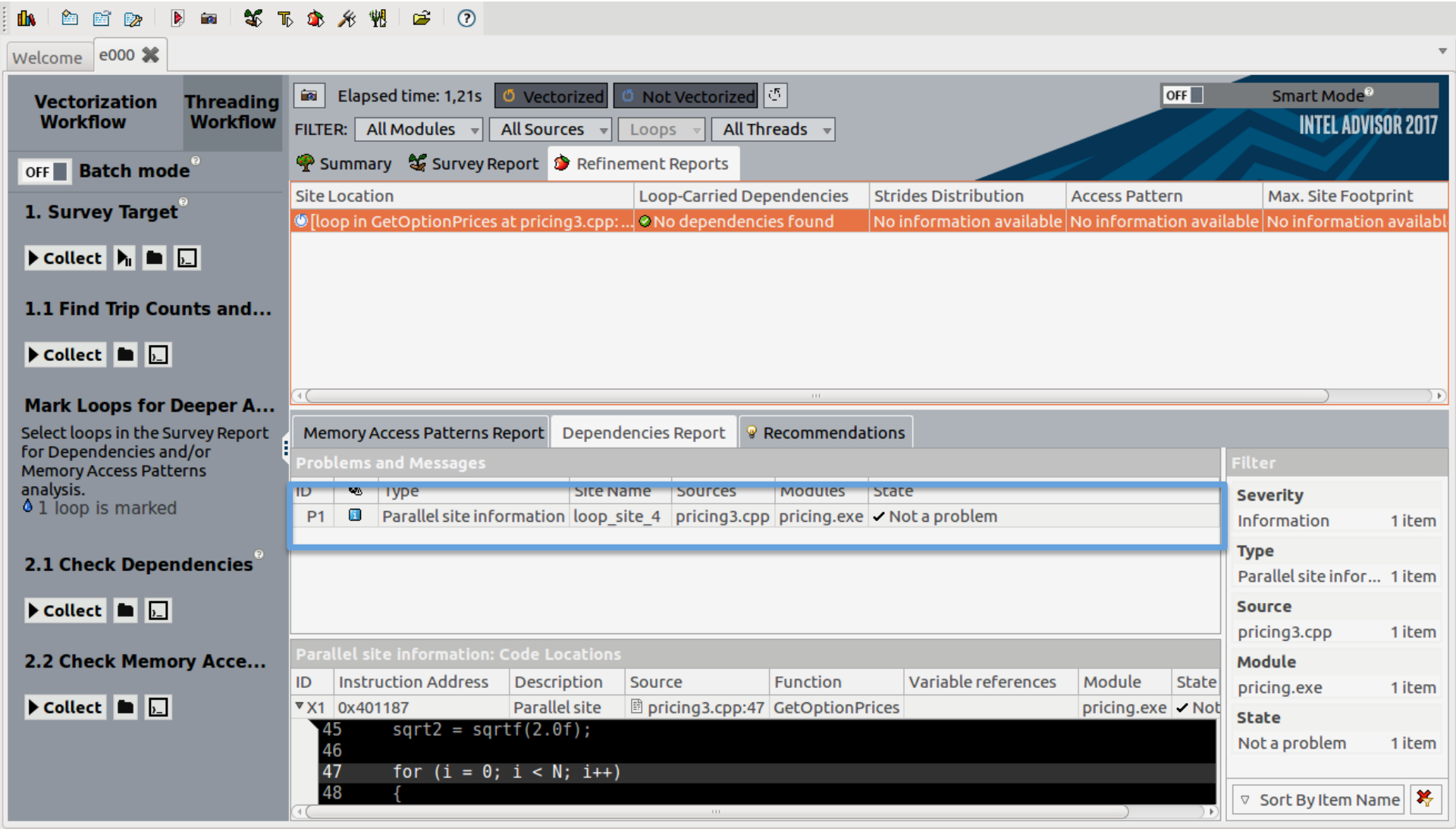
Survey Report



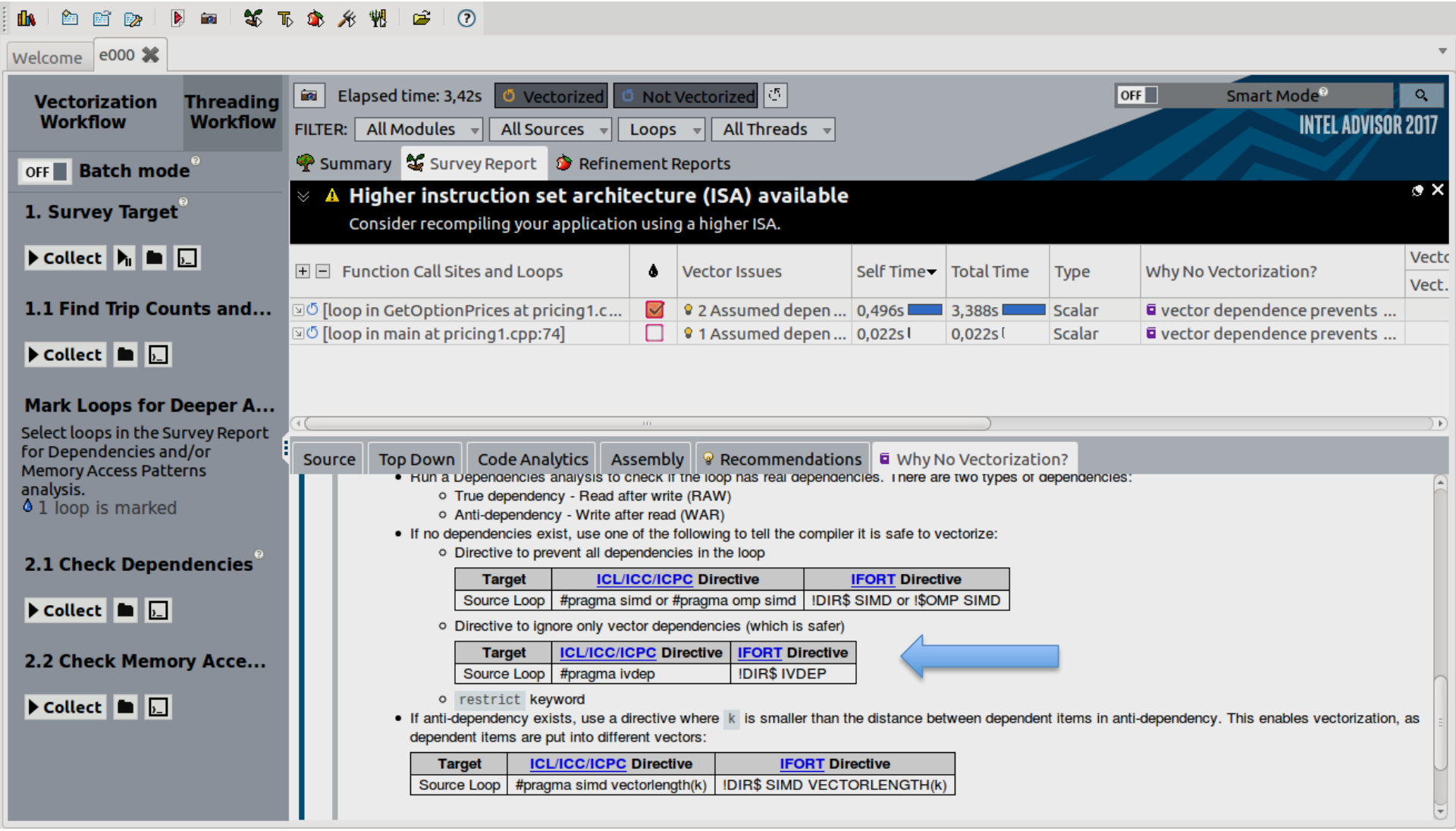
Recomendações



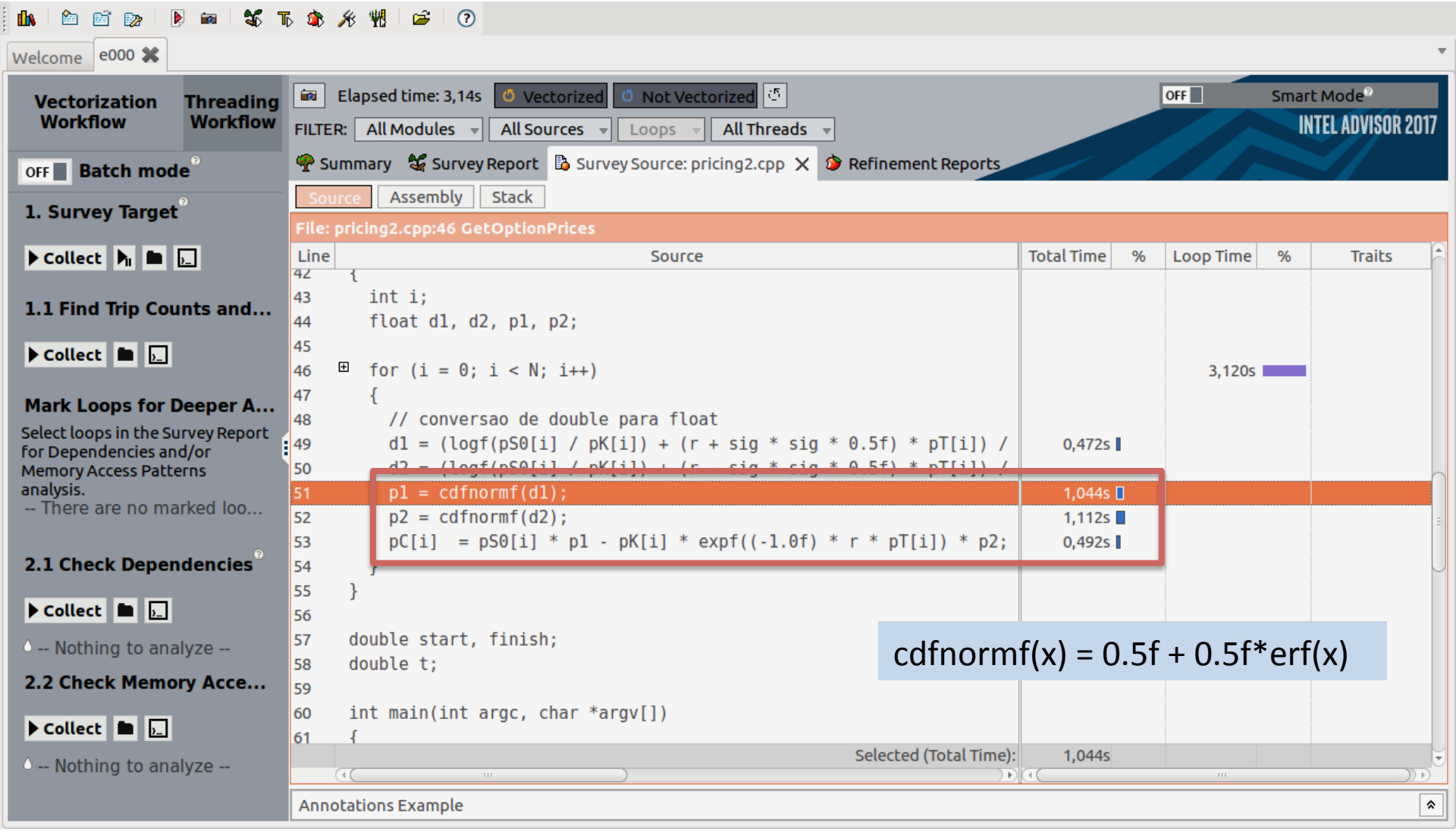
Checar dependências



Implementar as recomendações

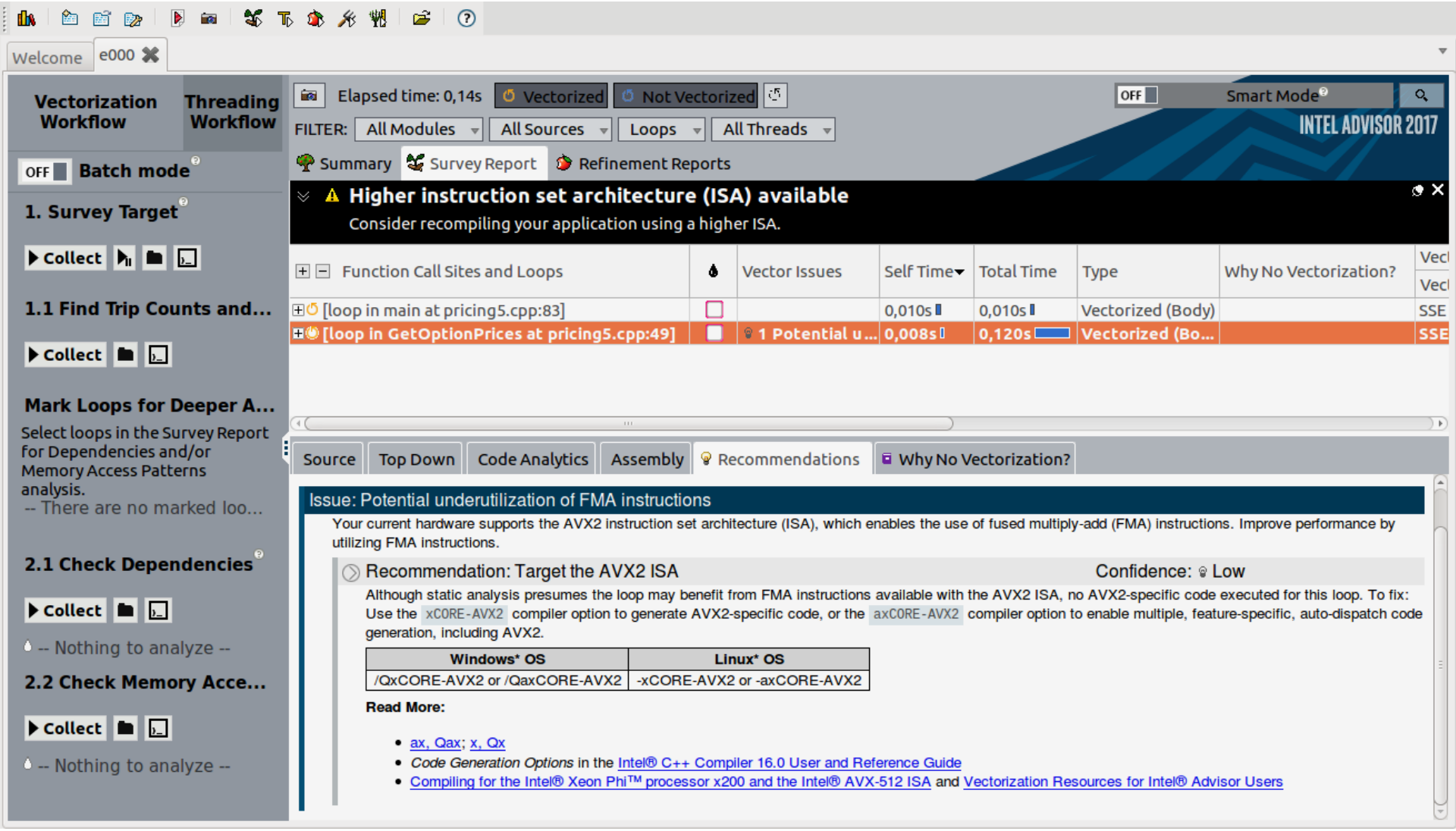


Rodar Survey Analysis

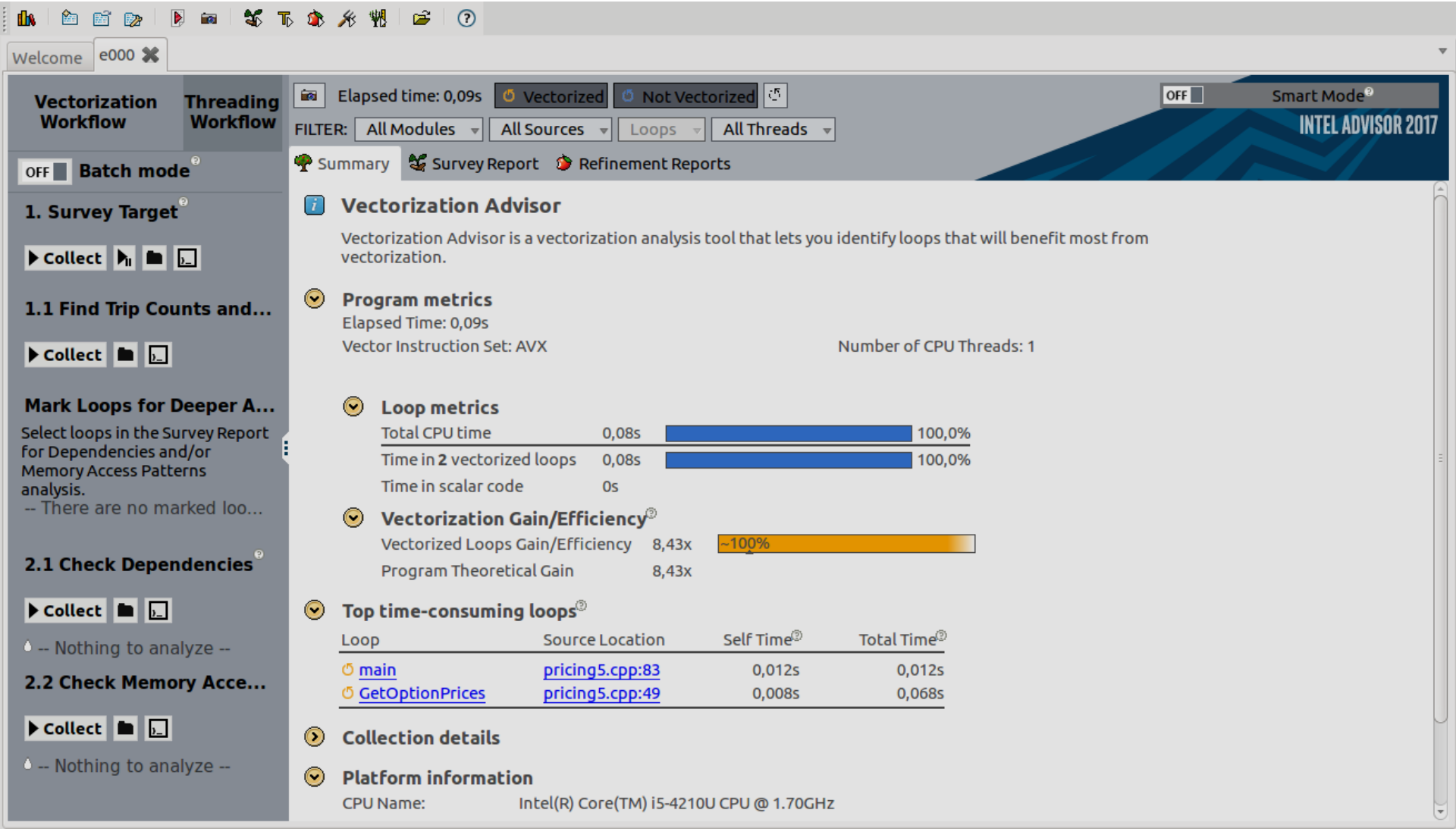


cdfnormf(x) = 0.5f + 0.5f*erf(x)

Uso de Instruções Vetoriais AVX



Relatorio Final



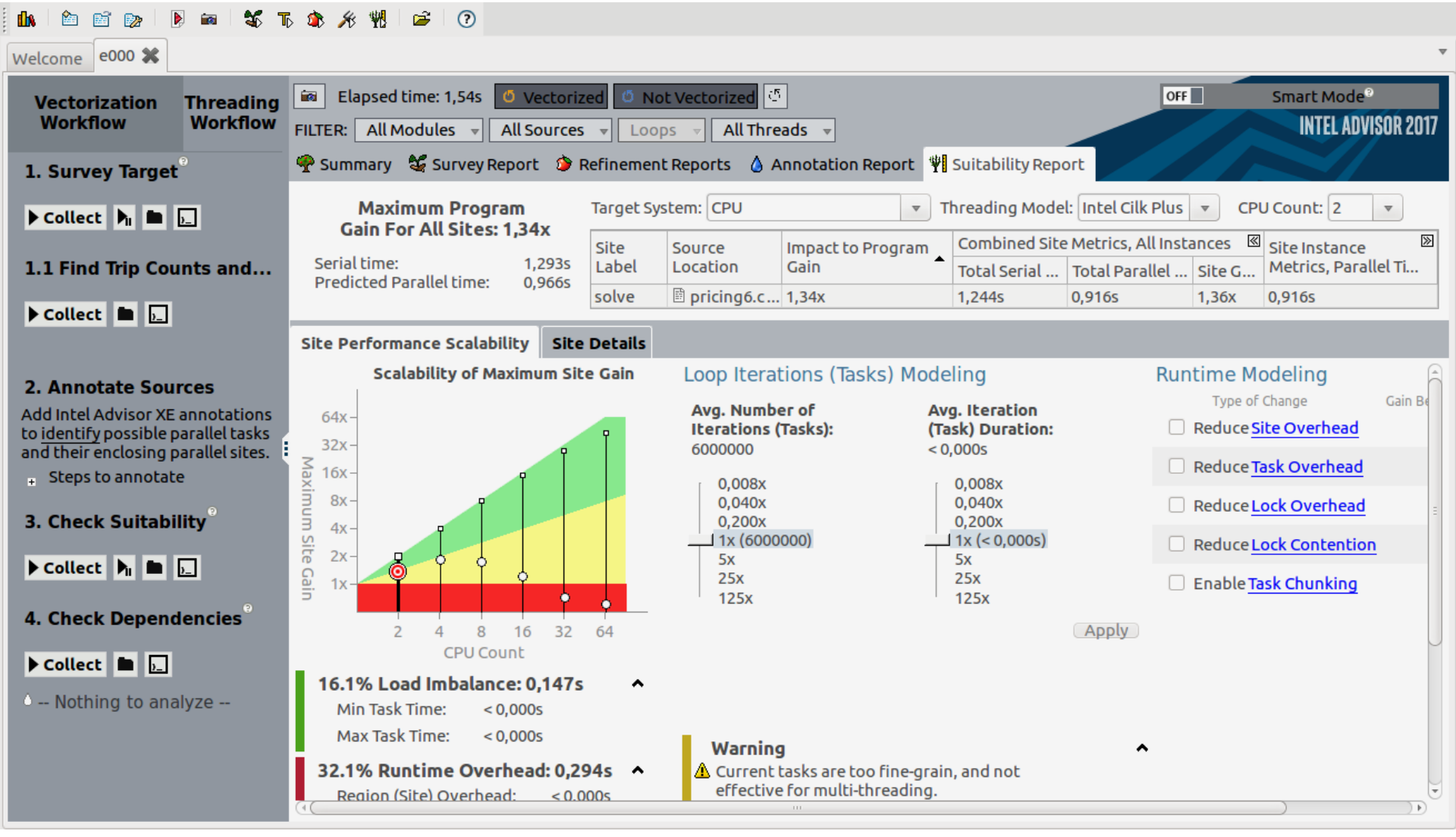
Identificando Oportunidades de Paralelismo

Inserindo as “anotações” do Advisor para executar a próxima fase: Check Suitability

```
__declspec(noinline) void GetOptionPrices(float *pT, float *pK, float *pS0, float *pC)
{
  int i;
  float d1, d2, p1, p2;

//#pragma cilk grainsize=256
//#pragma omp parallel for
  #pragma ivdep
  #pragma unroll(4)
  ANNOTATE_SITE_BEGIN( MySite1 );  // Place before the loop control statement to begin a parallel code region (parallel site).
  for (i = 0; i < N; i++)
  {
    ANNOTATE_ITERATION_TASK( MyTask1 );  // Place at the start of loop body. This annotation identifies an entire body as a task.
    /* - Float usage */
    d1 = (logf(pS0[i] / pK[i]) + (r + sig * sig * 0.5f) * pT[i]) / (sig * sqrtf(pT[i]));
    d2 = (logf(pS0[i] / pK[i]) + (r - sig * sig * 0.5f) * pT[i]) / (sig * sqrtf(pT[i]));
    p1 = cdfnormf(d1);
    p2 = cdfnormf(d2);
    pC[i]  = pS0[i] * p1 - pK[i] * exp((-1.0f) * r * pT[i]) * p2;

  }
  ANNOTATE_SITE_END();  // End the parallel code region, after task execution completes


}
```

Linux - Compilando / Link com Advisor
icpc -O2 -openmp 02_pricing.cpp
-o pricing.exe
-I/opt/intel/advisor_xe/include/
-L/opt/intel/advisor_xe/lib64/

Identificando Oportunidades de Paralelismo



LAB 03: Threading Advisor

- Objetivos:
  - Identificar oportunidades de paralelismo multithread
- Arquivos
  - handson/lab03/nqueens_advisor
  - Copie para sua area de trabalho e segue as instruções dada.
- Tutorial detalhando está disponível em:
  - https://software.intel.com/en-us/intel-advisor-tutorial-vectorization-linux-cplusplus

Mais Informações

- Modernização de códigos:
  - https://software.intel.com/en-us/modern-code
- Books:
  - Intel Xeon Phi Processor High Performance programming: Knights Landing Edition
  - High Performance Parallelism Pears
  - lotsofcores.com