# ANÁLISE DE DESEMPENHO DE COMPILADORES BASEADA EM OTIMIZAÇÕES DE COMPILAÇÃO

# COMPILER PERFORMANCE ANALYSIS BASED ON COMPILING OPTIMIZATIONS

Geovane Sant'Ana da Silva[*]
Gabriel Teixeira Galam[**]
Maurício Acconcia Dias[***]

## RESUMO

Atualmente o desenvolvimento de software encontra-se em um patamar avançado com diversas opções de paradigmas. Os compiladores têm apresentado os mais diferentes tipos de otimização, porém esta área é pouco explorada pelos desenvolvedores que na maioria dos casos tenta otimizar o código diretamente esquecendo da qualidade da solução alcançada pelas otimizações de compilação. Considerando este cenário este trabalho de pesquisa aborda o tema comparando dois compiladores amplamente utilizados, o Gnu Compiler Collection (GCC) e o compilador do Microsoft Visual Studio 2015. As otimizações foram aplicadas a três algoritmos considerados benchmarks no mesmo ambiente de hardware e sistema operacional. Os resultados mostram que o compilador do Visual Studio apresentou melhores tempos de execução em todas as comparações realizadas.

**Palavras-chave**: Otimização. Profiling. Compilador. Comparação. GCC.

## ABSTRACT

Software design is at an advanced level with several paradigms. Compilers have presented the most different types of optimization; however this area is little explored by the developers who in most cases tries to optimize the code directly forgetting the quality of the solution achieved by the compiler optimizations. Considering this scenario, this research deals with the topic by comparing two widely used compilers, the Gnu Compiler Collection (GCC) and the Microsoft Visual Studio 2015 compiler. The optimizations were applied to three algorithms considered benchmarks using the same hardware and operating system. The results show that the Visual Studio compiler presented better execution times for all analyzed cases.

**Keywords**: Optimization. Profile. Compiler. Analysis GCC.

[*] Graduação em Ciência da Computação. Faculdade de Tecnologia, Ciências e Educação (FATECE). geovanesantanasilva@gmail.com
[**] Graduação em Ciência da Computação. Faculdade de Tecnologia, Ciências e Educação (FATECE). gabriel.tex.galam@gmail.com
[***] Universidade de São Paulo (USP) – Laboratório de Robótica Móvel (LRM). Faculdade de Tecnologia, Ciências e Educação (FATECE). macdias@icmc.usp.br

## Introdução

O desenvolvimento de software atualmente está em um estágio de evolução ainda maior em comparação com anos recentes devido ao crescente número de núcleos em processadores e a necessidade de se paralelizar os algoritmos para um melhor desempenho. Outro cenário que ocorre com frequência é a necessidade de se criar códigos otimizados para sistemas embarcados que necessitam de execução em tempo-real e baixo consumo de energia.

O nível exigido de otimização de código é alto e muitas vezes os programadores não são capazes de atingir o nível desejado apenas com otimizações no código-fonte devido à complexidade. Considerando este cenário, os desenvolvedores de compiladores estão melhorando as ferramentas de otimização de código com objetivo de atingir os requisitos dos sistemas.

Apesar das melhorias apresentadas, programadores não são totalmente familiarizados com as possibilidades de otimização de código disponibilizadas pelos compiladores. Este não é um assunto tratado com cuidado nos cursos de computação e a formação dos profissionais é deficiente neste sentido. Visando apresentar de forma mais consistente os benefícios da otimização de código disponibilizada pelos compiladores, este trabalho de pesquisa apresenta um comparativo de compilação de *benchmarks* compilados por dois compiladores amplamente utilizados o *Gnu Compiler Collection* (GCC) e o compilador do *Microsoft Visual Studio 2015*. Os dois compiladores possuem otimizações de compilação compatíveis e foram executados no mesmo sistema para garantir uma igualdade de condições. Os resultados demonstram uma superioridade do código gerado pelo compilador do *Microsoft Visual Studio 2015*.

## 1 *Benchmarks*

Segundo Gray (1993), nenhuma métrica sozinha é capaz de medir o desempenho de um sistema e suas aplicações. Cada sistema é desenvolvido para trabalhar em determinados domínios e é incapaz de realizar outras tarefas. Para medir o quão bom é um sistema em determinada tarefa, segundo determinada métrica, é necessário que se execute uma tarefa padrão a fim de permitir a comparação entre sistemas.

Um *benchmark* nada mais é que um software determinado que irá exigir uma determinada quantidade de trabalho de um sistema. Por exemplo, quando se quer determinar a capacidade de uma placa de vídeo utiliza-se um *benchmark* com o 3DMark[1] cujo resultado pode ser comparado com uma lista considerável de placas testadas anteriormente.

Para ser útil, um *benchmark* para domínio específico deve ser (GRAY, 1993):

- Relevante: o desempenho máximo da aplicação deve ser mensurado e também a relação preço/desempenho para operações típicas de determinado sistema.
- Portável: deve ser fácil executar o *benchmark* em diversas plataformas, arquiteturas e sistemas.
- Escalável: o *benchmark* deve ser aplicável a pequenos e grandes sistemas computacionais, tanto de um núcleo quanto de vários núcleos.
- Simples: o resultado deve ser fácil de entender, caso contrário não terá credibilidade.

A avaliação também deve ser quantitativa e não qualitativa, pois quando um sistema é avaliado é necessário saber exatamente os resultados para cada métrica a fim de se ter um dado preciso e comparável.

Para este trabalho foram escolhidos 3 *benchmarks* que pertencem a um jogo de avaliação de linguagens de programação[2]. A linguagem escolhida para as implementações foi a linguagem C e o sistema operacional *Windows* devido ao fato de ser possível utilizar o compilador do *Visual Studio* apenas neste sistema.

O primeiro programa calcula o conjunto de *Mandelbrot*[3] no intervalo [$-1.5-i, 0.5+i$] em um bitmap *NxN*. A saída é impressa *byte* por *byte* em um arquivo de formato *portable bitmap*.

O segundo algoritmo escolhido gera sequências de DNA segundo uma dada sequência inicial. As sequências de DNA são geradas por uma seleção aleatória ponderada de 2 alfabetos da seguinte maneira: inicialmente a probabilidade esperada de selecionar um nucleotídeo é convertida em probabilidades cumulativas, em seguia um número aleatório é

[1] http://www.3dmark.com/
[2] http://benchmarksgame.alioth.debian.org/
[3] http://mathworld.wolfram.com/MandelbrotSet.html

combinado contra as probabilidades para selecionar um nucleotídeo utilizando busca binária ou linear. O site recomenda que seja utilizado um gerador de números aleatórios baseado em congruência linear. Este algoritmo no site é denominado *FASTA.*

O terceiro algoritmo calcula o complemento reverso de uma cadeia de DNA. O software deve utilizar um arquivo que seja resultado do algoritmo *FASTA* definido anteriormente. Para cada sequência de DNA no arquivo de entrada devem ser escritas, em um arquivo de saída, a identificação, a descrição e a sequência reversa no formato da saída do algoritmo *FASTA.*

Utilizando estes três *benchmarks* segundo todas as especificações contidas no site de origem, os experimentos foram realizados e os resultados foram coletados e analisados.

## 2 Método e Ferramentas

O objetivo principal deste trabalho é comparar as otimizações de código que podem ser realizadas por dois compiladores diferentes ao executar *3 benchmarks* e comparar seus resultados. Este trabalho será realizado da seguinte maneira (Figura 1): incialmente o ambiente computacional será configurado para que os programas possam ser compilados pelos dois compiladores; em seguida serão definidos os níveis de otimização de código que serão objeto de estudo; os algoritmos serão então compilados e executados e, por fim, os resultados serão analisados para que se possa avaliar se houve melhora e qual seria o compilador mais indicado.


Figura 1 – Diagrama do método utilizado para desenvolvimento do trabalho

A configuração do ambiente computacional não apresentou maiores problemas. Inicialmente foi instalado o compilador GCC juntamente com a IDE Code::Blocks[4]. No

[4] http://www.codeblocks.org/

sistema operacional *Windows* a versão disponível do compilador GCC é o *MinGW*[5] *5.0.* A instalação ocorreu com as configurações padrão do aplicativo.

Em seguida foi instalado o *Microsoft Visual Studio 2015.* Esta IDE é extremamente complexa e completa, sendo que a versão utilizada neste trabalho foi a versão *Enterprise Edition – Trial.* O suporte à linguagem de programação C não é nativo, porém caso seja criado um projeto vazio e arquivos *.c* forem incluídos o compilador é capaz de realizar a compilação. Dessa forma, os mesmos programas compilados no GCC foram utilizados no *Visual Studio.* Após a correta instalação e configuração dos ambientes de programação e dos compiladores era necessário definir quais as otimizações de compilação seriam utilizadas.

O compilador GCC dividiu suas funções de otimização em grupos e, segundo Kumar e Singh (2015), estes grupos de otimizações possuem um relacionamento complexo entre eles de modo que uma otimização pode gerar um executável maior e mais rápido. Os grupos nada mais são que agrupamentos previamente disponibilizados das diversas *flags* de otimização de compilação disponíveis no compilador GCC[6]. No caso do GCC, os agrupamentos disponíveis são:

- O1 – neste caso as *flags* irão tentar reduzir o tamanho do código e o tempo de execução, sem utilizar otimizações que causam um aumento muito grande no tempo de compilação.
- O2 – neste caso o compilador irá executar todas as otimizações suportadas que não envolvam o *tradeoff* espaço x tempo de compilação. Esta opção faz com que as otimizações ocorram no tempo de compilação e no desempenho do código gerado.
- O3 – Executa um nível alto de otimização aumentando ainda mais as *flags* utilizadas.
- OS – Otimização de tamanho do programa, onde todas as *flags* que não aumentam o tamanho do código utilizadas na opção O2 são ativadas.

[5] http://www.mingw.org/
[6] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

O compilador do *Visual Studio* possui algumas configurações mais específicas, porém possui agrupamentos de otimizações da mesma maneira. As possíveis otimizações para este compilador são:

- O1 – combinação das opções abaixo para criar o menor código sempre que possível
- O2 – combinação das opções abaixo para criar o código com menor tempo de execução possível
- Ob – controla o nível de expansão de funções *inline.* Ativado da forma correta para O1 e O2
- Od – simplifica o processo de *debugging*
- Og – esta opção não está mais disponível atualmente pois o compilador executa-a automaticamente. As otimizações executadas são as eliminações de subexpressões, alocação automática de registradores e otimização de *loops.*
- Oi – troca todas as chamadas de função pelas funções propriamente ditas. Aumenta o tamanho do executável.
- Os – favorece um tamanho pequeno de executável
- Ot – favorece um executável mais rápido
- Ox – executa a otimização completa do código considerando tempo de execução e tamanho final do executável
- Oy – desabilita a utilização do *frame pointer* utilizado pelo processador para gerenciar as chamadas de sub-rotinas na memória. Esta opção melhora o tempo de execução, porém prejudica o *debugger*. Só está disponível para arquiteturas *x86*.

Analisando as opções existentes em cada um dos compiladores é possível notar uma equivalência entre os grupos O1, O2 e O3 do compilador GCC, e os grupos O1, O2 e Ox do compilador *Visual Studio*. Sendo assim, estas foram as opções escolhidas para comparação entre os compiladores.

Além das opções de otimização, seria interessante analisar também a questão da ferramenta da *profiling* de código dos compiladores. Existe uma regra chamada "*regra 90-10*" na compilação. Esta regra diz que *90%* do tempo de execução de um código é resultado

de apenas *10%* de seu código fonte (KUMAR; SINGH, 2015). Encontrar qual parte do código é responsável pela maior fatia do tempo de execução é crucial para as otimizações feitas nó código-fonte do software. Estas ferramentas são extremamente intrusivas no código executável gerado e devem ser utilizadas apenas para fins de análise de código.

No caso do GCC a ferramenta é chamada de *gprof.* A utilização da ferramenta está condicionada à adição da *flag* de compilação *-pg* ao se compilar o código. Neste caso o código irá ficar maior e mais lento. Quando o programa é executado um arquivo de saída *gmon.out* é gerado. Este arquivo contém as informações da execução do código e deve ser tratado pelo programa *gprof.* Após a geração do arquivo é necessário executar o *gprof* com as opções desejadas para a análise do código[7]. Existem basicamente dois tipos de perfis para análise de código: o chamado *flat profile* (Figura 2) onde é mostrada a percentagem de tempo que cada função possui em relação a um programa com diversas medidas de tempo (tempo relativo, tempo cumulativo, quantas vezes a função foi invocada); e também o *call graph* (Figura 3) que exibe todas estas informações em formato de DAG (*Directed Acyclic Graph*, ou grafo acíclico direcionado).

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
33.11    801.33   801.33       19    42.18    42.18  ED::Ordenacao<int>::selectSort(int*, unsigned int)
31.44   1562.29   760.97       19    40.05    40.05  ED::Ordenacao<int>::bubbleSort(int*, unsigned int)
20.65   2062.02   499.73       19    26.30    26.30  ED::Ordenacao<int>::insertSort(int*, int)
13.11   2379.36   317.34        9    35.26    35.26  ED::Ordenacao<int>::inverteVetor(int*, unsigned int)
 1.45   2414.43    35.08   570052     0.00     0.00  ED::Ordenacao<int>::particiona(int*, int, int)
 0.06   2415.80     1.37       19     0.07     0.07  ED::Ordenacao<int>::shellSortShell(int*, int)
 0.05   2416.97     1.17       19     0.06     0.06  ED::Ordenacao<int>::shellSortKnuth(int*, int)
 0.05   2418.11     1.13       19     0.06     0.06  ED::Ordenacao<int>::shellSortPardons(int*, int)
 0.03   2418.86     0.75  1244981     0.00     0.00  ED::Ordenacao<int>::heapify(int*, int, int)
 0.03   2419.49     0.63       19     0.03     1.91  ED::Ordenacao<int>::quickSortNonRec(int*, int, int)
 0.03   2420.12     0.63   829981     0.00     0.00  ED::Ordenacao<int>::merge(int*, int*, int, int, int)
 0.01   2420.38     0.26  2280246     0.00     0.00  Pilha<int>::desempilha()
 0.01   2420.62     0.24  2280246     0.00     0.00  Pilha<int>::empilha(int const&)
 0.01   2420.76     0.14  1140142     0.00     0.00  Pilha<int>::vazia()
 0.01   2420.89     0.13 12385997     0.00     0.00  void std::swap<int>(int&, int&)
 0.00   2420.99     0.10      171     0.00     0.00  iniVetor(int, int)
 0.00   2421.07     0.08       19     0.00     0.04  ED::Ordenacao<int>::m_sort(int*, int*, int, int)
 0.00   2421.14     0.07       19     0.00     0.02  ED::Ordenacao<int>::build_heap(int*, int)
 0.00   2421.17     0.03       19     0.00     0.05  ED::Ordenacao<int>::heapSort(int*, int)
 0.00   2421.17     0.00       19     0.00     0.04  ED::Ordenacao<int>::mergeSort(int*, int)
 0.00   2421.17     0.00       19     0.00     0.00  Pilha<int>::Pilha()
 0.00   2421.17     0.00        2     0.00     0.00  std::operator|(std::_Ios_Openmode, std::_Ios_Openmode)
 0.00   2421.17     0.00        1     0.00     0.00  global constructors keyed to iniVetor(int, int)
 0.00   2421.17     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
```

Figura 2 – Exemplo de *flat profile* (STOFFALETTE, 2012).

Esta ferramenta apesar de apresentar todas as informações necessárias não é uma ferramenta que prioriza usabilidade. A apresentação das informações em formato de texto não é mais atrativa. Neste caso a ferramenta de *profiling* do *Visual Studio* apresenta diversas vantagens. A opção é ativada no menu Depurar > Criador de Perfil de Desempenho.

[7] ftp://ftp.gnu.org/pub/old-gnu/Manuals/gprof/html_chapter/gprof_4.html#SEC4

Quando é iniciado o processo clicando-se no botão iniciar, um assistente de desempenho é aberto onde podem ser escolhidos 4 métodos diferentes (Figura 4): a **Amostragem de CPU** que irá apresentar o desempenho linha a linha do código de forma completa, a **Instrumentação** que irá apresentar o tempo gasto em cada função do código, **Alocação de memória do .NET** para aplicações que utilizam a plataforma e também a opção **Dados de contenção de recursos** para a verificação de chamadas de *threads*.

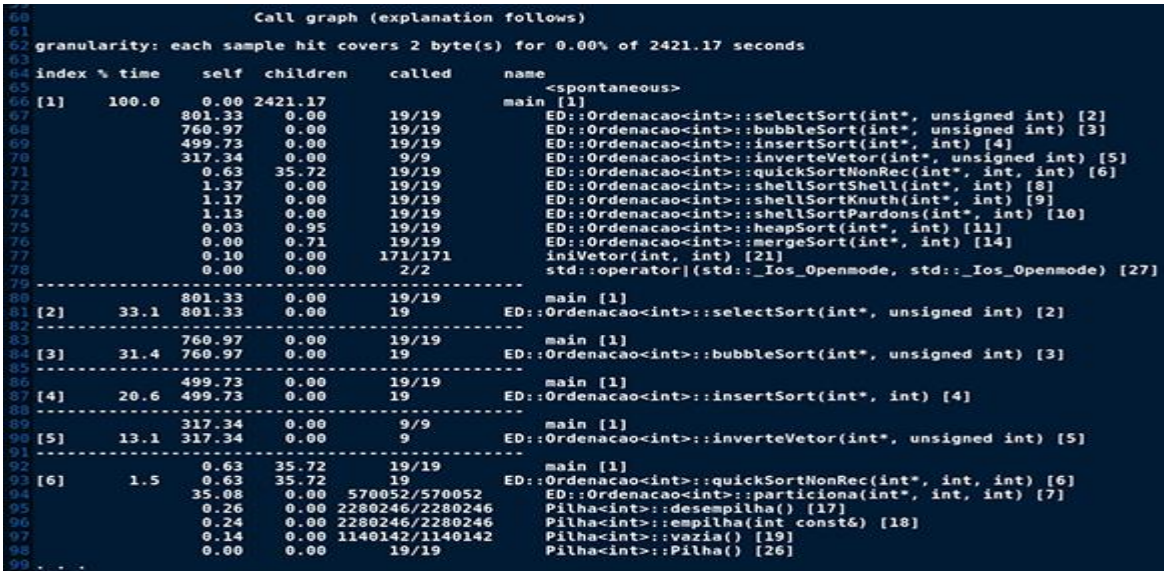


Figura 3 – Exemplo de *call graph* (STOFFALETTE, 2012)

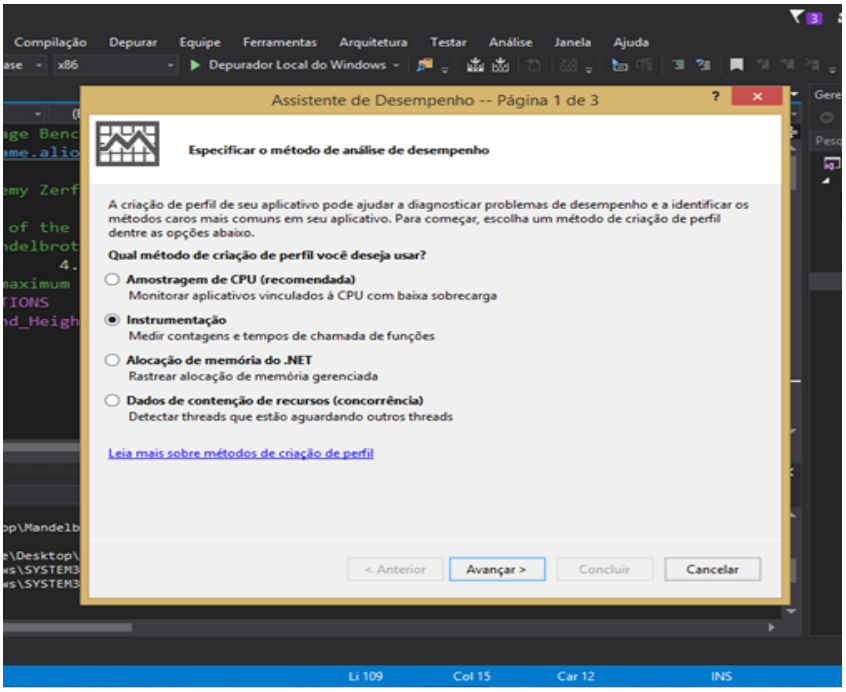


Figura 4 – Escolha do método de análise no *Visual Studio 2015*

No caso dos *benchmarks* deste trabalho as opções válidas são a Amostragem de CPU e a Instrumentação. Após a escolha do método é possível escolher qual projeto, executável, aplicação ASP .NET ou DLL será avaliada. A análise gerada pela opção Instrumentação gera um gráfico de uso de CPU, e permite que seja escolhida a opção Modo de Exibição Atual > Funções para análise mais detalhada do código (Figuras 5 e 6).


Figura 5 – Exemplo de resultado da análise Instrumentação

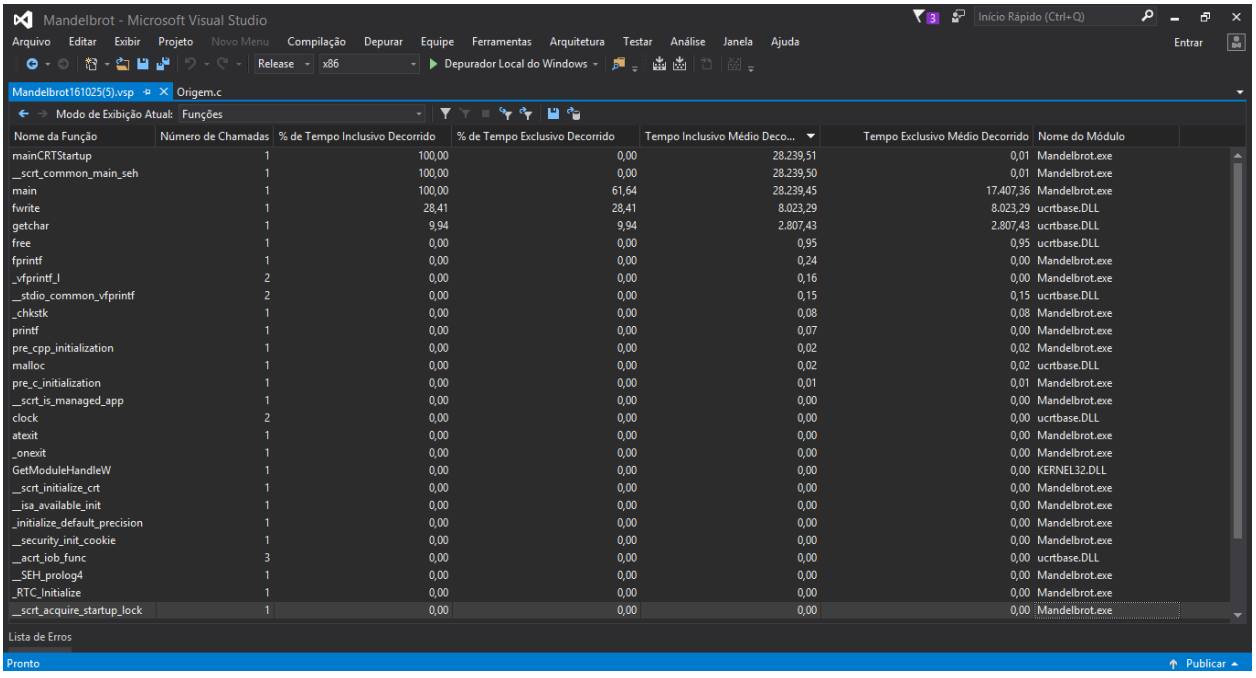


Figura 6 – Exemplo de listagem de tempo de funções do código

Caso seja escolhida a opção Amostragem de CPU o resultado da tela inicial será a porcentagem de uso da CPU utilizada para executar o código, e quando solicitado o resultado das linhas de código em Modo de Exibição Atual > Detalhes da Função diversas informações sobre a função são exibidas (incluindo a indicação das linhas de código). Exemplos de resultados podem ser vistos nas Figuras 7 e 8.

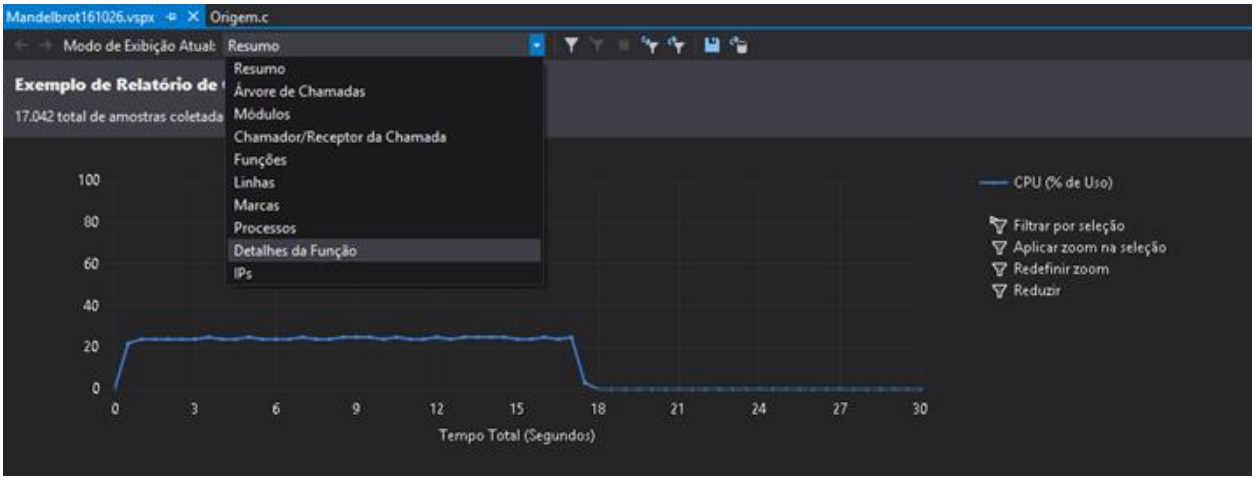


Figura 7 – Exemplo de resultado da análise Amostragem de CPU

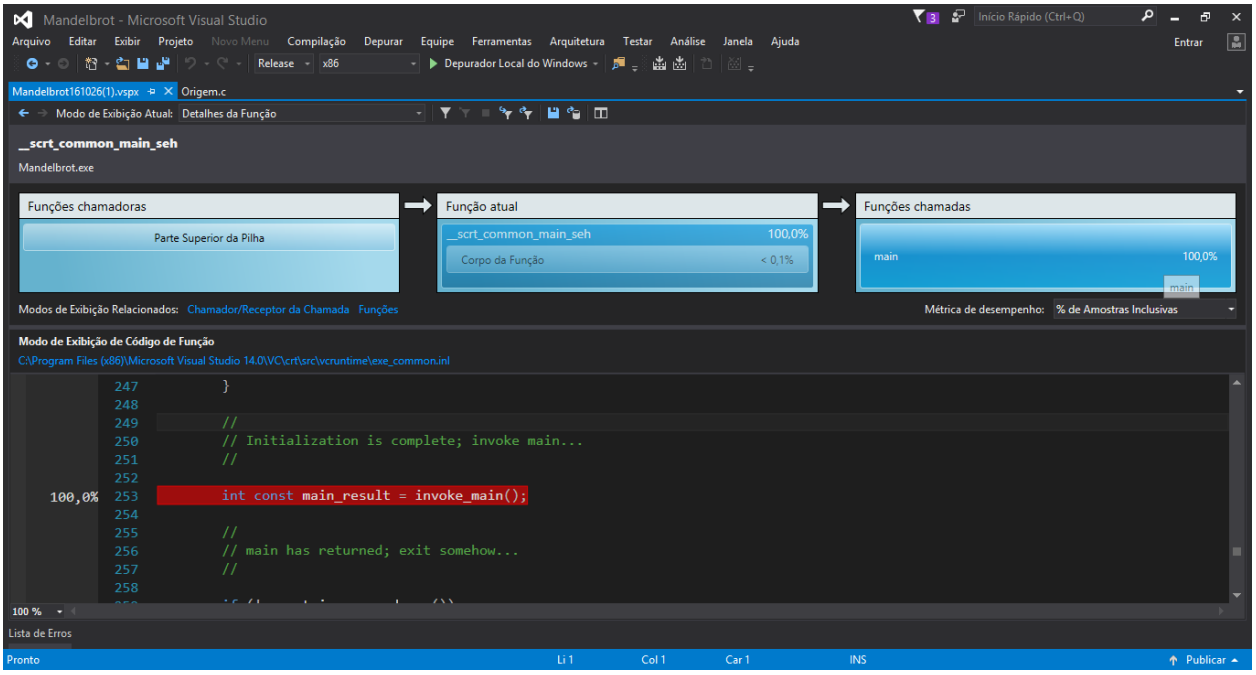


Figura 8 – Exemplo de relatório de detalhamento de funções

Considerando a importância das ferramentas apresentadas e as possibilidades de otimização de compilação, os códigos foram compilados e executados para as *3* otimizações possíveis e também para a utilização de ferramentas de *profiling* de código. Os resultados e a analise são apresentados a seguir.

## 3 Resultados

A Tabela 1 apresenta os dados de todos os resultados da execução dos códigos. Na tabela a linha GCC representa o resultado para o compilador GCC e a linha Visual para o compilador do *Visual Studio.* Os três *benchmarks* são indicados por *Mandelbrot,* Gerador (representando o algoritmo *FASTA)* e Complementos (para o *benchmark* que calcula a cadeia complementar para as sequências de DNA apresentadas). O tempo de execução é apresentado em segundos e foi medido com funções da biblioteca *time.h*. O tamanho dos arquivos são apresentados em KB segundo medição do sistema operacional *Windows* utilizando sistema de arquivos *ntfs*.

Tabela 1 – Comparação de resultados para os parâmetros definidos

| | | ***Mandelbrot*** | | ***Gerador*** | | ***Complementos*** | |
|---|---|---|---|---|---|---|---|
| | | Tempo Execução(s) | Tam Arq em KB | Tempo Execução(s) | Tam Arq em KB | Tempo Execução(s) | Tam Arq em KB |
| GCC | Default | 68.632 | 28 | 14.252 | 30 | 5.001 | 27 |
| | O1 | 36.938 | 27 | 8.376 | 29 | 4.735 | 27 |
| | O2 | 36.350 | 27 | 8.016 | 29 | 4.812 | 27 |
| | O3 | 36.461 | 28 | 10.867 | 29 | 4.547 | 27 |
| | Profile | 68.803 | 39 | 13.457 | 40 | 5.063 | 38 |
| Visual | Default | 35.294 | 7 | 5.122 | 8 | 3.834 | 6 |
| | O1 | 25.970 | 5 | 2.512 | 7 | 3.496 | 6 |
| | O2 | 24.452 | 6 | 2.441 | 7 | 3.455 | 6 |
| | OX | 24.113 | 6 | 3.167 | 8 | 3.437 | 6 |
| | Profile | 29.512 | 12 | 5.845 | 11 | 3.556 | 12 |

Considerando as informações presentes na Tabela 1 é possível analisar com detalhes alguns resultados. Primeiramente a questão do tamanho do arquivo, que apresenta um padrão. As otimizações O1 e O2 de ambos os compiladores apresentam o mesmo tamanho de executável exceto pelo primeiro *benchmark* compilado no *Visual Studio.* Este resultado mostra que independe do grupo escolhido a solução final também irá promover uma otimização neste sentido.

A diferença entre o tamanho dos arquivos executáveis dos dois compiladores é muito significativa. Este resultado vem do fato de que o compilador GCC é desenvolvido por programadores que não possuem acesso ao sistema operacional *Windows* e, portanto, não possuem condição de utilizar todas as suas ferramentas para uma geração de código otimizada. Já no caso do código compilado pelo *Visual Studio* a situação é inversa, pois os desenvolvedores do compilador têm acesso total ao sistema por serem a mesma empresa, podendo utilizar melhor seus recursos computacionais.

Comportando-se como esperado, o resultado da compilação sem otimizações (*default*) apresenta um tempo de execução consideravelmente maior em relação aos resultados otimizados.

Isso não acontece apenas no caso do programa que calcula as sequências complementares de DNA que, pelos resultados, apresenta o menor desafio computacional entre os *3 benchmarks* escolhidos. Este fato faz com que as otimizações tenham pouco efeito em seu código.

O tempo de execução dos programas mostra que o compilador do *Visual Studio* é muito mais eficiente na geração de código otimizado em comparação ao GCC para os testes realizados, apresentando uma melhora de pelo menos *15%* em todos os casos.

Ao descrever as ferramentas de *profiling* no capítulo anterior foi mencionado que as medições normalmente são feitas através de rotinas intrusivas modificando o código original e piorando tanto o tamanho do executável quanto o tempo de execução. Os resultados deste trabalho mostram que as ferramentas de *profiling* dos dois compiladores causam um aumento significativo no tempo de execução e no tamanho final do código.

## Considerações finais

Este trabalho apresentou um estudo comparativo entre dois compiladores amplamente utilizados para desenvolvimento de software, o GCC e o compilador do *Microsoft Visual Studio 2015.* Foram escolhidos *3 benchmarks* para que a comparação fosse validada e os resultados foram apresentados seguidos de uma análise.

Este trabalho permite concluir que ambos os compiladores possuem sua importância no cenário do desenvolvimento, porém o compilador do *Visual Studio* apresenta resultados

melhores em diversas situações. Este resultado é consequência direta do fato de ser desenvolvido pelo mesmo fabricante do sistema operacional utilizado. Também é possível notar que as ferramentas de *profiling* são úteis e intrusivas, sendo que devem ser utilizadas apenas nas versões de desenvolvimento do software.

Os trabalhos futuros devem priorizar a comparação de um número maior de compiladores e também a utilização de outros *benchmarks.* O ideal seria a utilização de um sistema operacional neutro, como alguma distribuição do Linux, para avaliação evitando que alguns casos apresentem viés no resultado.

Também é importante salientar que os trabalhos mais recentes na área de compiladores estão voltados ao desenvolvimento de otimizações de código específicas para determinadas operações (COX et al, 2009), análise de desempenho para um amplo espaço de aplicações (KUMAR; SINGH, 2015), construção de ferramentas para configuração automática de compiladores (PLOTNIKOV et al, 2013) e também a construção de compiladores que geram um co-projeto de hardware/software para execução de um código e não mais uma solução apenas baseada em software (CARDOSO et al, 2013).

## Referências

CARDOSO, J. M. P.; DINIZ, P. C.; COUTINHO, J. G. F.; PETROV, Z. M. **Compilation and synthesis for embedded reconfigurable systems an aspect-oriented approach**. Nova York: Springer-Verlag, 2013.

COX, A. et al. **Compiler Optimization Techniques**. 2009. Disponível em: <oustafamohamed.com/papers/compiler.pdf>. Acesso em: 2 fev. 2017.

GRAY, J. **The Benchmark Handbook for Database and Transaction Systems**. 2. ed. Burlington, Massachusetts, EUA: Morgan Kaufmann, 1993.

KUMAR, T.; SINGH, R. K.; Analysis of compiler optimization techniques by using feature mining technique. In: NATIONAL SYSTEMS CONFERENCE (NSC), 39., **Proceedings...** Noida, 2015. p. 1-6.

PLOTINIKOV, D. et al. An Automatic tool for tuning compiler optimizations. In: NINTH INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE AND INFORMATION TECHNOLOGIES. **Proceedings...** Yerevan, 2013. p. 1-7.

STOFFALETTE, J. R. **Introdução ao GPROF**. IBM Developer Works Brasil. 2012.