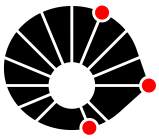
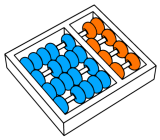




LLVM - Implementando um Pragma OpenMP

F. O. Emos *G. C. S. Araujo*

Relatório Técnico - IC-PFG-19-47
Projeto Final de Graduação
2019 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

LLVM - Implementando um Pragma OpenMP

Felipe de Oliveira Emos * Guido Costa Souza de Araujo†

Resumo

Por diversos interesses tanto empresariais quanto acadêmicos, o compilador LLVM é hoje um dos compiladores mais relevantes que existe. Modificar e adaptar o código do LLVM, seja qual for o motivo, é uma tarefa que vem se tornando cada vez mais comum para pesquisadores e desenvolvedores da área.

Infelizmente, tal tarefa não é tão convidativa quanto poderia ser. O LLVM é um projeto muito grande e rebuscado e por mais que exista a documentação oficial, ela se preocupa muito mais em enumerar e descrever os compontentes do compilador do que em propriamente explicar os funcionamentos se suas partes. Materiais de apoio (que explicam tais funcionamentos) existem e inclusive vários deles são oficiais do LLVM, mas eles são focados ou nas partes mais gerais do compilador ou nos primeiros passos de um iniciante na plataforma. Existe uma carência de explicação mais detalhada para algumas partes mais específicas do LLVM.

O projeto "LLVM - Implementando um Pragma OpenMP"foi a confecção de um material didático explicativo. A ideia do material é facilitar o trabalho de qualquer um que queira se aventurar na exploração de uma parte específica do compilador LLVM, que são as partes do Clang que dizem respeito às anotações "Pragma"da interface de programação "OpenMP".

1 Introdução

Compiladores como LLVM ou GCC são códigos de alta performance, ou seja, a natureza da aplicação impõe a eles um enorme incentivo para que sejam o mais otimizados o possível. Qualquer otimização em seus funcionamentos resulta em um impacto considerável na velocidade de produção de software.

Eis então o dilema entre fazer um código de boa compreeensão/manutenção ou fazer o código mais rápido possível. Claro, na maioria das vezes um não impede o outro, mas se tratando de códigos assim tão grandes como o LLVM é inevitável que ocorram estruturações um tanto quanto confusas *"for the sake of optimization"*. A própria escolha da linguagem desses compiladores, C para o GCC e C++ para o LLVM, revela a natureza da preocupação com performance, visto que estas são algumas das linguagens de alto nível mais rápidas que se conhece.

*felipe.emos.computacao@gmail.com

†Inst. de Computação, UNICAMP, 13083-852 Campinas, SP. guido@ic.unicamp.br

O problema ocorre quando se abre mão demais da compreensão e o código se torna o oposto de convidativo. Estamos muito acostumados a subestimar o fator humano, entretanto é um fato que a qualidade de código é importantíssima e não deveria ser deixada de lado. O GCC, por exemplo, é famoso por ter um código bastante selvagem para os aventureiros iniciantes. Mesmo os aventureiros mais experientes acabam evitando modificar o GCC, já que é possível sofrer bem menos e obter praticamente o mesmo resultado desejado em uma modificação do LLVM. O LLVM é mais modular, tem orientação a objetos, tem um bom tutorial para iniciantes e tem uma grande comunidade ativa e bastante convidativa, conhecida por sua rápida resposta a emails de novatos.

Este é um bom exemplo em que o grau de convidatividade do código e a qualidade do código se mostram fatores decisivos na seleção natural que ocorre entre softwares, no caso LLVM e GCC. Por enquanto o GCC ainda vive (muito por uma questão de legado), porém em certos ambientes, como por exemplo a pesquisa acadêmica, o LLVM parece que já venceu a batalha. É preciso boa convidatividade para que novos desenvolvedores e pesquisadores se engajem e modifiquem o código com facilidade, contribuindo para a evolução da plataforma ao longo prazo. Este é um fator humano muito forte e, para os softwares, é uma questão de sobrevivência.

Dada a importância desta questão, é preciso dizer que o LLVM não é perfeito, C++ ainda é uma linguagem bastante verbosa e um certo grau de "não convidatividade"beira o inevitável. Colocando em perspectiva, hoje o código do GCC contém aproximadamente 5 milhões de linhas de código e o LLVM contém 1.6 milhões de linhas de código [2]. Esta quantidade impensável de linhas de código pode ser mais ou menos convidativa, mas é fato que sua compreensão ainda será bastante trabalhosa. O LLVM não é um mar de rosas.

2 Objetivo

O objetivo deste trabalho "LLVM - Implementando um Pragma OpenMP"vai nesse sentido de facilitar a compreensão do compilador, se atendo a uma parte específica do LLVM: as anotações Pragma OpenMP. O LLVM já tem materiais de apoio para facilitar o desenvolvedor novato, como por exemplo o famoso tutorial "Getting Started", da página inicial do compilador [3]. Infelizmente, graças à grande escala do compilador, eles acabam tendo um escopo bastante reduzido em relação ao todo. Outro ponto importante é que a documentação do LLVM (também por uma questão da grande escala de seu código) se preocupa mais em enumerar e descrever as diversas partes do compilador, se atentando bem menos a explicar os funcionamentos das peças.

Um documento que explicasse a mecânica de um Pragma OpenMP foi uma necessidade que os autores deste trabalho sentiram durante um outro projeto de pesquisa, feito pela UNICAMP em parceria com a empresa IBM no ano de 2018. A falta de experiência em LLVM do pesquisador, que é algo perfeitamente esperado de um começo de projeto de pesquisa, se combinou à falta de um tutorial que tangesse tal pedaço do Clang/LLVM. O resultado foi que muito tempo e recursos valiosos foram gastos em questões puramente técnicas de especificidades do LLVM. Julgou-se depois que isso foi desnecessário, já que bastaria ter em mãos um bom mapa que a história teria sido diferente.

Este documento didático agora está feito, o mapa está disponível para consulta e espera-se que ele seja compreensivo o suficiente para que outros não tenham que passar pelo mesmo esforço dos autores. O trabalho que você está lendo é apenas um relatório sucinto sobre a confecção de tal documento, é como um reconhecimento (na forma de relatório) de que ele foi feito e existe. As especificidades técnicas do LLVM, que estão no escopo do documento, não serão explicadas aqui. O leitor interessado deve sentir-se convidado a examinar o documento didático por si só. O link está ao final do relatório, na sessão de "Referências": "LLVM - Implementando um Pragma OpenMP"[1].

3 Conclusão

A diferença que um material didático como este pode fazer não é algo que deve ser subestimado. Por exemplo, a NVIDIA fez uma apresentação chamada *Targeting GPUs with OpenMP4.5 Device Directives* [4] na GPU Technology Conference de 2016. Tal apresentação foi de fundamental importância para a pesquisa dos autores, feita em parceria com a IBM e que acabou inspirando este projeto de material didático. O estudo, dentre outras coisas, envolvia offloading de códigos OpenMP usando Clang. Nenhum outro material didático encontrado na época explicava com clareza o que era preciso ser feito para que o offloading funcionasse corretamente. O pedaço da apresentação que foi verdadeiramente valioso foi bem pequeno, coisa de poucas linhas, porém estas poucas linhas de especificidades do Clang/OpenMP facilmente se perdem no mar das documentações e, se não fosse a apresentação da NVIDIA, existiria ali um bom potencial de perda de tempo e recursos.

Em um código de 1.6 milhões de linhas, é de se esperar que muita coisa não precise de uma documentação explicativa, o código é autoexplicativo. Entretanto, também é de se esperar que várias linhas de código que precisam ficarão desamparadas em documentação.

Na prática, quem mexe com esses trechos de código mais específicos (como por exemplo os Pragmas do OpenMP) costuma ser alguém experiente em LLVM e esta pessoa terá facilidade na compreensão e não irá se dar o trabalho de descrever cautelosamente o seu funcionamento para os outros. É bastante natural que aquele que está mais apto a produzir um documento explicativo é também aquele que menos precisa de um. Logo, a menos que exista um incentivo muito grande para a criação de um documento mais detalhado, que é o caso dos materiais de apoio sobre as partes mais gerais e importantes do compilador, ou mesmo sobre os primeiros passos que um iniciante deve tomar (como o tutorial "Getting Started"[3]), dificilmente alguém competente irá dedicar seu tempo precioso para criar um tutorial de LLVM. O resultado é que partes específicas do compilador ficam desamparadas de explicação.

Por esses motivos, dificilmente alguém que tenha explorado esta parte do LLVM faria um tutorial como este. Mas, agora que ele existe, talvez seja de boa valia para alguém no futuro. Provavelmente será, já que avaliando apenas a UNICAMP as pesquisas em LLVM se mostram bastante promissoras e atrativas. As perspectivas de modificações do LLVM/Clang são boas e oportunidades de pesquisa no assunto se projetam em fartura.

Referências

[1] Documento Didático - *LLVM - Implementando um Pragma OpenMP*

https://drive.google.com/file/d/1S4A3zcNHhYYLenn55hvsVmbUiolr07jC/view?usp=sharing

[2] How much does a compiler cost?, *An analysis by David A Wheeler's SLOCCount.*

https://www.embecosm.com/2018/02/26/how-much-does-a-compiler-cost/
https://dwheeler.com/sloccount/

[3] LLVM Documentation, *Getting Started/Tutorials.*

https://llvm.org/docs/GettingStartedTutorials.html

[4] NVIDIA Presentation, *Targeting GPUs with OpenMP4.5 Device Directives.*

http://on-demand.gputechconf.com/gtc/2016/presentation/s6510-jeff-larkin-targeting-gpus-openmp.pdf

https://www.youtube.com/watch?v=4nusBV3gWJc

LLVM - Implementing an OpenMP Pragma - Overview

Document created using
OpenMP 4.5
Clang-7

Unicamp IC - 2019
Author: Felipe de Oliveira Emos
Guidance: Guido C. Souza Araujo

Parse Pragmas

Pragmas are analysed using a class called PragmaHandler

This is for all pragmas, it’s not OpenMP specific

PragmaHandlers are initialized and defined in the file ParsePragma.cpp

ParsePragma.cpp

```
void Parser::initializePragmaHandlers() {
        . . .
        . . .
        if (getLangOpts().OpenMP)
                OpenMPHandler.reset(new PragmaOpenMPHandler());
        else
                OpenMPHandler.reset(new PragmaNoOpenMPHandler());

        PP.AddPragmaHandler(OpenMPHandler.get());
        . . .
        . . .
}
```

ParsePragma.cpp

```
void Parser::initializePragmaHandlers() {
        ...
        ...
        if (getLangOpts().OpenMP)
                OpenMPHandler.reset(new PragmaOpenMPHandler());
        else
                OpenMPHandler.reset(new PragmaNoOpenMPHandler());

        PP.AddPragmaHandler(OpenMPHandler.get());
        ...
        ...
}
```

This variable is true if the flag "-fopenmp" was used during compilation

These driver options are defined in the file "Driver/Options.td"

Driver/Options.td

```
1500 def fopenmp : Flag<["-"], "fopenmp">, Group<f_Group>, Flags<[CC1Option, NoArgumentUnused]>,
1501   HelpText<"Parse OpenMP pragmas and generate parallel code.">;
```

ParsePragma.cpp

```
void Parser::initializePragmaHandlers() {
        . . .
        . . .
        if (getLangOpts().OpenMP)
                OpenMPHandler.reset(new PragmaOpenMPHandler());
        else
                OpenMPHandler.reset(new PragmaNoOpenMPHandler());

        PP.AddPragmaHandler(OpenMPHandler.get());
        . . .
        . . .
}
```

Handler when compiled with flag “-fopenmp”

Handler when compiled without flag “-fopenmp”

ParsePragma.cpp

```
struct PragmaOpenMPHandler : public PragmaHandler {
        PragmaOpenMPHandler() : PragmaHandler("omp") { }

        void HandlePragma(
                Preprocessor &PP,
                PragmaIntroducerKind Introducer,
                Token &FirstToken) override;
};
```

```
struct PragmaNoOpenMPHandler : public PragmaHandler {
        PragmaNoOpenMPHandler() : PragmaHandler("omp") { }

        void HandlePragma(
                Preprocessor &PP,
                PragmaIntroducerKind Introducer,
                Token &FirstToken) override;
};
```

ParsePragma.cpp

```
struct PragmaOpenMPHandler : public PragmaHandler {
        PragmaOpenMPHandler() : PragmaHandler("omp") { }

        void HandlePragma(
                Preprocessor &PP,
                PragmaIntroducerKind Introducer,
                Token &FirstToken) override;
};
```

```
struct PragmaNoOpenMPHandler : public PragmaHandler {
        PragmaNoOpenMPHandler() : PragmaHandler("omp") { }

        void HandlePragma(
                Preprocessor &PP,
                PragmaIntroducerKind Introducer,
                Token &FirstToken) override;
};
```

PragmaHandler objects are used mainly through their method “**HandlePragma**” this is the important part.

So the proper way to do this is to derive your new handler from the “PragmaHandler” class and define your own “HandlePragma” method.

ParsePragma.cpp

```
struct PragmaOpenMPHandler : public PragmaHandler {
        PragmaOpenMPHandler() : PragmaHandler("omp") { }

        void HandlePragma(
                Preprocessor &PP,
                PragmaIntroducerKind Introducer,
                Token &FirstToken) override;
};
```

Identifier of the pragma

Ex:
#pragma *ID*
#pragma ***omp***

```
struct PragmaNoOpenMPHandler : public PragmaHandler {
        PragmaNoOpenMPHandler() : PragmaHandler("omp") { }

        void HandlePragma(
                Preprocessor &PP,
                PragmaIntroducerKind Introducer,
                Token &FirstToken) override;
};
```

Actual computation of tokens.
This is called when clang encounters a "#pragma omp"

This is just the signature of the method. The definition is also in this file . . .

ParsePragma.cpp

```
1521 /// \brief Handle '#pragma omp ...' when OpenMP is enabled.
1522 ///
1523 void
1524 PragmaOpenMPHandler::HandlePragma(Preprocessor &PP,
1525                                   PragmaIntroducerKind Introducer,
1526                                   Token &FirstTok) {
1527   SmallVector<Token, 16> Pragma;
1528   Token Tok;
1529   Tok.startToken();
1530   Tok.setKind(tok::annot_pragma_openmp);
1531   Tok.setLocation(FirstTok.getLocation());
1532
1533   while (Tok.isNot(tok::eod)) {
1534     Pragma.push_back(Tok);
1535     PP.Lex(Tok);
1536   }
1537   SourceLocation EodLoc = Tok.getLocation();
1538   Tok.startToken();
1539   Tok.setKind(tok::annot_pragma_openmp_end);
1540   Tok.setLocation(EodLoc);
1541   Pragma.push_back(Tok);
1542
1543   auto Toks = llvm::make_unique<Token[]>(Pragma.size());
1544   std::copy(Pragma.begin(), Pragma.end(), Toks.get());
1545   PP.EnterTokenStream(std::move(Toks), Pragma.size(),
1546                       /*DisableMacroExpansion=*/false);
1547 }
```

ParsePragma.cpp

```
1521 /// \brief Handle '#pragma omp ...' when OpenMP is enabled.
1522 ///
1523 void
1524 PragmaOpenMPHandler::HandlePragma(Preprocessor &PP,
1525                                   PragmaIntroducerKind Introducer,
1526                                   Token &FirstTok) {
1527   SmallVector<Token, 16> Pragma;
1528   Token Tok;
1529   Tok.startToken();
1530   Tok.setKind(tok::annot_pragma_openmp);
1531   Tok.setLocation(FirstTok.getLocation());
1532
1533   while (Tok.isNot(tok::eod)) {
1534     Pragma.push_back(Tok);
1535     PP.Lex(Tok);
1536   }
1537   SourceLocation EodLoc = Tok.getLocation();
1538   Tok.startToken();
1539   Tok.setKind(tok::annot_pragma_openmp_end);
1540   Tok.setLocation(EodLoc);
1541   Pragma.push_back(Tok);
1542
1543   auto Toks = llvm::make_unique<Token[]>(Pragma.size());
1544   std::copy(Pragma.begin(), Pragma.end(), Toks.get());
1545   PP.EnterTokenStream(std::move(Toks), Pragma.size(),
1546                       /*DisableMacroExpansion=*/false);
1547 }
```

Token that represents: #pragma omp

Get the next Token from Lexer

Token that marks the end of pragma

```
1521 /// \brief Handle '#pragma omp ...' when OpenMP is enabled.
1522 ///
1523 void
1524 PragmaOpenMPHandler::HandlePragma(Preprocessor &PP,
1525                                   PragmaIntroducerKind Introducer,
1526                                   Token &FirstTok) {
1527   SmallVector<Token, 16> Pragma;
1528   Token Tok;
1529   Tok.startToken();
1530   Tok.setKind(tok::annot_pragma_openmp);
1531   Tok.setLocation(FirstTok.getLocation());
1532
1533   while (Tok.isNot(tok::eod)) {
1534     Pragma.push_back(Tok);
1535     PP.Lex(Tok);
1536   }
1537   SourceLocation EodLoc = Tok.getLocation();
1538   Tok.startToken();
1539   Tok.setKind(tok::annot_pragma_openmp_end);
1540   Tok.setLocation(EodLoc);
1541   Pragma.push_back(Tok);
1542
1543   auto Toks = llvm::make_unique<Token[]>(Pragma.size());
1544   std::copy(Pragma.begin(), Pragma.end(), Toks.get());
1545   PP.EnterTokenStream(std::move(Toks), Pragma.size(),
1546                       /*DisableMacroExpansion=*/false);
1547 }
```

ParsePragma.cpp

Creates a SmallVector Pragma

All Tokens are pushed to this vector

The vector is copyed and passed to (PP) Preprocessor's TokenStream

Parse OpenMP

Although the pragma handlers deal with OpenMP pragmas, their job is pretty much to pass on the computation to other parts of Clang. How this computation is structured and executed is what's coming next in this document.

The OpenMP definitions are mostly contained within the files of OpenMPKinds

We'll take a look at OpenMPKinds, but before that it's important to get ourselves comfortable with a recurrent construct in Clang: the Definition Files and how their inclusion affects the code

Definition files

In clang you will find definition files being used many times

Usually the code that includes the definition file looks like this:

OpenMPKinds.h

```
22 /// OpenMP directives.
23 enum OpenMPDirectiveKind {
24 #define OPENMP_DIRECTIVE(Name) \
25   OMPD_##Name,
26 #define OPENMP_DIRECTIVE_EXT(Name, Str) \
27   OMPD_##Name,
28 #include "clang/Basic/OpenMPKinds.def"
29   OMPD_unknown
30 };
```

First there are definitions of macros that are going to be used. They work like a “filter”.

Then the file is included and code is generated through the macro.

OpenMPKinds.h

```
22 /// OpenMP directives.
23 enum OpenMPDirectiveKind {
24 #define OPENMP_DIRECTIVE(Name) \
25   OMPD_##Name,
26 #define OPENMP_DIRECTIVE_EXT(Name, Str) \
27   OMPD_##Name,
28 #include "clang/Basic/OpenMPKinds.def"
29   OMPD_unknown
30 };
```

OpenMPKinds.def

```
// OpenMP directives.
OPENMP_DIRECTIVE(threadprivate)
OPENMP_DIRECTIVE(parallel)
OPENMP_DIRECTIVE(task)
OPENMP_DIRECTIVE(simd)
OPENMP_DIRECTIVE(for)
```

```
OPENMP_DIRECTIVE_EXT(target_data, "target data")
OPENMP_DIRECTIVE_EXT(target_enter_data, "target enter data")
OPENMP_DIRECTIVE_EXT(target_exit_data, "target exit data")
OPENMP_DIRECTIVE_EXT(target_parallel, "target parallel")
OPENMP_DIRECTIVE_EXT(target_parallel_for, "target parallel for")
```

First there are definitions of macros that are going to be used. They work like a “filter”.

Then the file is included and code is generated through the macro.

Many of the lines in the definition file will be matched with the macro.

OpenMPKinds.h

```
22 /// OpenMP directives.
23 enum OpenMPDirectiveKind {
24 #define OPENMP_DIRECTIVE(Name) \
25   OMPD_##Name,
26 #define OPENMP_DIRECTIVE_EXT(Name, Str) \
27   OMPD_##Name,
28 #include "clang/Basic/OpenMPKinds.def"
29   OMPD_unknown
30 };
```

OpenMPKinds.def

```
15 #ifndef OPENMP_DIRECTIVE
16 #  define OPENMP_DIRECTIVE(Name)
17 #endif
18 #ifndef OPENMP_DIRECTIVE_EXT
19 #define OPENMP_DIRECTIVE_EXT(Name, Str)
20 #endif
```

First there are definitions of macros that are going to be used. They work like a “filter”.

Then the file is included and code is generated through the macro.

All this is only possible because of those countless #ifndef placed at the beginning of the definition file, they define blank macros for all of those definitions out of the “filter”, therefore they are “erased” and do not generate code.

OpenMPKinds.h

```
22 /// OpenMP directives.
23 enum OpenMPDirectiveKind {
24 #define OPENMP_DIRECTIVE(Name) \
25   OMPD_##Name,
26 #define OPENMP_DIRECTIVE_EXT(Name, Str) \
27   OMPD_##Name,
28 #include "clang/Basic/OpenMPKinds.def"
29   OMPD_unknown
30 };
```

OpenMPKinds.def

```
// OpenMP directives.
OPENMP_DIRECTIVE(threadprivate)
OPENMP_DIRECTIVE(parallel)
OPENMP_DIRECTIVE(task)
OPENMP_DIRECTIVE(simd)
OPENMP_DIRECTIVE(for)
```

```
OPENMP_DIRECTIVE_EXT(target_data, "target data")
OPENMP_DIRECTIVE_EXT(target_enter_data, "target enter data")
OPENMP_DIRECTIVE_EXT(target_exit_data, "target exit data")
OPENMP_DIRECTIVE_EXT(target_parallel, "target parallel")
OPENMP_DIRECTIVE_EXT(target_parallel_for, "target parallel for")
```

In summary, **what is going on**?

When OpenMPKinds.h includes the definition file, many of the lines in OpenMPKinds.def will be matched with the #define macros and we'll get the desired effect, which in this case is to populate the "enum" with all the definitions of OpenMP directives

OpenMPKinds.h

```
22 /// OpenMP directives.
23 enum OpenMPDirectiveKind {
24 #define OPENMP_DIRECTIVE(Name) \
25   OMPD_##Name,
26 #define OPENMP_DIRECTIVE_EXT(Name, Str) \
27   OMPD_##Name,
28 #include "clang/Basic/OpenMPKinds.def"
29   OMPD_unknown
30 };
```

OpenMPKinds.def

```
// OpenMP directives.
OPENMP_DIRECTIVE(threadprivate)
OPENMP_DIRECTIVE(parallel)
OPENMP_DIRECTIVE(task)
OPENMP_DIRECTIVE(simd)
OPENMP_DIRECTIVE(for)
```

```
OPENMP_DIRECTIVE_EXT(target_data, "target data")
OPENMP_DIRECTIVE_EXT(target_enter_data, "target enter data")
OPENMP_DIRECTIVE_EXT(target_exit_data, "target exit data")
OPENMP_DIRECTIVE_EXT(target_parallel, "target parallel")
OPENMP_DIRECTIVE_EXT(target_parallel_for, "target parallel for")
```

Tip: Attention is key

If at some point an error or strange behaviour happens and it has something to do with the variable “OMPD_parallel”, before you start your depuration journey, think for a moment:

Does this variable exist on the code?

Well, it **doesn’t**, don’t waste your time looking for it. It was created during compilation time.

Be careful

It’s not always that this technique of macro (“filter”) and inclusion uses a **definition file from source**.
Sometimes the definition file is generated during compilation.

```
59   // Initialize the table on the first use.
60   Initialized = true;
61 #define ABSTRACT_STMT(STMT)
62 #define STMT(CLASS, PARENT) \
63   StmtClassInfo[(unsigned)Stmt::CLASS##Class].Name = #CLASS;    \
64   StmtClassInfo[(unsigned)Stmt::CLASS##Class].Size = sizeof(CLASS);
65 #include "clang/AST/StmtNodes.inc"
66
```

StmtNodes.inc doesn’t exist in clang’s code, don’t waste your time looking for it.

Be careful

The rule of thumb is:

- if the definition file has extension “.def”, then it is somewhere in source code.

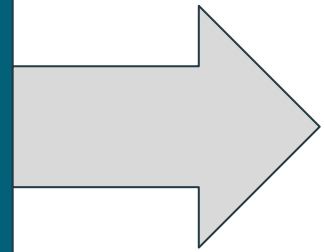
- if the definition file has extension “.inc”, then it was generated during compilation time and it is located inside the build directory of your choice.

  All “.inc” are generated based on a “.td” file of the same name. All “.td” are **source** files, so modify “.td” in order to modify the “.inc” that will be generated during compilation

StmtNodes.td

```
200 // OpenMP Directives.
201 def OMPExecutableDirective : Stmt<1>;
202 def OMPLoopDirective : DStmt<OMPExecutableDirective, 1>;
203 def OMPParallelDirective : DStmt<OMPExecutableDirective>;
204 def OMPSimdDirective : DStmt<OMPLoopDirective>;
205 def OMPForDirective : DStmt<OMPLoopDirective>;
206 def OMPForSimdDirective : DStmt<OMPLoopDirective>;
207 def OMPSectionsDirective : DStmt<OMPExecutableDirective>;
208 def OMPSectionDirective : DStmt<OMPExecutableDirective>;
209 def OMPSingleDirective : DStmt<OMPExecutableDirective>;
210 def OMPMasterDirective : DStmt<OMPExecutableDirective>;
211 def OMPCriticalDirective : DStmt<OMPExecutableDirective>;
212 def OMPParallelForDirective : DStmt<OMPLoopDirective>;
213 def OMPParallelForSimdDirective : DStmt<OMPLoopDirective>;
214 def OMPParallelSectionsDirective : DStmt<OMPExecutableDirective>;
215 def OMPTaskDirective : DStmt<OMPExecutableDirective>;
216 def OMPTaskyieldDirective : DStmt<OMPExecutableDirective>;
217 def OMPBarrierDirective : DStmt<OMPExecutableDirective>;
218 def OMPTaskwaitDirective : DStmt<OMPExecutableDirective>;
```

This file generates ...



StmtNodes.inc

```
882 #ifndef OMPEXECUTABLEDIRECTIVE
883 #  define OMPEXECUTABLEDIRECTIVE(Type, Base) STMT(Type, Base)
884 #endif
885 ABSTRACT_STMT(OMPEXECUTABLEDIRECTIVE(OMPExecutableDirective, Stmt))
886 #ifndef OMPATOMICDIRECTIVE
887 #  define OMPATOMICDIRECTIVE(Type, Base) OMPEXECUTABLEDIRECTIVE(Type, Base)
888 #endif
889 OMPATOMICDIRECTIVE(OMPAtomicDirective, OMPExecutableDirective)
890 #undef OMPATOMICDIRECTIVE
891
892 #ifndef OMPBARRIERDIRECTIVE
893 #  define OMPBARRIERDIRECTIVE(Type, Base) OMPEXECUTABLEDIRECTIVE(Type, Base)
894 #endif
895 OMPBARRIERDIRECTIVE(OMPBarrierDirective, OMPExecutableDirective)
896 #undef OMPBARRIERDIRECTIVE
897
```

OpenMPKinds

Now that you understand how Clang uses this technique of inclusion of definition files, let's have a look at the definition file OpenMPKinds.def:

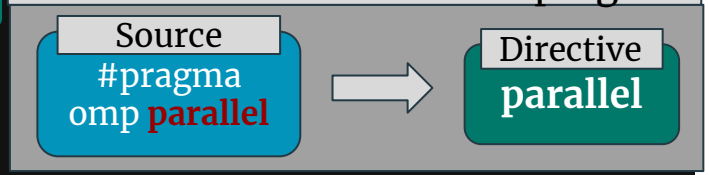
```
175 // OpenMP directives.
176 OPENMP_DIRECTIVE(threadprivate)
177 OPENMP_DIRECTIVE(parallel)
178 OPENMP_DIRECTIVE(task)
179 OPENMP_DIRECTIVE(simd)
180 OPENMP_DIRECTIVE(for)
181 OPENMP_DIRECTIVE(sections)
182 OPENMP_DIRECTIVE(section)
183 OPENMP_DIRECTIVE(single)
184 OPENMP_DIRECTIVE(master)
185 OPENMP_DIRECTIVE(critical)
186 OPENMP_DIRECTIVE(taskyield)
187 OPENMP_DIRECTIVE(barrier)
188 OPENMP_DIRECTIVE(taskwait)
189 OPENMP_DIRECTIVE(taskgroup)
190 OPENMP_DIRECTIVE(flush)
191 OPENMP_DIRECTIVE(ordered)
192 OPENMP_DIRECTIVE(atomic)
193 OPENMP_DIRECTIVE(target)
194 OPENMP_DIRECTIVE(teams)
195 OPENMP_DIRECTIVE(cancel)
196 OPENMP_DIRECTIVE_EXT(target_data, "target data")
197 OPENMP_DIRECTIVE_EXT(target_enter_data, "target enter data")
198 OPENMP_DIRECTIVE_EXT(target_exit_data, "target exit data")
199 OPENMP_DIRECTIVE_EXT(target_parallel, "target parallel")
200 OPENMP_DIRECTIVE_EXT(target_parallel_for, "target parallel for")
201 OPENMP_DIRECTIVE_EXT(target_update, "target update")
```

OpenMPKinds.def

All directives from OpenMP have some entry in this file

```
175 // OpenMP directives.
176 OPENMP_DIRECTIVE(threadprivate)
177 OPENMP_DIRECTIVE(parallel)
178 OPENMP_DIRECTIVE(task)
179 OPENMP_DIRECTIVE(simd)
180 OPENMP_DIRECTIVE(for)
181 OPENMP_DIRECTIVE(sections)
182 OPENMP_DIRECTIVE(section)
183 OPENMP_DIRECTIVE(single)
184 OPENMP_DIRECTIVE(master)
185 OPENMP_DIRECTIVE(critical)
186 OPENMP_DIRECTIVE(taskyield)
187 OPENMP_DIRECTIVE(barrier)
188 OPENMP_DIRECTIVE(taskwait)
189 OPENMP_DIRECTIVE(taskgroup)
```

Directives are created when the **name** matches the token of pragma



Some directives are combined in order to compose larger ones


```
196 OPENMP_DIRECTIVE_EXT(target_data, "target data")
197 OPENMP_DIRECTIVE_EXT(target_enter_data, "target enter data")
198 OPENMP_DIRECTIVE_EXT(target_exit_data, "target exit data")
199 OPENMP_DIRECTIVE_EXT(target_parallel, "target parallel")
200 OPENMP_DIRECTIVE_EXT(target_parallel_for, "target parallel for")
201 OPENMP_DIRECTIVE_EXT(target_update, "target update")
```

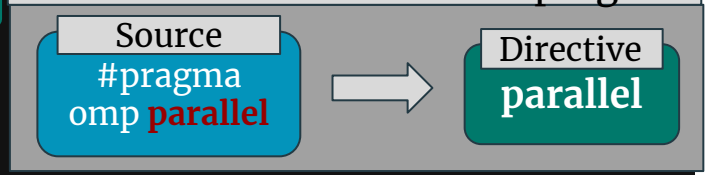
OpenMPKinds.def

All directives from OpenMP have some entry in this file

OpenMPKinds.def

```
175 // OpenMP directives.
176 OPENMP_DIRECTIVE(threadprivate)
177 OPENMP_DIRECTIVE(parallel)
178 OPENMP_DIRECTIVE(task)
179 OPENMP_DIRECTIVE(simd)
180 OPENMP_DIRECTIVE(for)
181 OPENMP_DIRECTIVE(sections)
182 OPENMP_DIRECTIVE(section)
183 OPENMP_DIRECTIVE(single)
184 OPENMP_DIRECTIVE(master)
185 OPENMP_DIRECTIVE(critical)
186 OPENMP_DIRECTIVE(taskyield)
187 OPENMP_DIRECTIVE(barrier)
188 OPENMP_DIRECTIVE(taskwait)
189 OPENMP_DIRECTIVE(taskgroup)
```

Directives are created when the **name** matches the token of pragma



Some directives are combined in order to compose larger ones


```
193 OPENMP_DIRECTIVE(cancel)
196 OPENMP_DIRECTIVE_EXT(target_data, "target data")
197 OPENMP_DIRECTIVE_EXT(target_enter_data, "target enter data")
198 OPENMP_DIRECTIVE_EXT(target_exit_data, "target exit data")
199 OPENMP_DIRECTIVE_EXT(target_parallel, "target parallel")
200 OPENMP_DIRECTIVE_EXT(target_parallel_for, "target parallel for")
201 OPENMP_DIRECTIVE_EXT(target_update, "target update")
```

All directives from OpenMP have some entry in this file

OpenMPKinds.def

All clauses from OpenMP have some entry in this file

```
229 // OpenMP clauses.
230 OPENMP_CLAUSE(if, OMPIfClause)
231 OPENMP_CLAUSE(final, OMPFinalClause)
232 OPENMP_CLAUSE(num_threads, OMPNumThreadsClause)
233 OPENMP_CLAUSE(safelen, OMPSafelenClause)
234 OPENMP_CLAUSE(simdlen, OMPSimdlenClause)
235 OPENMP_CLAUSE(collapse, OMPCollapseClause)
236 OPENMP_CLAUSE(default, OMPDefaultClause)
237 OPENMP_CLAUSE(private, OMPPrivateClause)
238 OPENMP_CLAUSE(firstprivate, OMPFirstprivateClause)
239 OPENMP_CLAUSE(lastprivate, OMPLastprivateClause)
240 OPENMP_CLAUSE(shared,  OMPSharedClause)
241 OPENMP_CLAUSE(reduction,  OMPReductionClause)
242 OPENMP_CLAUSE(linear,  OMPLinearClause)
243 OPENMP_CLAUSE(aligned, OMPAlignedClause)
244 OPENMP_CLAUSE(copyin,  OMPCopyinClause)
245 OPENMP_CLAUSE(copyprivate,  OMPCopyprivateClause)
246 OPENMP_CLAUSE(proc_bind, OMPProcBindClause)
247 OPENMP_CLAUSE(schedule, OMPScheduleClause)
248 OPENMP_CLAUSE(ordered, OMPOrderedClause)
249 OPENMP_CLAUSE(nowait, OMPNowaitClause)
250 OPENMP_CLAUSE(untied, OMPUntiedClause)
251 OPENMP_CLAUSE(mergeable, OMPMergeableClause)
252 OPENMP_CLAUSE(flush, OMPFlushClause)
253 OPENMP_CLAUSE(read, OMPReadClause)
```

OpenMPKinds.def

Each clause must appear once in the macro OPENMP_CLAUSE pairwised with their corresponding class name

These classes are defined in the OpenMPClause.h and they are used a lot throughout clang

```
229 // OpenMP clauses.
230 OPENMP_CLAUSE(if, OMPIfClause)
231 OPENMP_CLAUSE(final, OMPFinalClause)
232 OPENMP_CLAUSE(num_threads, OMPNumThreadsClause)
233 OPENMP_CLAUSE(safelen, OMPSafelenClause)
234 OPENMP_CLAUSE(simdlen, OMPSimdlenClause)
235 OPENMP_CLAUSE(collapse, OMPCollapseClause)
236 OPENMP_CLAUSE(default, OMPDefaultClause)
237 OPENMP_CLAUSE(private, OMPPrivateClause)
238 OPENMP_CLAUSE(firstprivate, OMPFirstprivateClause)
239 OPENMP_CLAUSE(lastprivate, OMPLastprivateClause)
240 OPENMP_CLAUSE(shared,  OMPSharedClause)
241 OPENMP_CLAUSE(reduction,  OMPReductionClause)
242 OPENMP_CLAUSE(linear,  OMPLinearClause)
243 OPENMP_CLAUSE(aligned, OMPAlignedClause)
244 OPENMP_CLAUSE(copyin,  OMPCopyinClause)
245 OPENMP_CLAUSE(copyprivate,  OMPCopyprivateClause)
246 OPENMP_CLAUSE(proc_bind, OMPProcBindClause)
247 OPENMP_CLAUSE(schedule, OMPScheduleClause)
248 OPENMP_C
249 OPENMP_C
250 OPENMP_C
251 OPENMP_CLAUSE(mergeable, OMPMergeableClause)
252 OPENMP_CLAUSE(flush, OMPFlushClause)
253 OPENMP_CLAUSE(read, OMPReadClause)
```

Token id

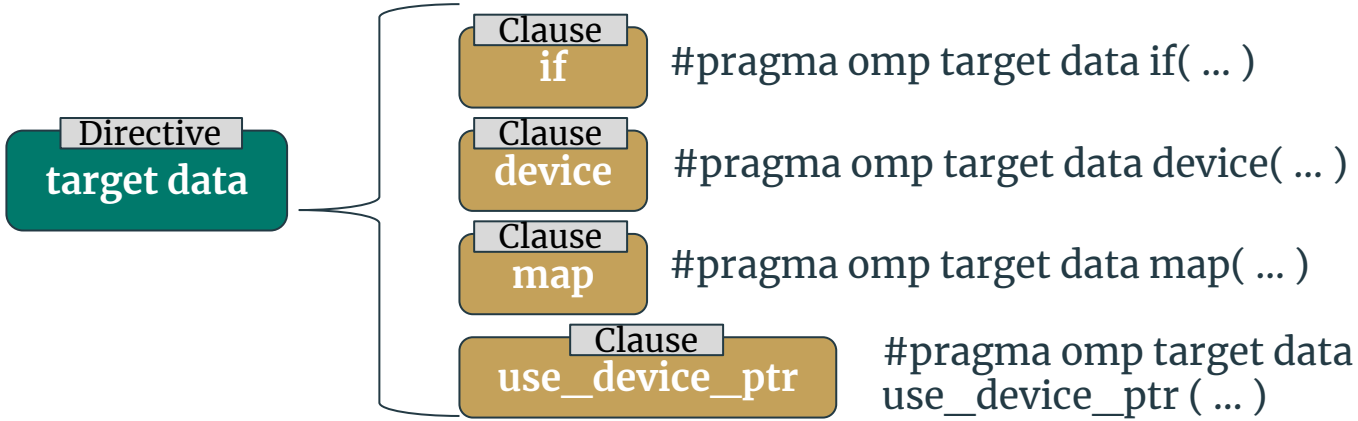
Class Name

OpenMPKinds.def

Each OpenMP directive allows different clauses to follow it. Only some constructs are possible.

```
459 // Clauses allowed for OpenMP directive 'target data'.
460 OPENMP_TARGET_DATA_CLAUSE(if)
461 OPENMP_TARGET_DATA_CLAUSE(device)
462 OPENMP_TARGET_DATA_CLAUSE(map)
463 OPENMP_TARGET_DATA_CLAUSE(use_device_ptr)
```

Each directive must have entries for all allowed clauses



Parse OpenMP as Statements

As important as the definitions in OpenMPKinds, we need to understand how clang uses them for computation.

OpenMP Pragmas are treated as Statements in Clang, so we'll take a look at "How to add a Statement".

How to add an expression or statement

Through these next sections we will follow some instructions from the manuals. Feel free to look at those for a more generic (and detailed) approach:

Internal Manuals: How to add an expression or statement

“Expressions and statements are one of the most fundamental constructs within a compiler, because they interact with many different parts of the AST, semantic analysis, and IR generation. Therefore, adding a new expression or statement kind into Clang requires some care”

How to add an expression or statement

Using the manual's enumerated instructions:

1 "Introduce parsing actions into the parser. Recursive-descent parsing is mostly self-explanatory"

Most likely you won't have to implement any of this section for OpenMP modifications, this is the core of all OpenMP parsing. This is located at:

ParseOpenMP.cpp

```
 912 StmtResult Parser::ParseOpenMPDeclarativeOrExecutableDirective(
 913     AllowedConstructsKind Allowed) {
 914 +--223 lines: assert(Tok.is(tok::annot_pragma_openmp) && "Not an OpenMP directive!");--
1137 }
1138
```

Parser & Semantics Convention

From the clang's internal manuals:

"Historically, the parser used to talk to an abstract Action interface that had virtual methods for parse events, for example ActOnBinOp().
When Clang grew C++ support, the parser stopped supporting general Action clients – it now always talks to the Sema library.

However, the Parser still accesses AST objects only through opaque types like ExprResult and StmtResult. Only Sema looks at the AST node contents of these wrappers."

Parser & Semantics Convention

From the clang's internal manuals:

"Historically, the parser used to talk to [illegible]tract Action interface that had virtual methods for p[illegible]rse event[illegible] ActOnBinOp().
When Clang grew C++ support, the par[illegible]r stopped general Action clients – it now always t[illegible]lks to the

However, the Parser still accesses AST [illegible]bjects only through opaque types like ExprResult and [illegible]Only Sema looks at the AST node contents of these wrappers.

StmtResult

StmtResult is the same as ActionResult <Stmt * >

This is the structure that stores OpenMP actions

How to add an expression or statement

Using the manual's enumerated instructions:

1 "Introduce parsing actions into the parser. Recursive-descent parsing is mostly self-explanatory"

Most likely you won't have to implement any of this section for OpenMP modifications, this is the core of all OpenMP parsing. This is located at:

ParseOpenMP.cpp

```
 912 StmtResult Parser::ParseOpenMPDeclarativeOrExecutableDirective(
 913     AllowedConstructsKind Allowed) {
 914 +--223 lines: assert(Tok.is(tok::annot_pragma_openmp) && "Not an OpenMP directive!");--
1137 }
1138
```

How to add an expression or statement

Using the manual's enumerated instructions:

1. "Introduce parsing actions into the parser. [illegible] parsing is mostly self-explanatory"

Mo[illegible] Op[illegible]nMP modifications, this is the core of all Op[illegible] Th[illegible] is located at:

ParseOpenMP.cpp

StmtResult is the same as ActionResult <Stmt * >

```
912  StmtResult Parser::ParseOpenMPDeclarativeOrExecutableDirective(
913      AllowedConstructsKind Allowed) {
914  +--223 lines: assert(Tok.is(tok::annot_pragma_openmp) && "Not an OpenMP directive!");--
1137 }
1138
```

How to add an expression or statement

Using the manual's enumerated instructions:

2. "Introduce semantic analysis actions into Sema". There are functions ActOnXXX and BuildXXX that will (eventually) build the AST node. Please note that there aren't OpenMP Build functions for directives, we should add only the ActOnXXX for each directive.

Sema.h

This is where the signatures are located

```
8873  /// Called on well-formed '\#pragma omp atomic' after parsing of the
8874  /// associated statement.
8875  StmtResult ActOnOpenMPAtomicDirective(ArrayRef<OMPClause *> Clauses,
8876                                        Stmt *AStmt, SourceLocation StartLoc,
8877                                        SourceLocation EndLoc);
8878  /// Called on well-formed '\#pragma omp target' after parsing of the
8879  /// associated statement.
8880  StmtResult ActOnOpenMPTargetDirective(ArrayRef<OMPClause *> Clauses,
8881                                        Stmt *AStmt, SourceLocation StartLoc,
8882                                        SourceLocation EndLoc);
8883  /// Called on well-formed '\#pragma omp target data' after parsing of
8884  /// the associated statement.
8885  StmtResult ActOnOpenMPTargetDataDirective(ArrayRef<OMPClause *> Clauses,
8886                                            Stmt *AStmt, SourceLocation StartLoc,
8887                                            SourceLocation EndLoc);
```

SemaOpenMP.cpp

And here is the body of those functions

```
6787
6788 StmtResult Sema::ActOnOpenMPTargetDataDirective(ArrayRef<OMPClause *> Clauses,
6789                                                 Stmt *AStmt,
6790                                                 SourceLocation StartLoc,
6791                                                 SourceLocation EndLoc) {
6792 +-- 18 lines: if (!AStmt)----------------------------------------------------
6810 }
6811
6812 StmtResult
6813 Sema::ActOnOpenMPTargetEnterDataDirective(ArrayRef<OMPClause *> Clauses,
6814                                           SourceLocation StartLoc,
6815                                           SourceLocation EndLoc, Stmt *AStmt) {
6816 +-- 31 lines: if (!AStmt)----------------------------------------------------
6847 }
6848
```

How to add an expression or statement

Using the manual's enumerated instructions:

3 Introduce an AST node for our new statement. This starts with declaring the node in include/Basic/StmtNodes.td

It's also important to create the correspondent class for our statement in include/AST/StmtOpenMP.h header

StmtNodes.td

```
200 // OpenMP Directives.
201 def OMPExecutableDirective : Stmt<1>;
202 def OMPLoopDirective : DStmt<OMPExecutableDirective, 1>;
203 def OMPParallelDirective : DStmt<OMPExecutableDirective>;
204 def OMPSimdDirective : DStmt<OMPLoopDirective>;
205 def OMPForDirective : DStmt<OMPLoopDirective>;
206 def OMPForSimdDirective : DStmt<OMPLoopDirective>;
207 def OMPSectionsDirective : DStmt<OMPExecutableDirective>;
208 def OMPSectionDirective : DStmt<OMPExecutableDirective>;
209 def OMPSingleDirective : DStmt<OMPExecutableDirective>;
210 def OMPMasterDirective : DStmt<OMPExecutableDirective>;
211 def OMPCriticalDirective : DStmt<OMPExecutableDirective>;
212 def OMPParallelForDirective : DStmt<OMPLoopDirective>;
```

This file holds the definitions of Stmts and Stmt inheritance

StmtNodes.td

```
201 def OMPExecutableDirective : Stmt<1>;
```

OMPExecutableDirective is a class that inherits from Stmt

StmtOpenMP.h

```
 33 class OMPExecutableDirective : public Stmt {
 34 +--233 lines: friend class ASTStmtReader;-------------
267 };
```

Every entry needs its correspondent implementation somewhere. In the case of OpenMP Statements, they are at StmtOpenMP.h and StmtOpenMP.cpp

StmtNodes.td

```
202 def OMPLoopDirective : DStmt<OMPExecutableDirective, 1>;
203 def OMPParallelDirective : DStmt<OMPExecutableDirective>;

214 def OMPParallelSectionsDirective : DStmt<OMPExecutableDirective>;
215 def OMPTaskDirective : DStmt<OMPExecutableDirective>;
216 def OMPTaskyieldDirective : DStmt<OMPExecutableDirective>;
217 def OMPBarrierDirective : DStmt<OMPExecutableDirective>;
218 def OMPTaskwaitDirective : DStmt<OMPExecutableDirective>;
```

All OpenMP Stmts inherit from OMPExecutableDirective. Some are direct children, others inherit through some more derivations.

```
202 def OMPLoopDirective : DStmt<OMPExecutableDirective, 1>;

212 def OMPParallelForDirective : DStmt<OMPLoopDirective>;
```

How to add an expression or statement

3 Before sending your code to production, there are “some specific things to watch for” the manual highlights

Those have to do with IDE support, like implementing printing support for your statement in StmtPrinter.cpp

Feel free to look at the manual section if you’d like to implement those

How to add an expression or statement

4 “Teach semantic analysis to build your AST node. At this point, you can wire up your Sema::BuildXXX function to actually create your AST”

In most cases this step isn’t necessary, as OpenMP’s infrastructure already handles this. Feel free to look at the manual if you’d like to modify those.

How to add an expression or statement

5 “Teach code generation to create IR to your AST node. This step is the first (and only) that requires knowledge of LLVM IR”

It’s strongly recommended that you read the manual section for this one. This step really depends on the behaviour you want to achieve.

Here are some useful advices:

Code Generation

It’s super important to be able to verify if the IR generation is happening correctly. Using the clang you built, compile some “C/C++” code using the flag “-emit-llvm”, this will output the IR of such code (IR’s default file name extension is “.ll”)

The IR is the intermediate representation for the machine code, that’s what will actually be executed. So to prove if your new clang is really generating the right code for your pragmas, examine the IR generated for your study case.

IR is usually complicated to read. Maybe it’s a good idea to work at first glance using comparisons. Create some test cases for you, see how pure OpenMP would generate such IR, how is it sending computation to cores, to devices?

Code Generation

If there is something bugging you, there's a chance what you are looking for is in one of these files:

CGStmtOpenMP.cpp
CGOpenMPRuntime.h
CGOpenMPRuntime.cpp
CodeGenFunction.h

Good luck

How to add an expression or statement

6 “Teach template instantiation how to cope with your AST node”

TreeTransform.h is a file that “implements a semantic tree transformation that takes a given AST and rebuilds it, possibly transforming some nodes in the process”.

This is about type checking and type conversion (through derivation of types). Maybe it’s important for your application to implement a tree transformation. All directives have a transformation implemented in this file:

TreeTransform.h

```
7653 //===----------------------------------------------------------------------===//
7654 // OpenMP directive transformation
7655 //===----------------------------------------------------------------------===//
7656 template <typename Derived>
7657 StmtResult TreeTransform<Derived>::TransformOMPExecutableDirective(
7658     OMPExecutableDirective *D) {
7659 +-- 53 lines: Transform the clauses-----------------------------------------
7712 }
7713
7714 template <typename Derived>
7715 StmtResult
7716 TreeTransform<Derived>::TransformOMPParallelDirective(OMPParallelDirective *D) {
7717 +--  6 lines: DeclarationNameInfo DirName;---------------------------------
7723 }
7724
```

How to add an expression or statement

7 “There are some “extras” that make other features work better”

These don’t apply to us. OpenMP’s implementation does not support code completion and other “extras” that this section describes.

Hope this material has helped you in some way!

Have a great time using pragmas in clang