# 使用Fortran调用Graham Markall编写的CUDA\_C\_PCG程序

## 求解大型线性方程组

参考：Yousef Saad. Iterative methods for sparse linear system

算法较容易理解，但落实到计算机编程，特别是并行化，难度还是很大的。

近年来，CPU性能的挖掘越来越依赖于SIMD指令(SSE, AVX)

还有复杂的架构，CUDA、Xeon Phi

## 求解器总结

常见的，用于求解常微分方程的求解器：SUNDIALS\_CVODE

用于求解偏微分方程的求解器：

（1）直接求解算法

MUMPS

（2）迭代求解算法

### a 单机，纯CPU，单线程（部分操作多线程）

Eigen

Blaze (blaze-lib/blaze-Bitbucket)

Armadillo (C++ linear algebra library)

Intel MKL

### b 单机，混合CPU/GPU, 多线程

MAGMA: CUDA+CPU, openCL, Xeon phi

ViennaCL (Linear algebra Library using CUDA, OpenCL and OpenMP)，与MAGMA类似

Paralution (MPI的是商业程序，收费)

ArrayFire

AmgX (Nvidia) (MPI等都是免费，开源代码)

### c 集群，MPI

PETSc (C)

Trilinos (C++)

Hypre (C)

## CUDA\_JCG (最简单的CUDA\_JCG)

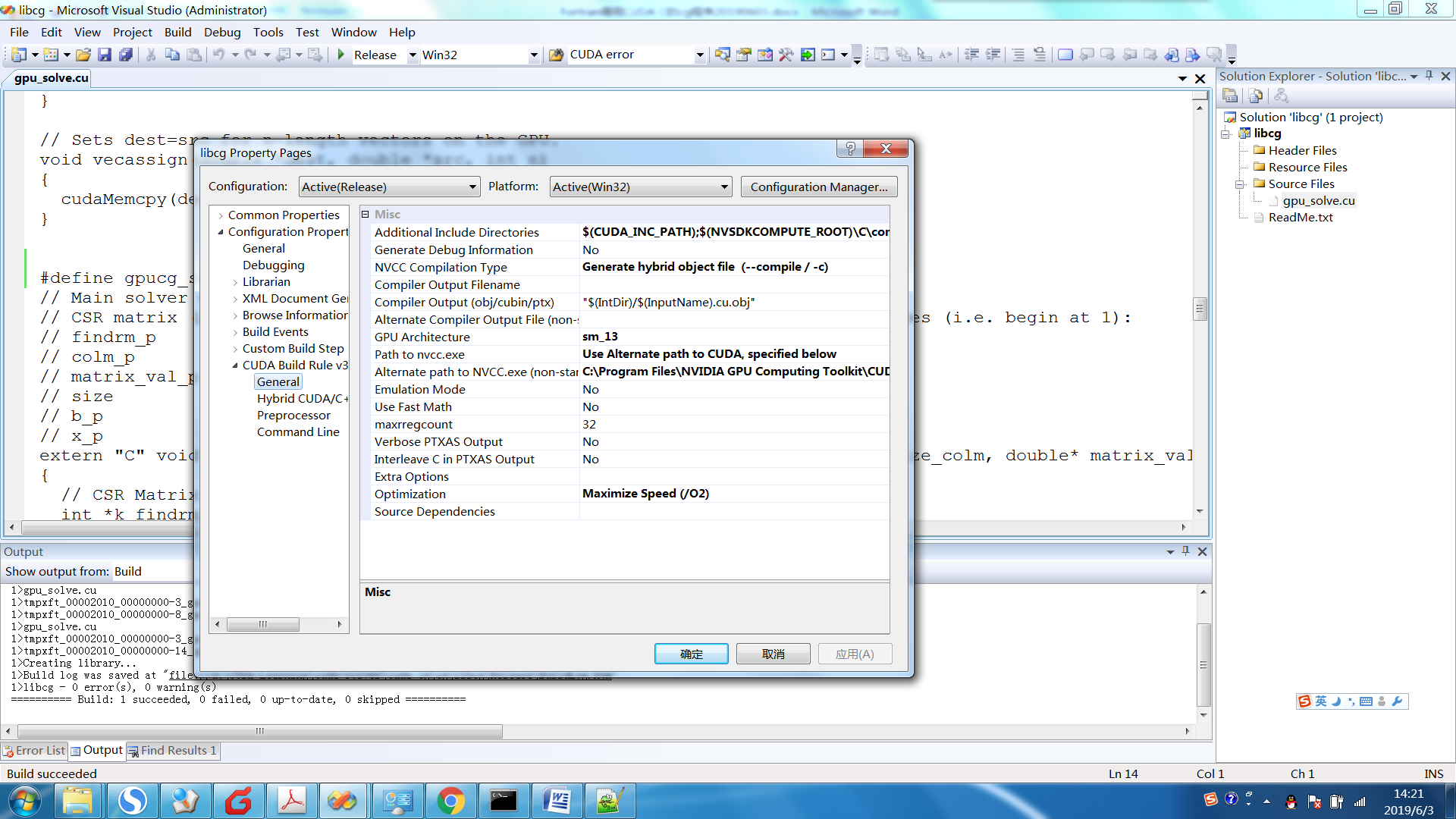
Graham Markall (2009)编写的CUDA C版本的PCG程序，是采用Jacobi预处理的CG法，原本是用于加速求解FLUIDITY模型（CFD）中离散的大型对称正定稀疏矩阵的子程序。采用CSR格式存储稀疏系数矩阵，然后可以被C或CPP语言的CFD代码调用。

因此，一般的海洋动力学模型（FVCOM和SCHISM模型）都是FORTRAN语言编写，本研究成功在FORTRAN程序中调用CUDA\_C\_PCG，编译过程记录如下：

（1）安装赵开勇博士开发的CUDA\_VS\_WIZARD程序；

（2）安装CUDA\_4.0的toolkit；

（3）编译PCG的lib文件：



GPU架构只能选择最高的sm\_13

指定nvcc的路径

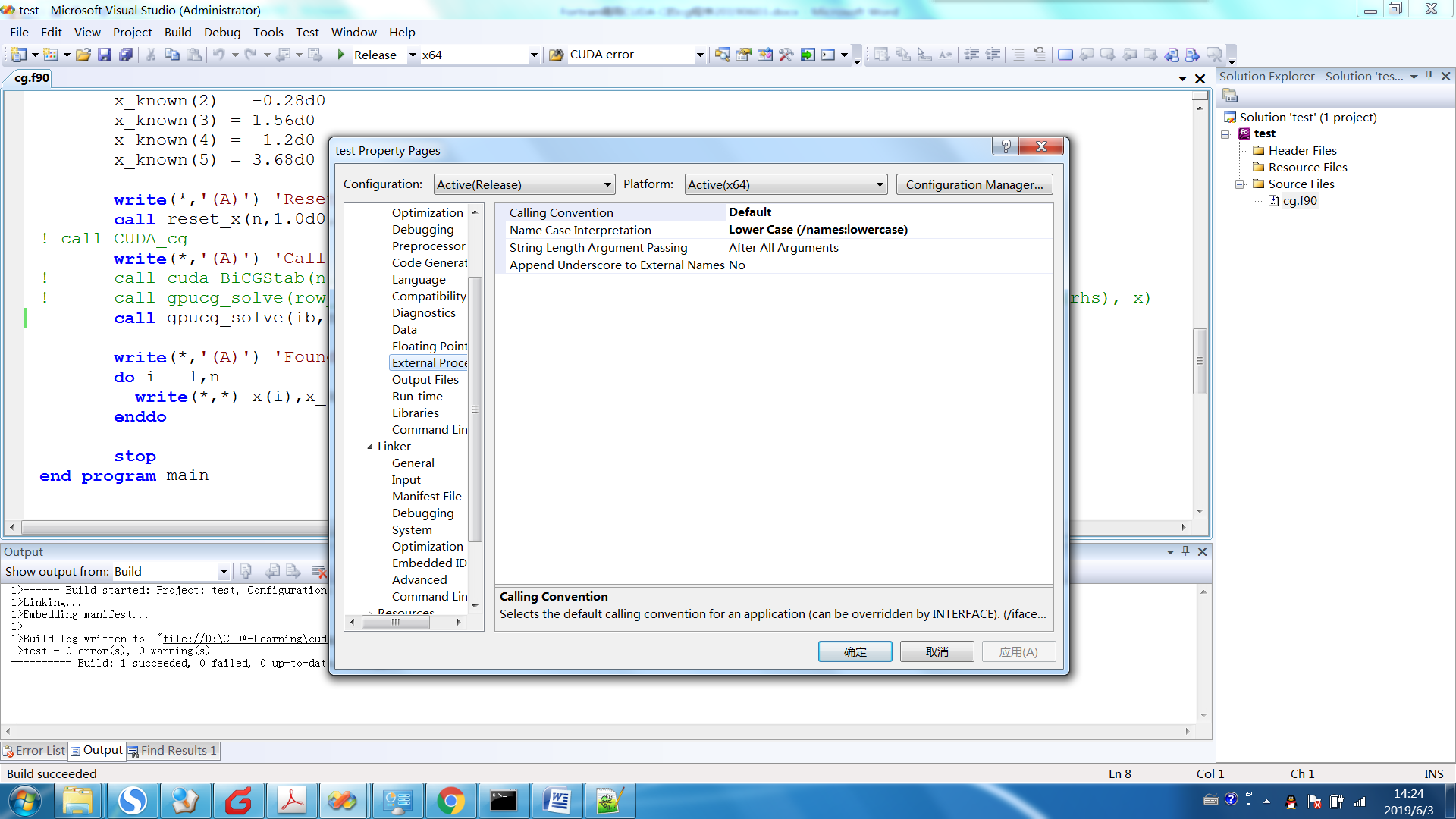
runtime library一定要选择 /MD

（4）调用lib程序：

配置好CUDA\_4.0的include和lib64路径；以及，刚才编译的lib文件路径

INPUT： cuda.lib cudart.lib libcg.lib

注意：External Procedures中设置：Name case interpretation: Lower Case



## CUDA\_BiCGStab

Fortran语言编写

## ViennaCL

Examples 5: Iterative Solvers

ViennaCL provides different iterative solvers for various classes of matrices and is not restricted to built-in types. Since the interface of ViennaCL types is compatible with [uBLAS](http://www.boost.org/doc/libs/release/libs/numeric/) objects, the iterative solvers can directly be called with [uBLAS](http://www.boost.org/doc/libs/release/libs/numeric/) objects. ViennaCL 1.0.x provides the following iterative solvers:

Conjugate Gradient (*viennacl::linalg::cg\_tag*)

Stabilized Bi-Conjugate Gradient (*viennacl::linalg::bicgstab\_tag*)

Generalized Minimum Residual (*viennacl::linalg::gmres\_tag*)

An optional incomplete LU factorization with threshold can be used as preconditioner.

Iterative solvers in ViennaCL

**typedef** **float** ScalarType;

//typedef double ScalarType; //use this if your GPU supports double precision

// Set up some ublas objects:

ublas::vector<ScalarType> ublas\_rhs;

ublas::vector<ScalarType> ublas\_result;

ublas::compressed\_matrix<ScalarType> ublas\_matrix;

// Set up some ViennaCL objects:

viennacl::vector<ScalarType> vcl\_rhs;

viennacl::vector<ScalarType> vcl\_result;

viennacl::compressed\_matrix<ScalarType> vcl\_matrix;

*/\* Initialize and fill all objects here \*/*

//

// Compute ILUT preconditioners for CPU and for GPU objects:

//

viennacl::linalg::ilut\_tag ilut\_conf(10, 1e-5); //10 entries, rel. tol. 1e-5

**typedef** viennacl::linalg::ilut\_precond<

ublas::compressed\_matrix<ScalarType> > ublas\_ilut\_t;

//preconditioner for ublas objects:

ublas\_ilut\_t ublas\_ilut(ublas\_matrix, ilut\_conf);

viennacl::linalg::ilut\_precond<

viennacl::compressed\_matrix<ScalarType> > vcl\_ilut\_t;

//preconditioner for ViennaCL objects:

vcl\_ilut\_t vcl\_ilut(vcl\_matrix, ilut\_conf);

//

// Conjugate gradient solver without preconditioner:

//

ublas\_result = solve(ublas\_matrix, //using ublas objects on CPU

ublas\_rhs,

viennacl::linalg::cg\_tag());

vcl\_result = solve(vcl\_matrix, //using viennacl objects on GPU

vcl\_rhs,

viennacl::linalg::cg\_tag());

//

// Conjugate gradient solver using ILUT preconditioner

//

ublas\_result = solve(ublas\_matrix, //using ublas objects on CPU

ublas\_rhs,

viennacl::linalg::cg\_tag(),

ublas\_ilut);

vcl\_result = solve(vcl\_matrix, //using viennacl objects on GPU

vcl\_rhs,

viennacl::linalg::cg\_tag(),

vcl\_ilut);

// for BiCGStab and GMRES, use the solver tags

// viennacl::linalg::bicgstab\_tag and viennacl::linalg::gmres\_tag

// instead of viennacl::linalg::cg\_tag in the calls above.