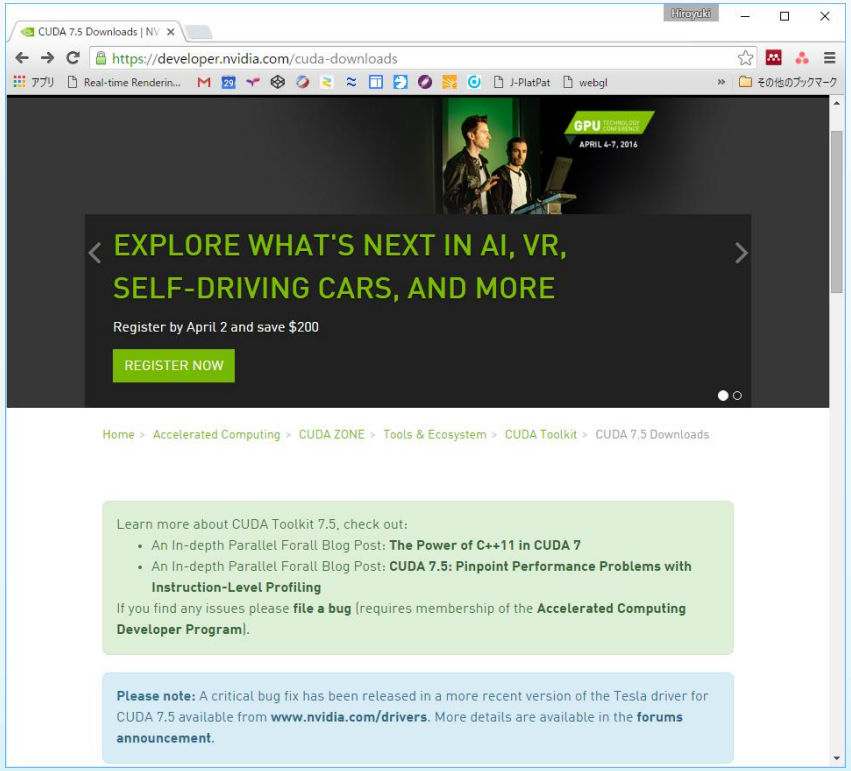
Introduction to CUDA 7.5 on Windows
とりあえず動かす編

2016/05/26

Hiroyuki Kubo

CUDA SDKのインストール

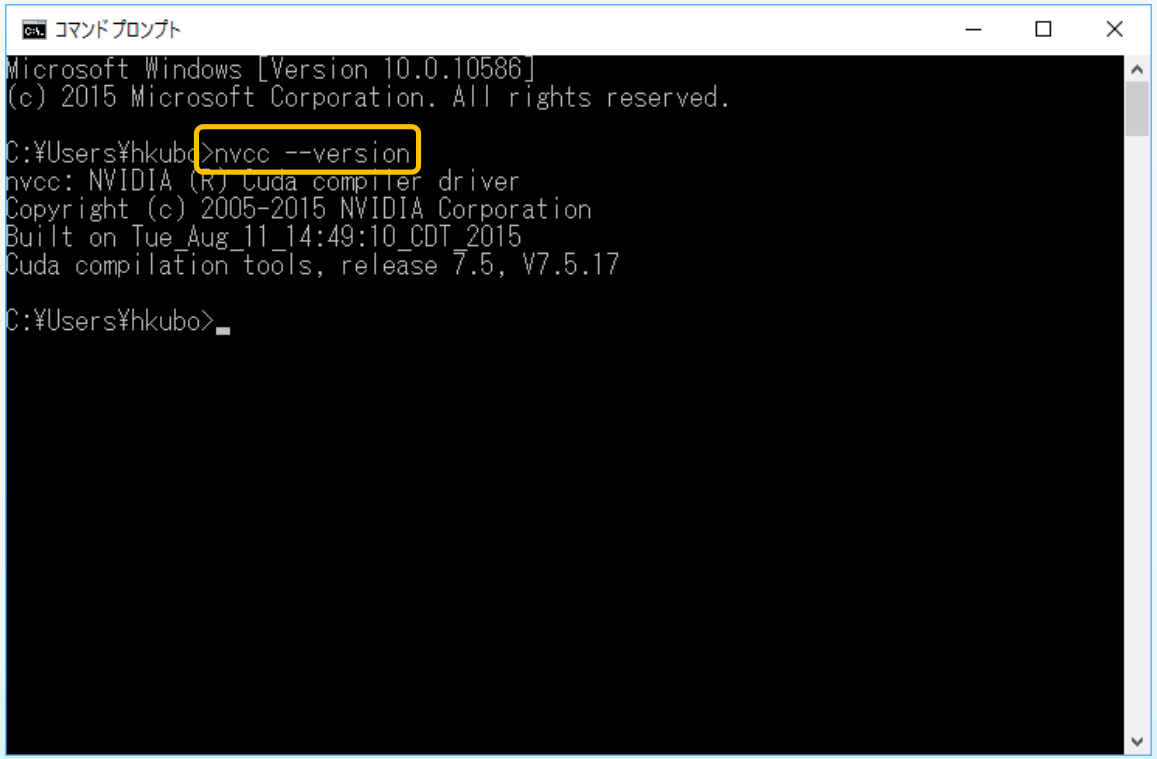
- CUDA ダウンロードサイト (https://developer.nvidia.com/cuda-downloads)から インストーラをダウンロードする
- インストーラを起動し，インストール．（『はい』を押し続ける．）
- この資料はCUDA7.5を前提としているがたぶん6.5でも動くと思います．



インストールの確認

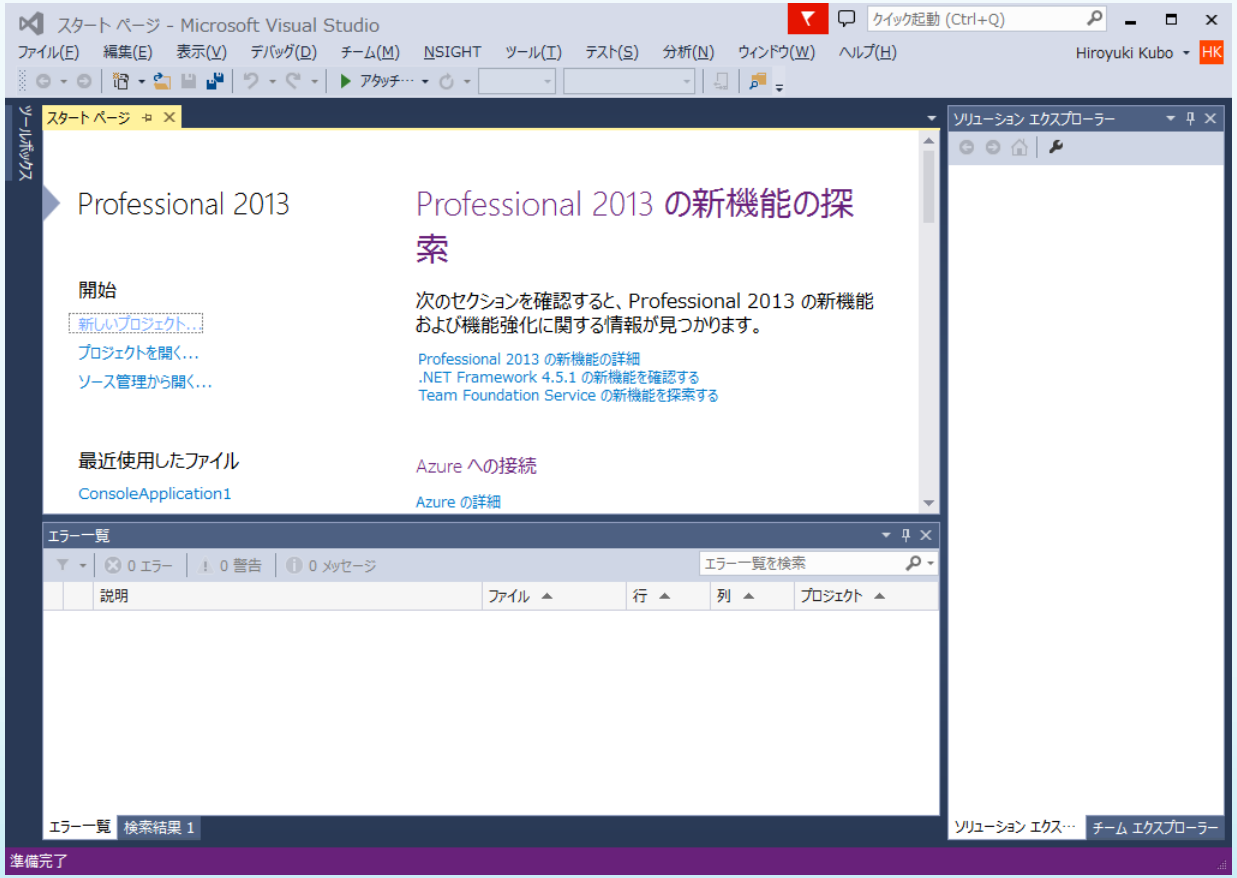
- コマンドプロンプトを起動して
  “nvcc --version“
  とタイプする.

- バージョン情報が表示されたらOK

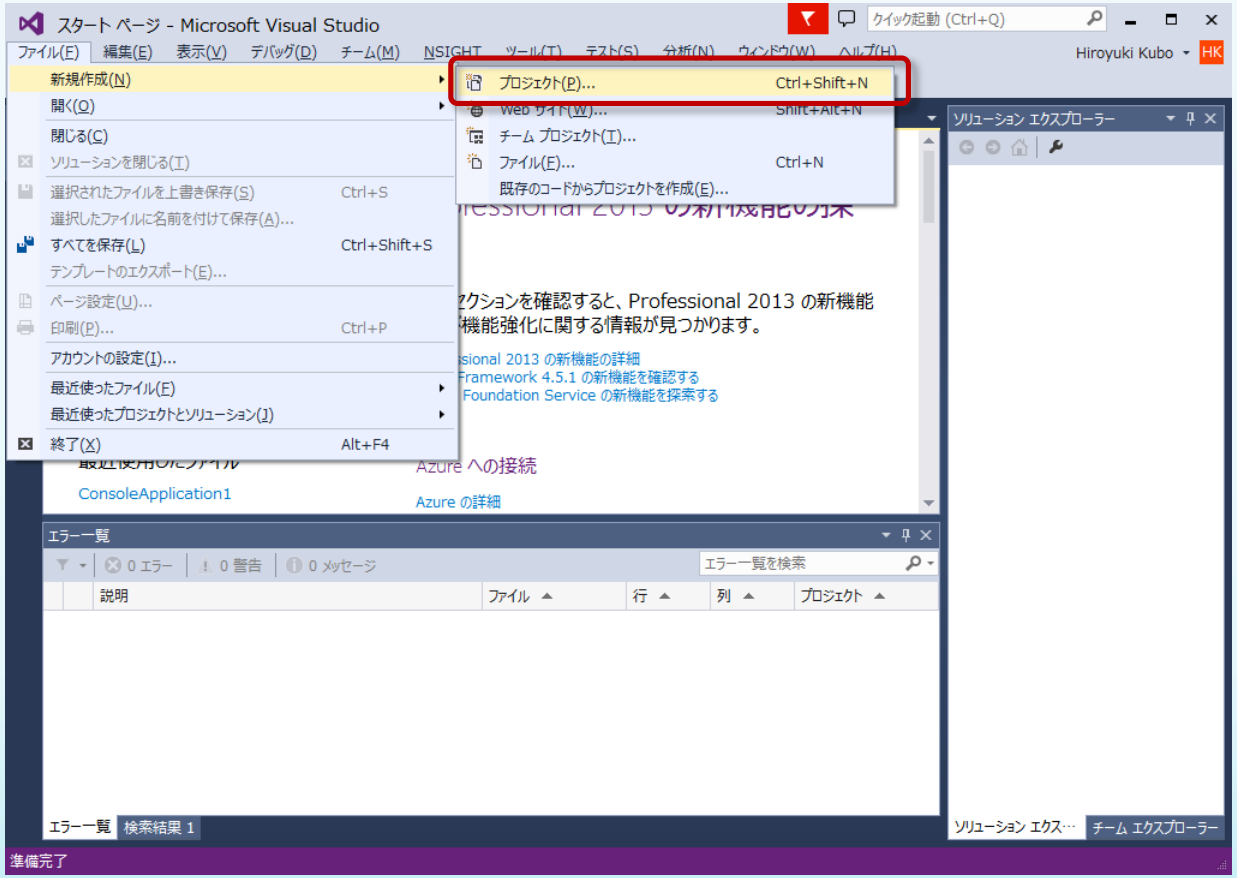


Visual Studio2013でCUDA

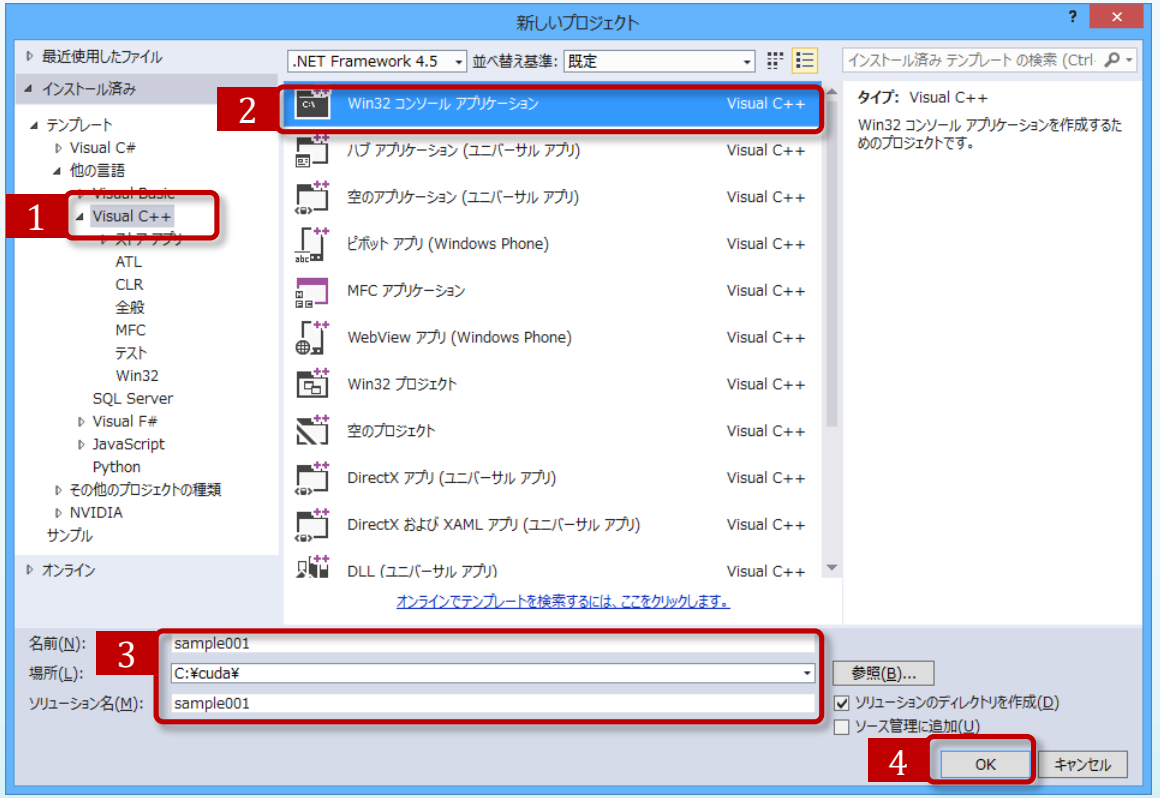
- Visual Studio2013を起動する



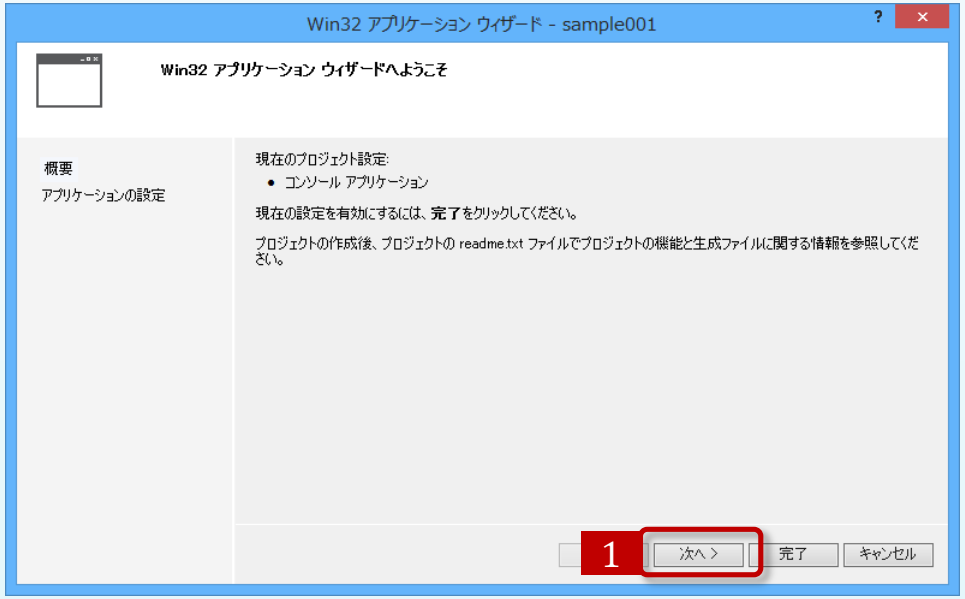
- ファイル-新規作成-プロジェクト



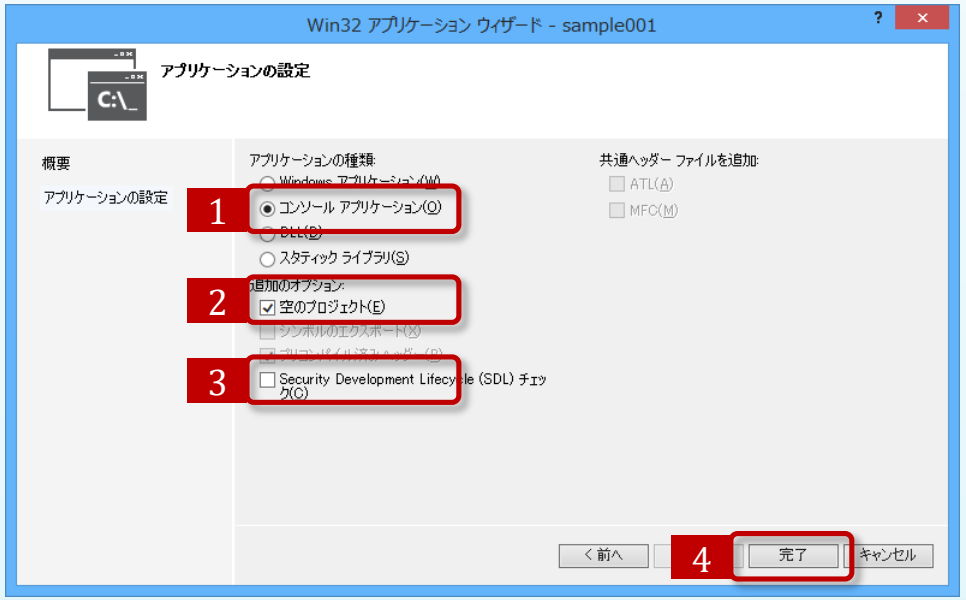
1. テンプレートの中から”Visual C++”を選択する
2. Win32コンソールアプリケーションを選択
3. 保存場所，名前，ソリューション名を適宜設定する
4. OKをクリック

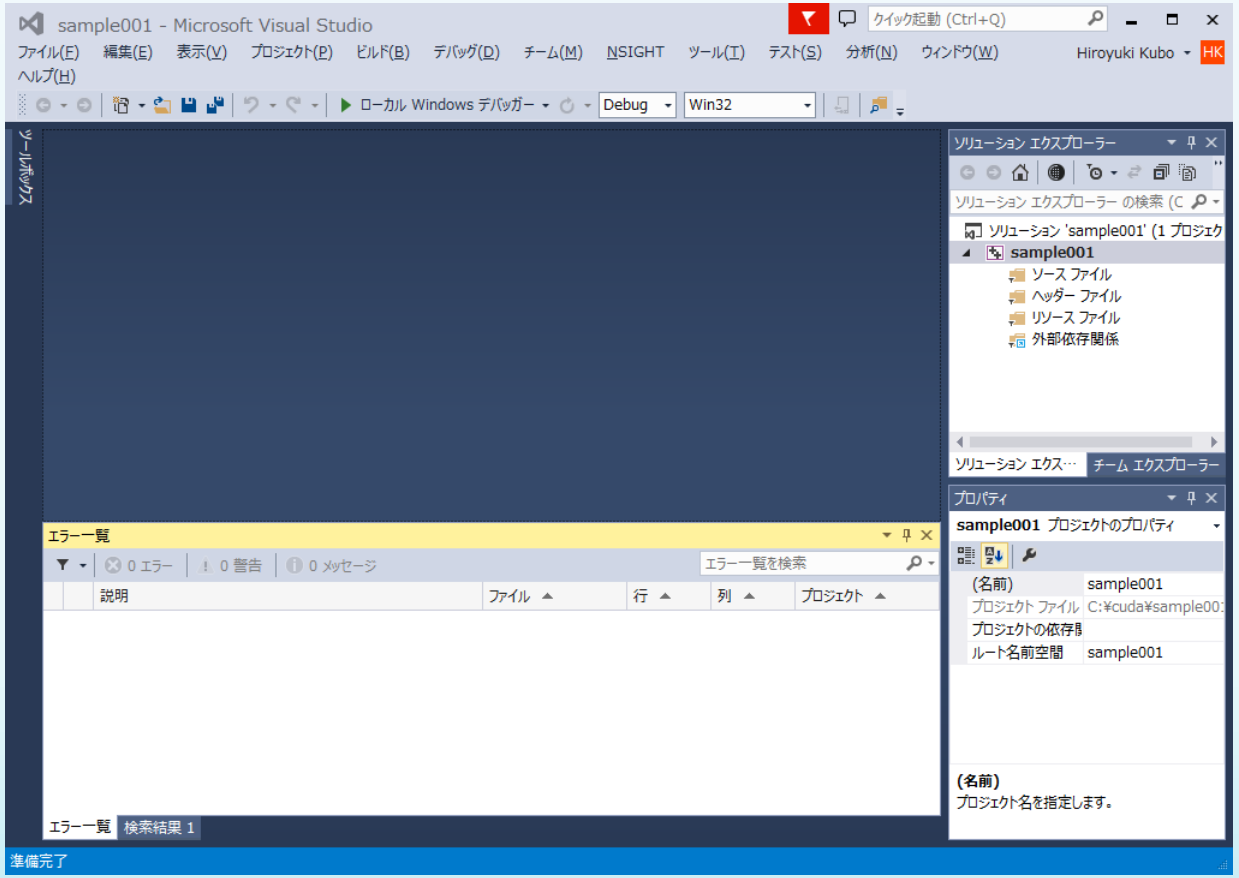


1. “次へ”をクリック


Win32 アプリケーション ウィザード - sample001
Win32 アプリケーション ウィザードへようこそ
概要
アプリケーションの設定
現在のプロジェクト設定:
コンソール アプリケーション
現在の設定を有効にするには、完了をクリックしてください。
プロジェクトの作成後、プロジェクトの readme.txt ファイルでプロジェクトの機能と生成ファイルに関する情報を参照してください。
1
次へ >
完了
キャンセル

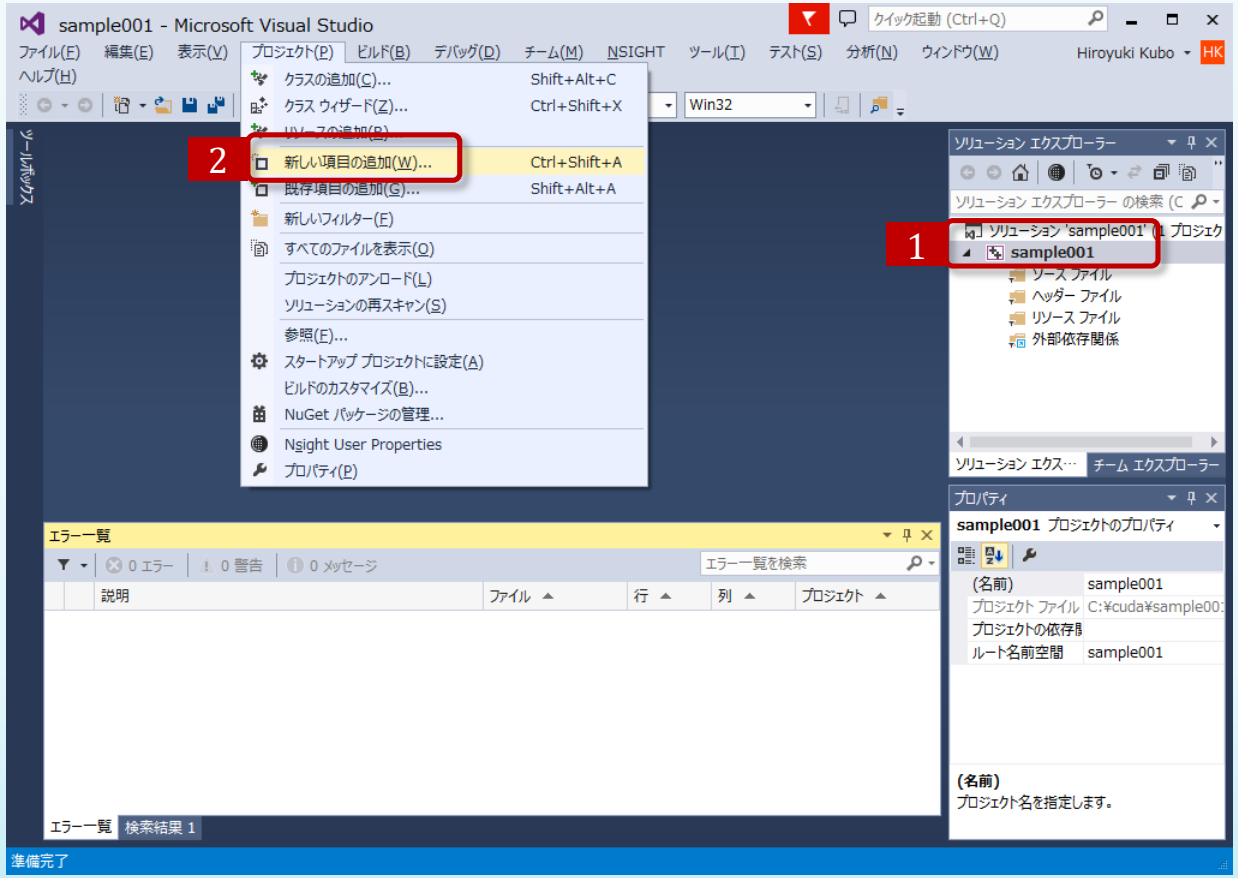
1. コンソールアプリケーションを選択
2. 空のプロジェクトにチェック
3. SDLのチェックを外す
4. 完了をクリック



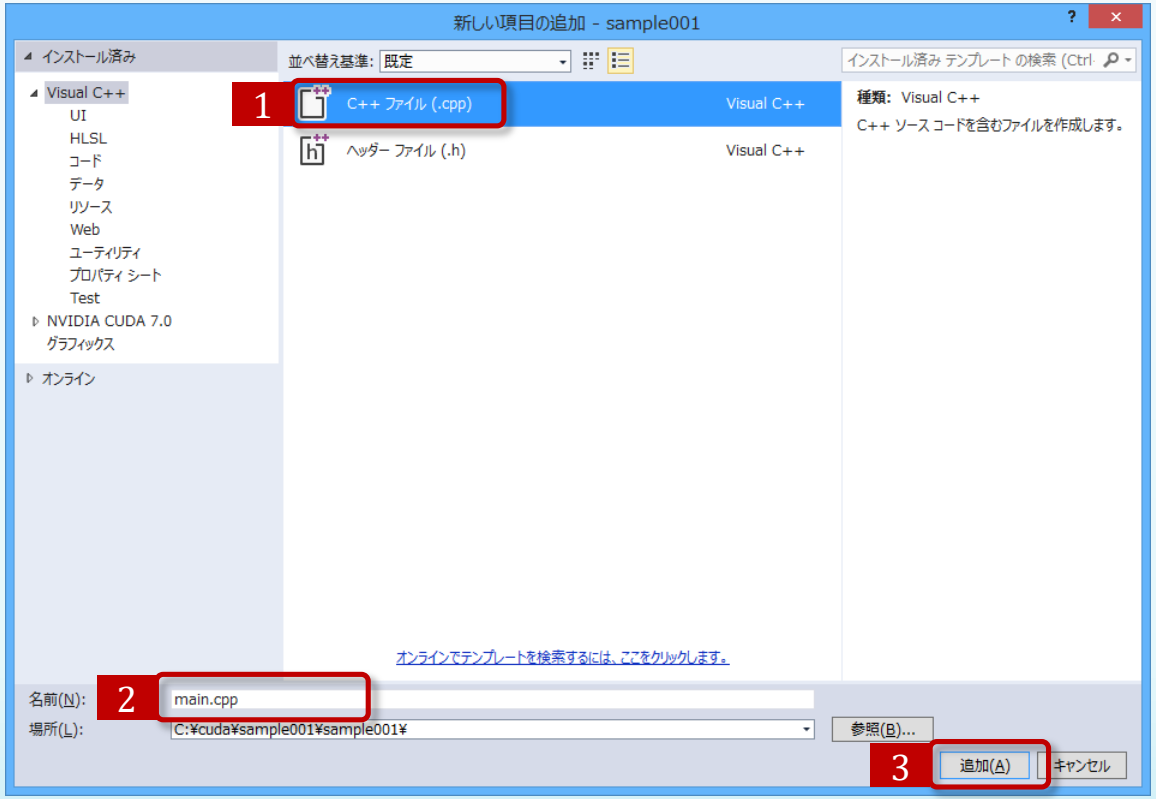

sample001 - Microsoft Visual Studio
ファイル(F) 編集(E) 表示(V) プロジェクト(P) ビルド(B) デバッグ(D) チーム(M) NSIGHT ツール(T) テスト(S) 分析(N) ウィンドウ(W) ヘルプ(H)
Hiroyuki Kubo
ローカル Windows デバッガー
Debug
Win32
ツールボックス
ソリューション エクスプローラー
ソリューション 'sample001' (1 プロジェク
sample001
ソース ファイル
ヘッダー ファイル
リソース ファイル
外部依存関係
ソリューション エクス…
チーム エクスプローラー
プロパティ
sample001 プロジェクトのプロパティ
(名前) sample001
ルート名前空間 sample001
(名前)
プロジェクト名を指定します。
エラー一覧
0 エラー
0 警告
0 メッセージ
エラー一覧を検索
説明
ファイル
行
列
プロジェクト
エラー一覧
検索結果 1
準備完了

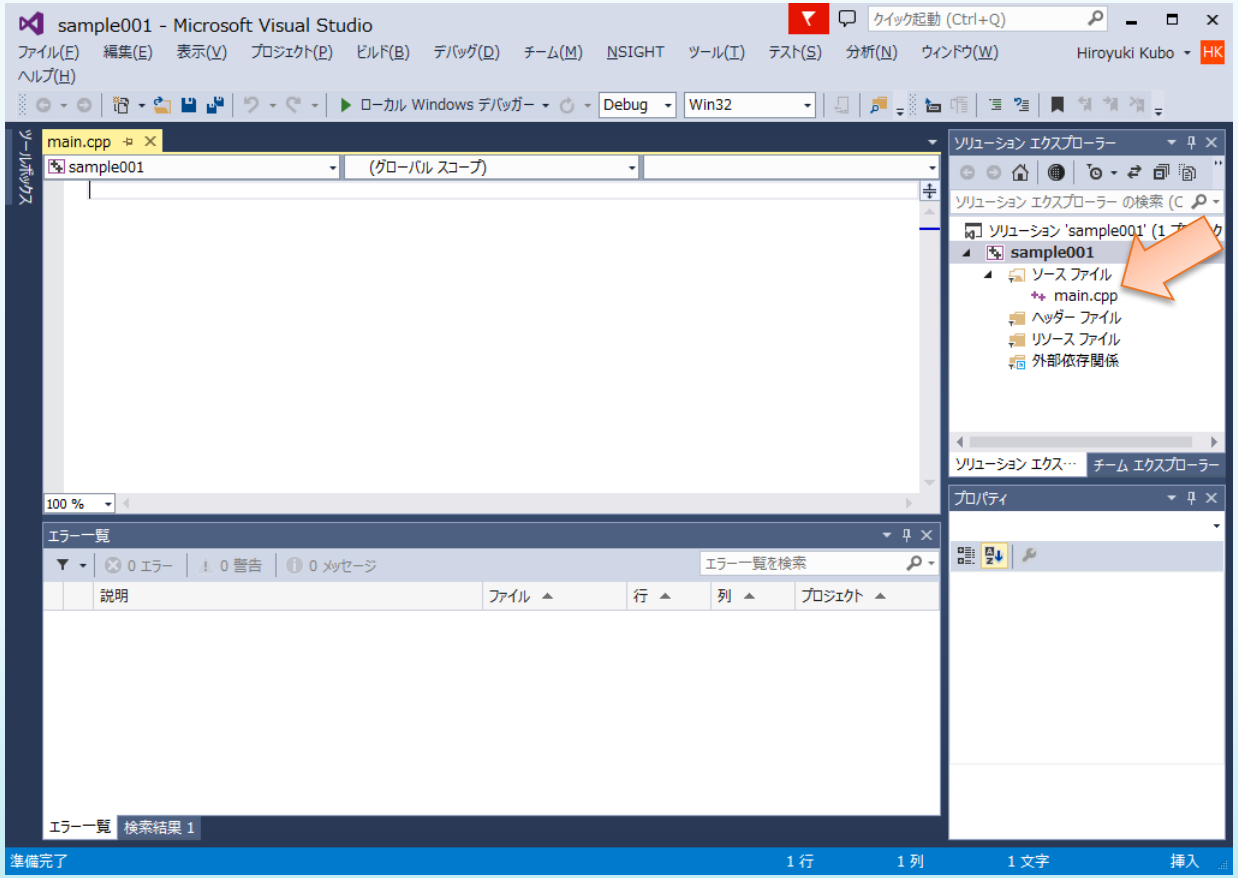
プロジェクトができた

1. プロジェクトをクリックして選択した状態で・・
2. プロジェクトー新しい項目の追加をクリック



1. C++を選択
2. ファイル名を適宜指定
3. “追加”をクリック

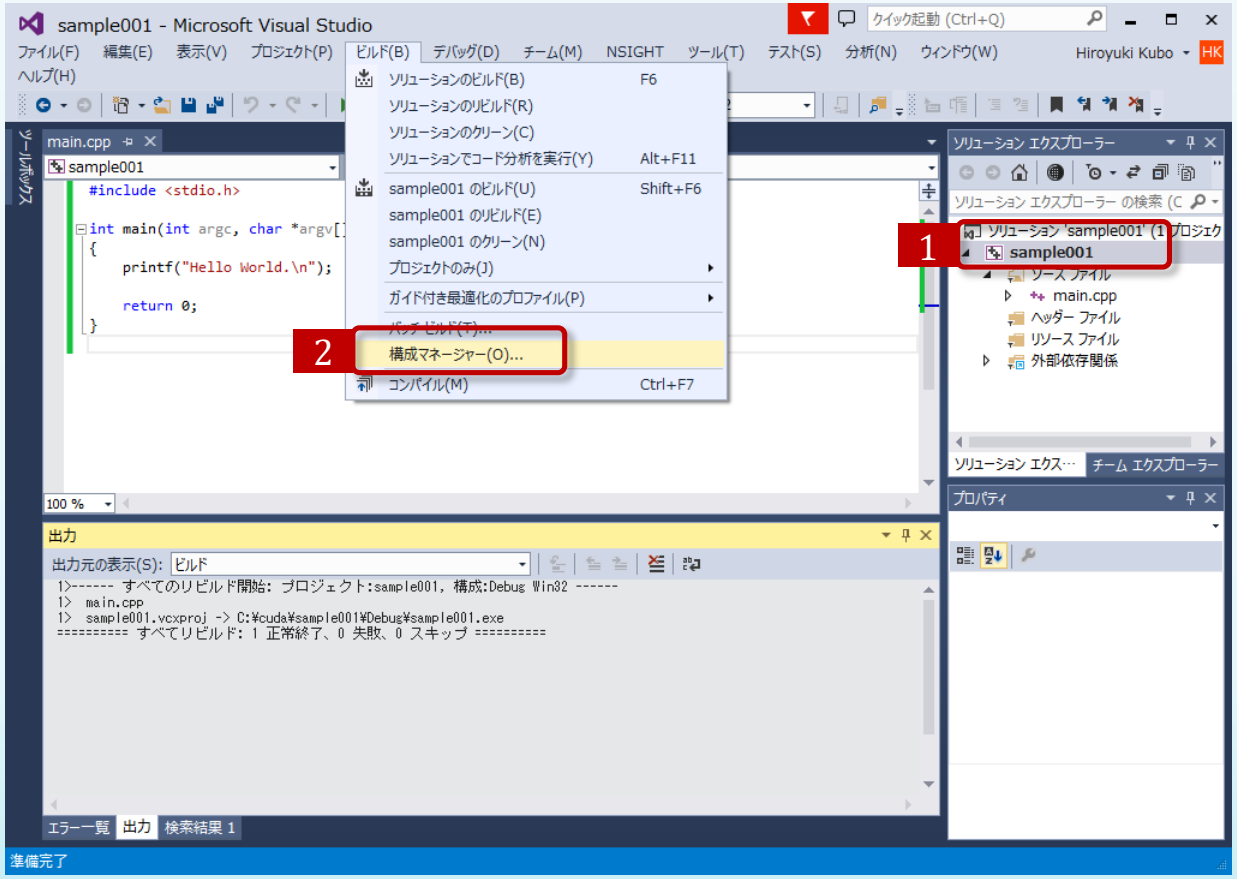



sample001 - Microsoft Visual Studio
ファイル(F) 編集(E) 表示(V) プロジェクト(P) ビルド(B) デバッグ(D) チーム(M) NSIGHT ツール(T) テスト(S) 分析(N) ウィンドウ(W)
ヘルプ(H)
Hiroyuki Kubo
ローカル Windows デバッガー
Debug
Win32
main.cpp
sample001
(グローバル スコープ)
100 %
ソリューション エクスプローラー
ソリューション 'sample001' (1
sample001
ソース ファイル
main.cpp
ヘッダー ファイル
リソース ファイル
外部依存関係
チーム エクスプローラー
プロパティ
エラー一覧
0 エラー
0 警告
0 メッセージ
説明
ファイル
行
列
プロジェクト
検索結果 1
準備完了
1 行
1 列
1 文字
挿入

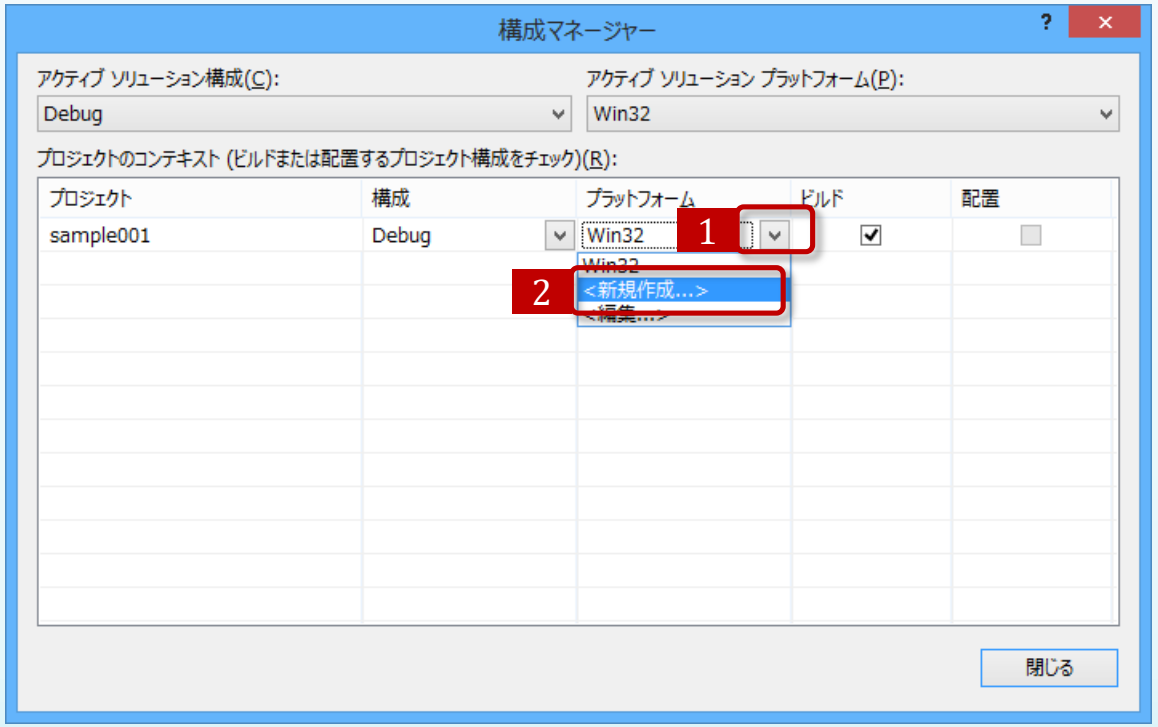
追加された！！

64bit ビルドに変更

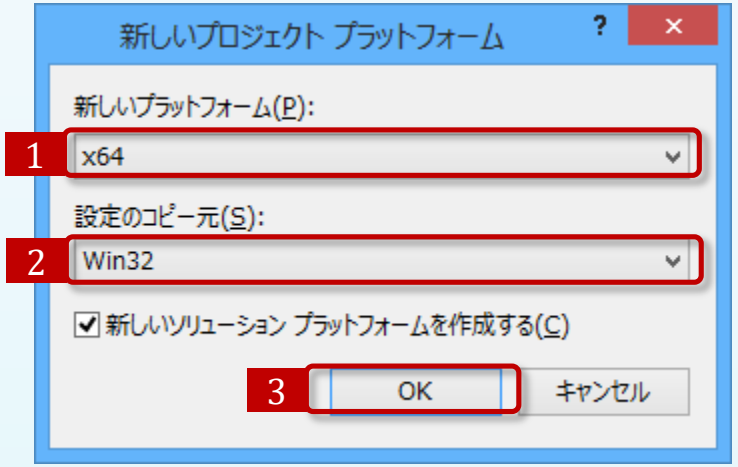
1. プロジェクトが選択された状態で・・・
2. ビルドー構成マネージャをクリック



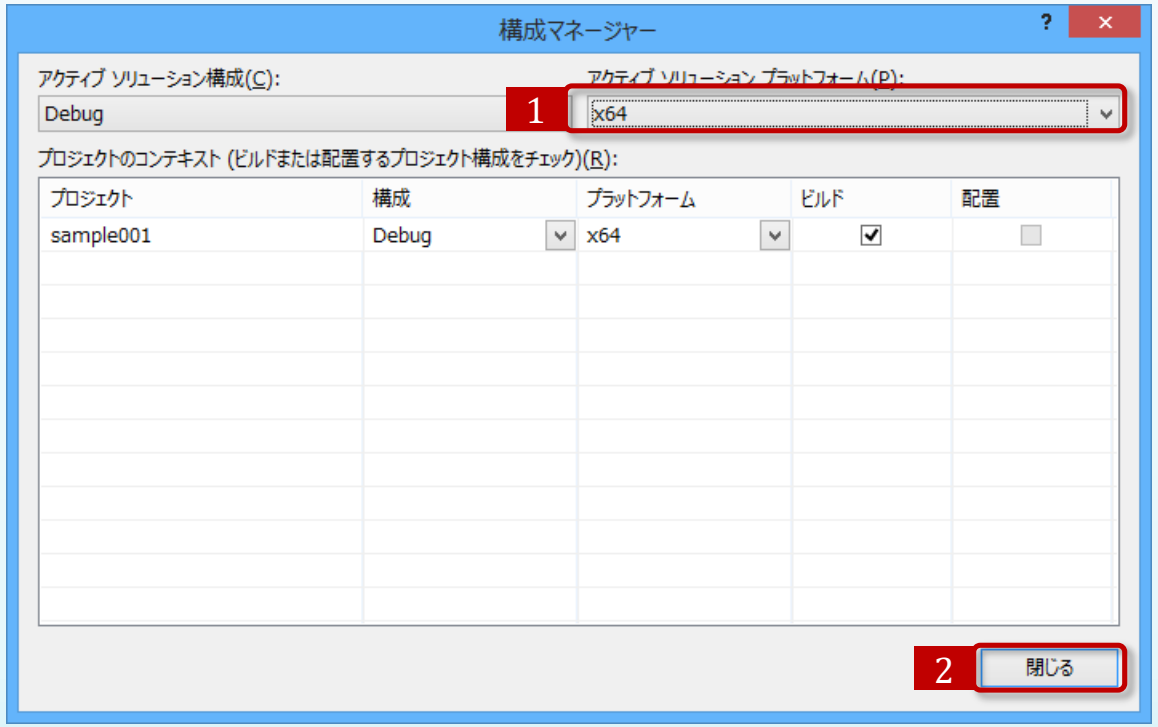
- プラットフォームから
<新規作成>をクリック



1. 新しいプラットフォームに
   x64を選択
2. 設定のコピー元に
   Win32を選択

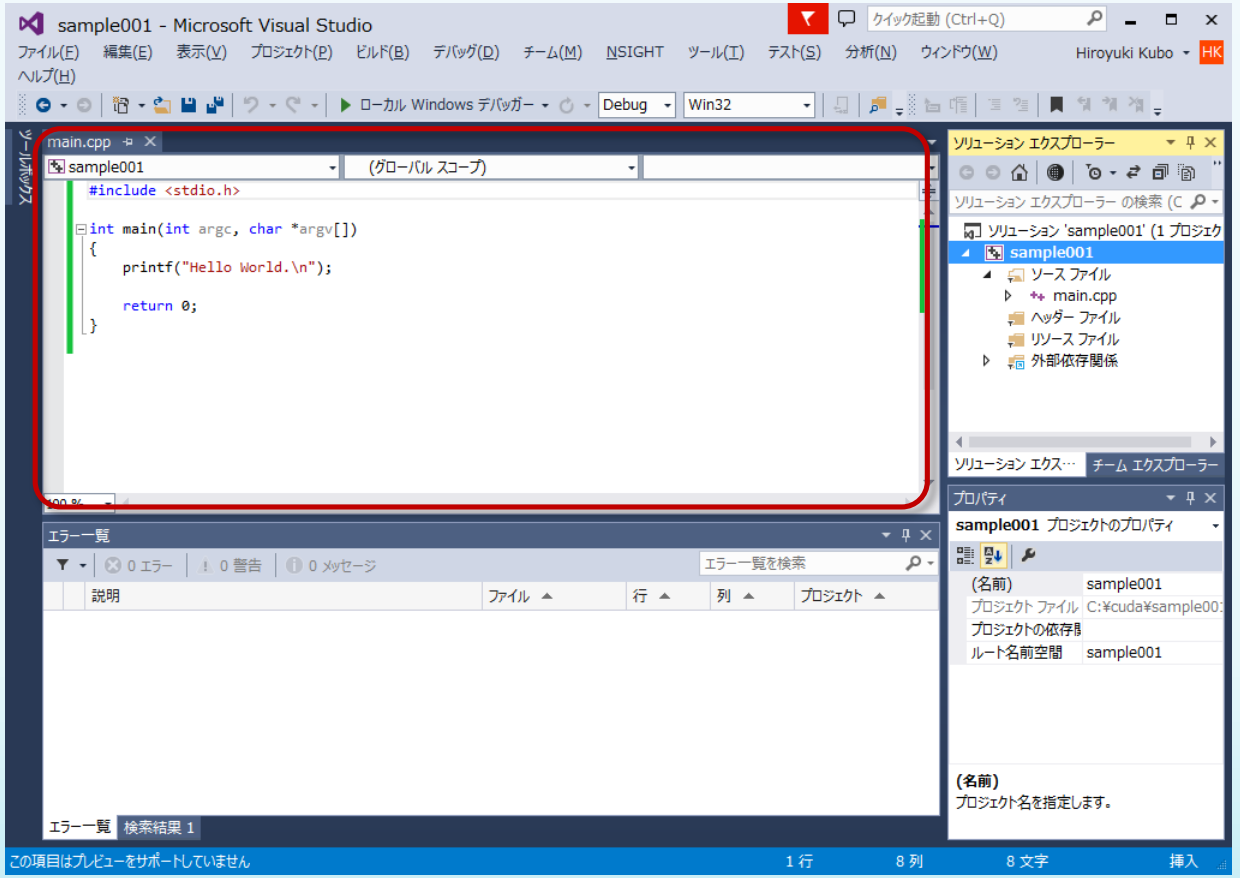


1. アクティブソリューション
   プラットフォームにx64を選択する.
   （32bit版でもCUDAは動作するが機能に制限があるため，特に制約が無い限り64bitビルドを推奨します．）

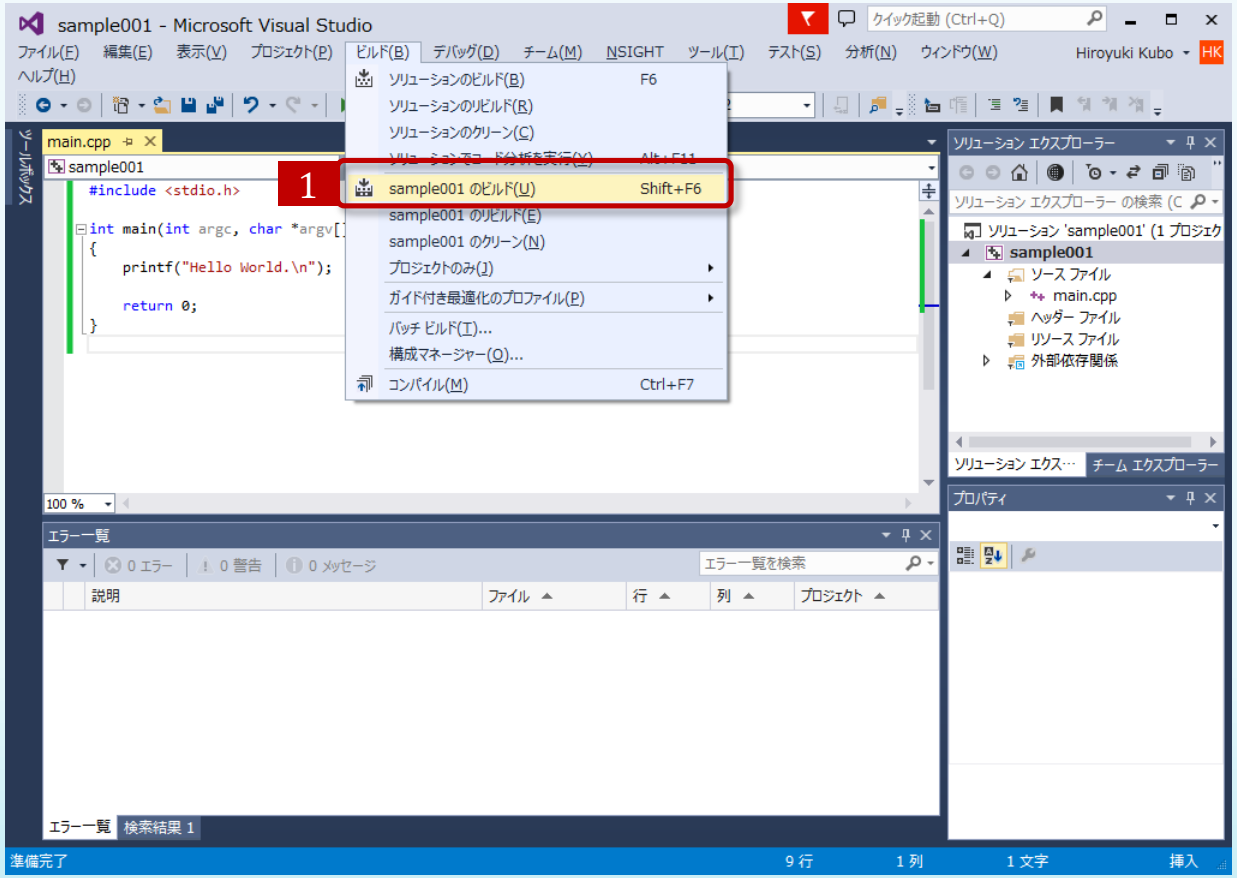


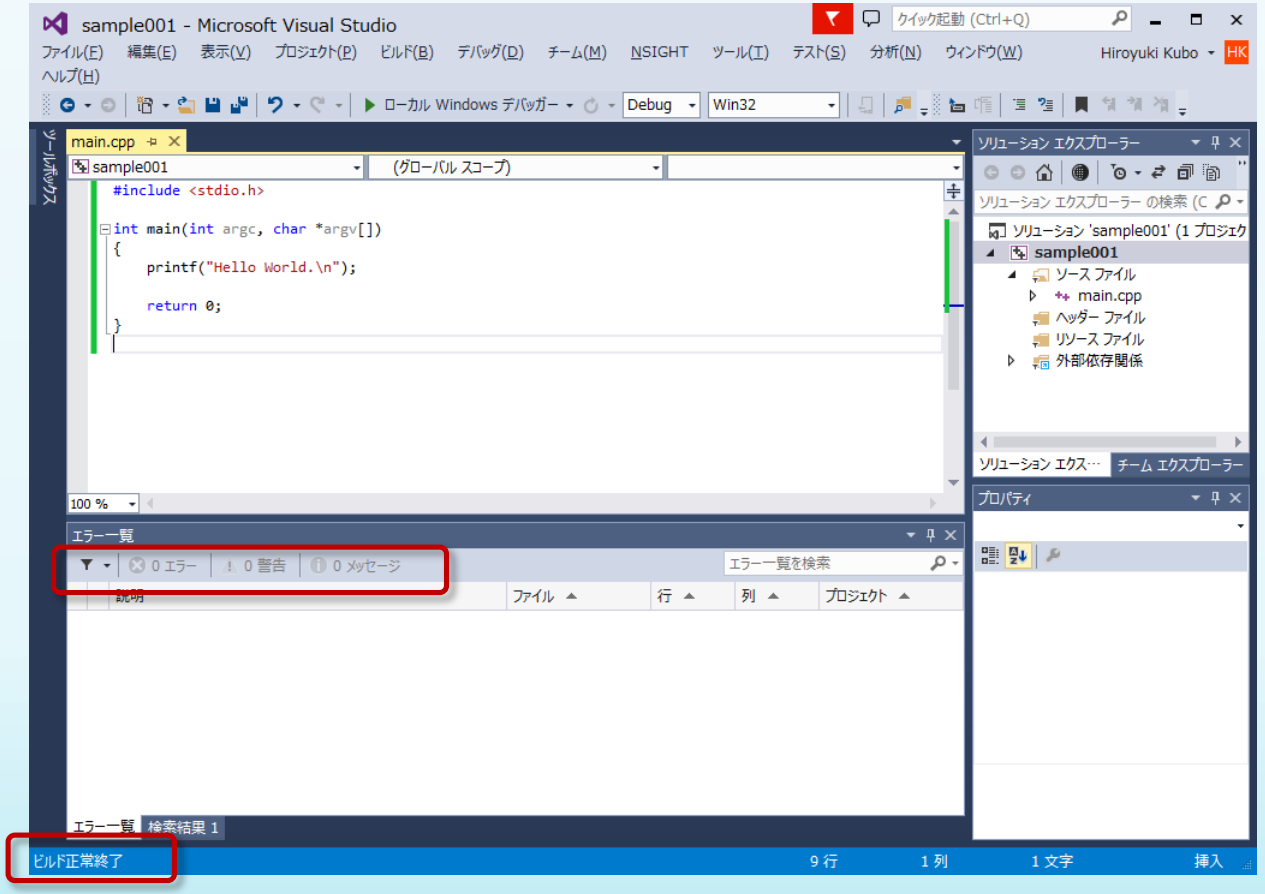
Hello World

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
	printf("Hello World.¥n");

	return 0;
}
```



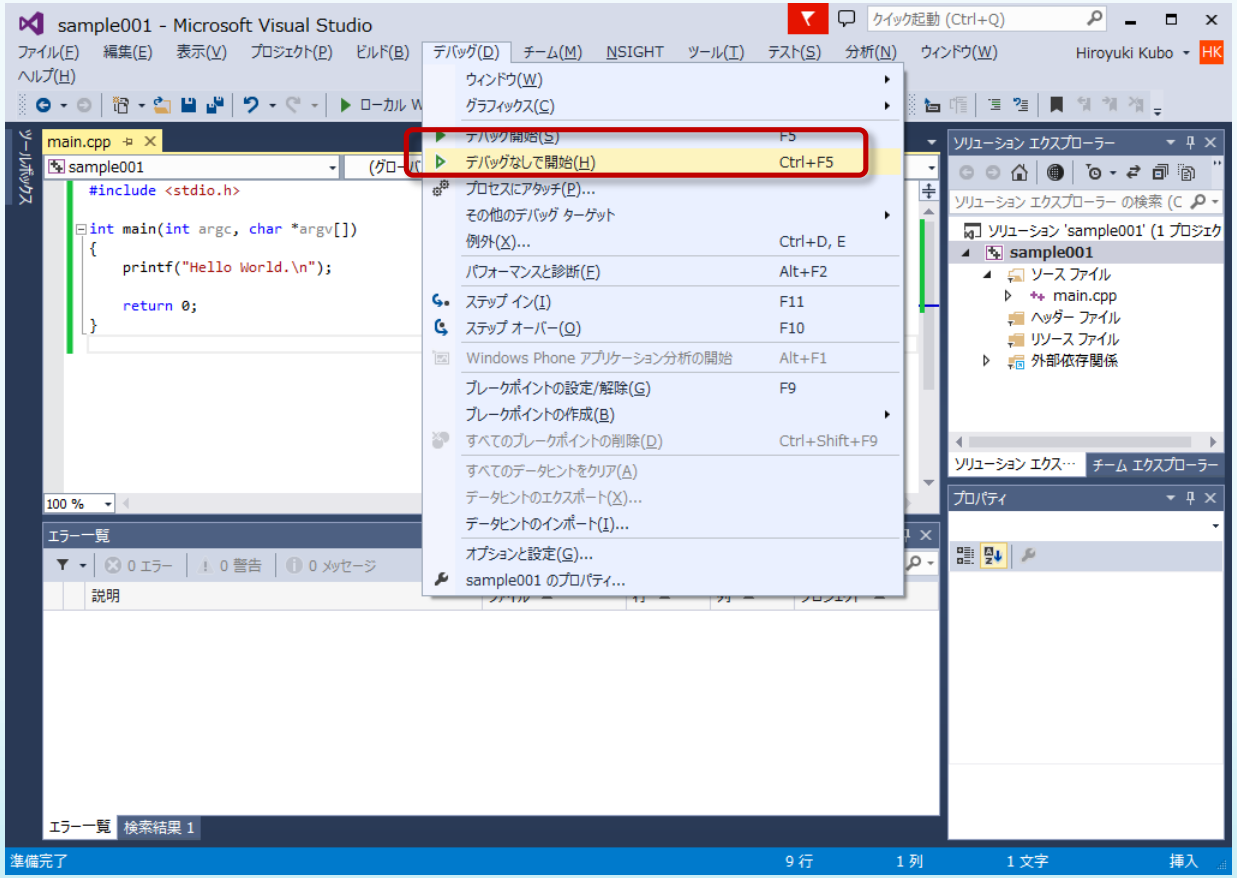
1. プログラムをコンパイルしてみる.
ビルド-sample001のビルド
をクリック

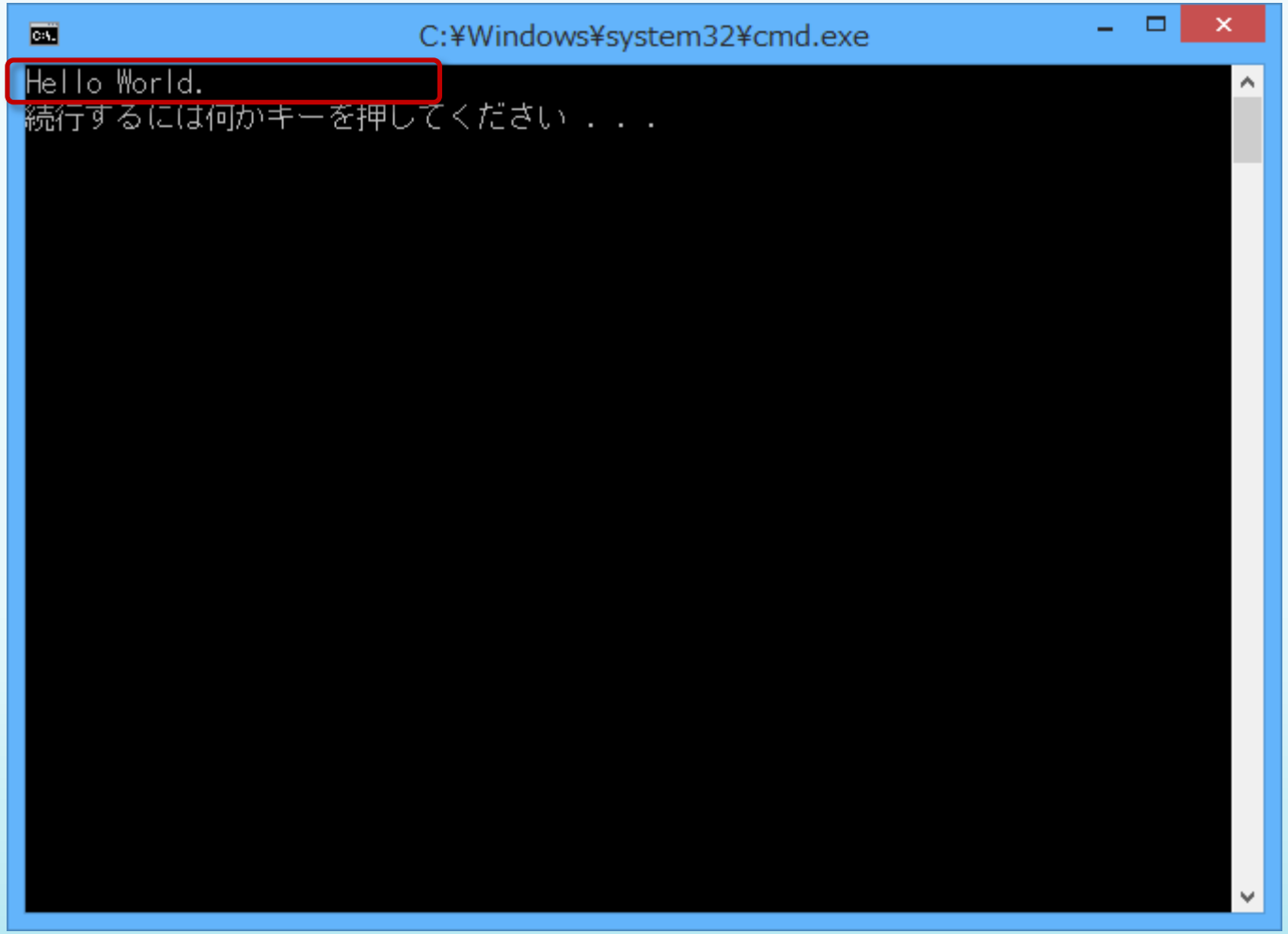




エラーが無いことを確認（おめでとう）

- せっかくだから実行してみる.
  デバッグ-デバッグなしで開始
  をクリック

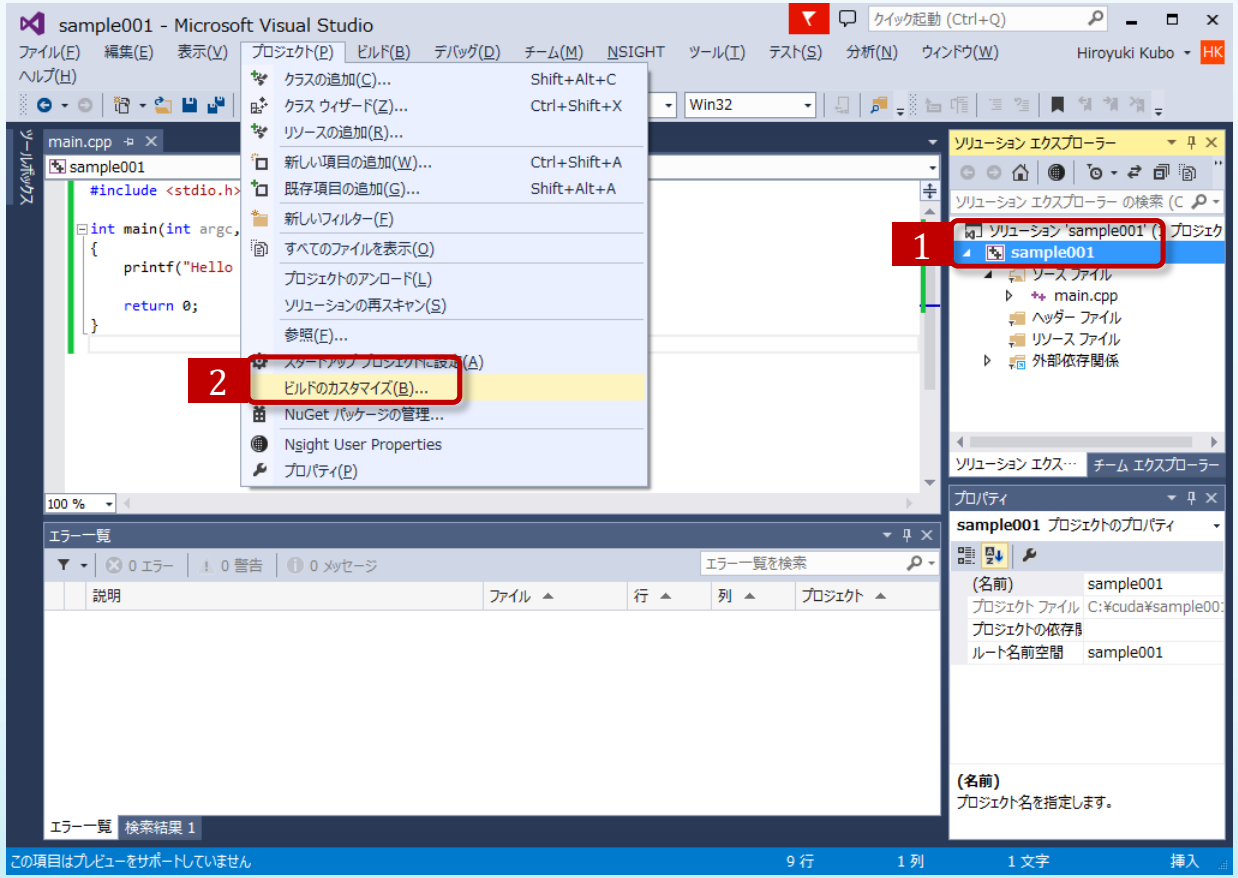



C:¥Windows¥system32¥cmd.exe
Hello World.
続行するには何かキーを押してください . . .

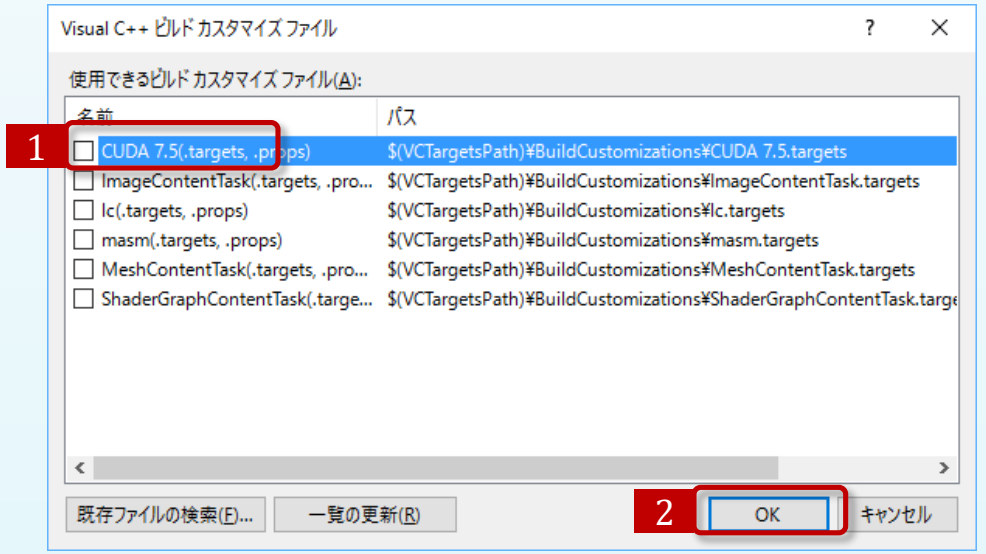
でた（おめでとう）

CUDAの設定

1. プロジェクトを選択した状態で．．．
2. プロジェクトービルドのカスタマイズをクリック

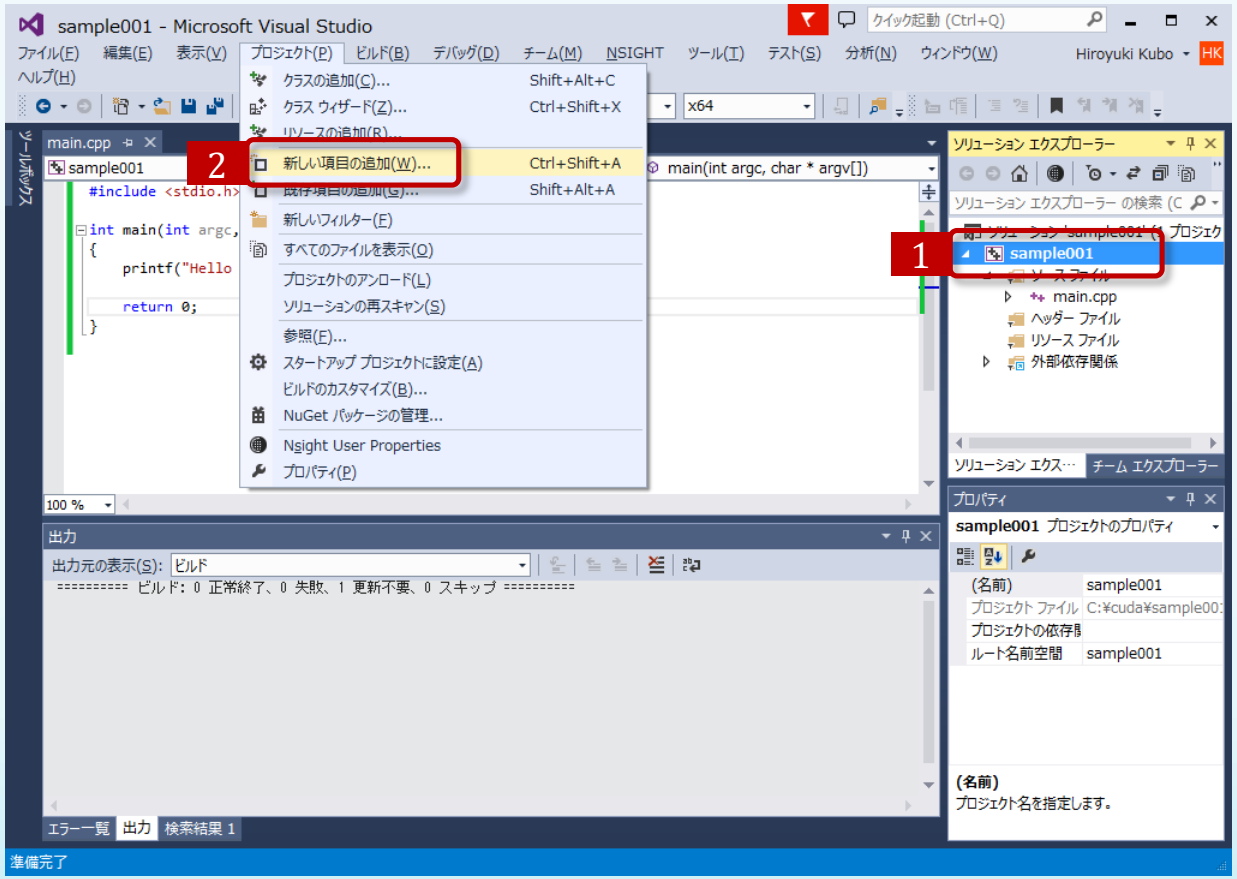


1. CUDA 7.5 にチェックを入れる.
2. OKをクリック

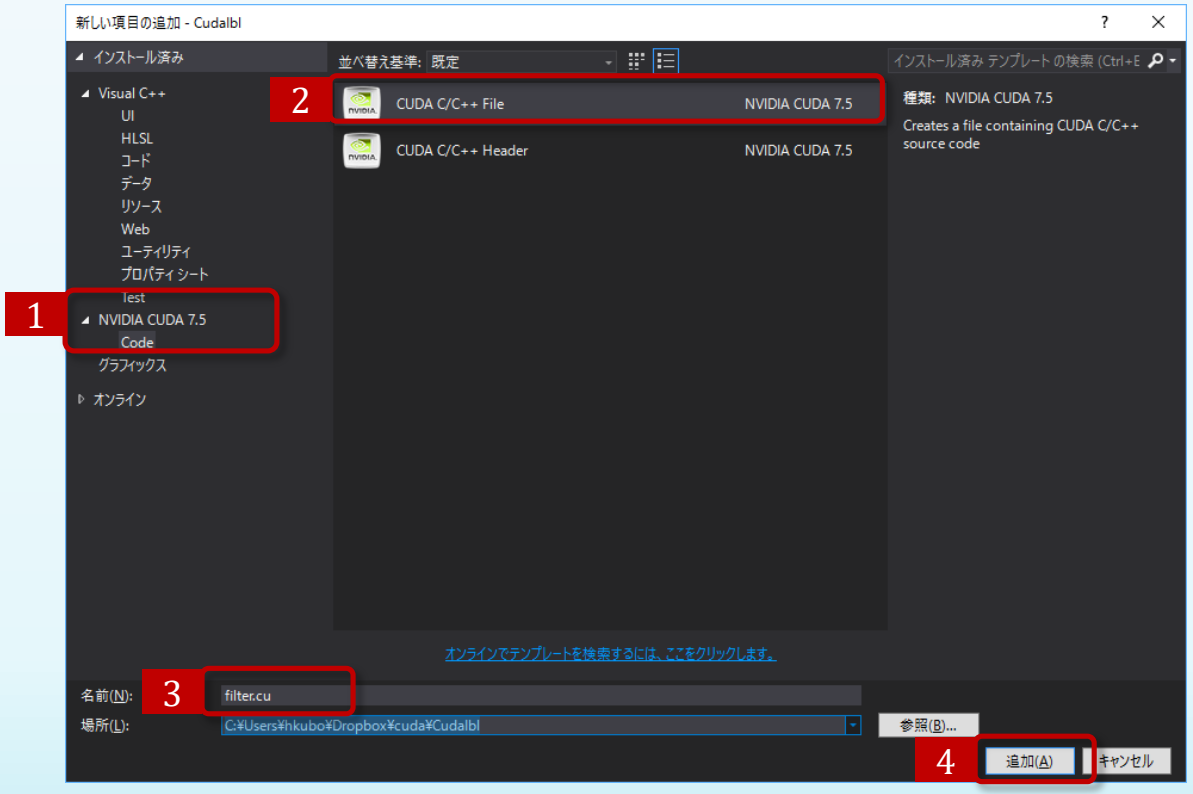


CUDA ソースコードの追加

1. プロジェクトを選択した状態で・・
2. 新しい項目の追加　をクリック

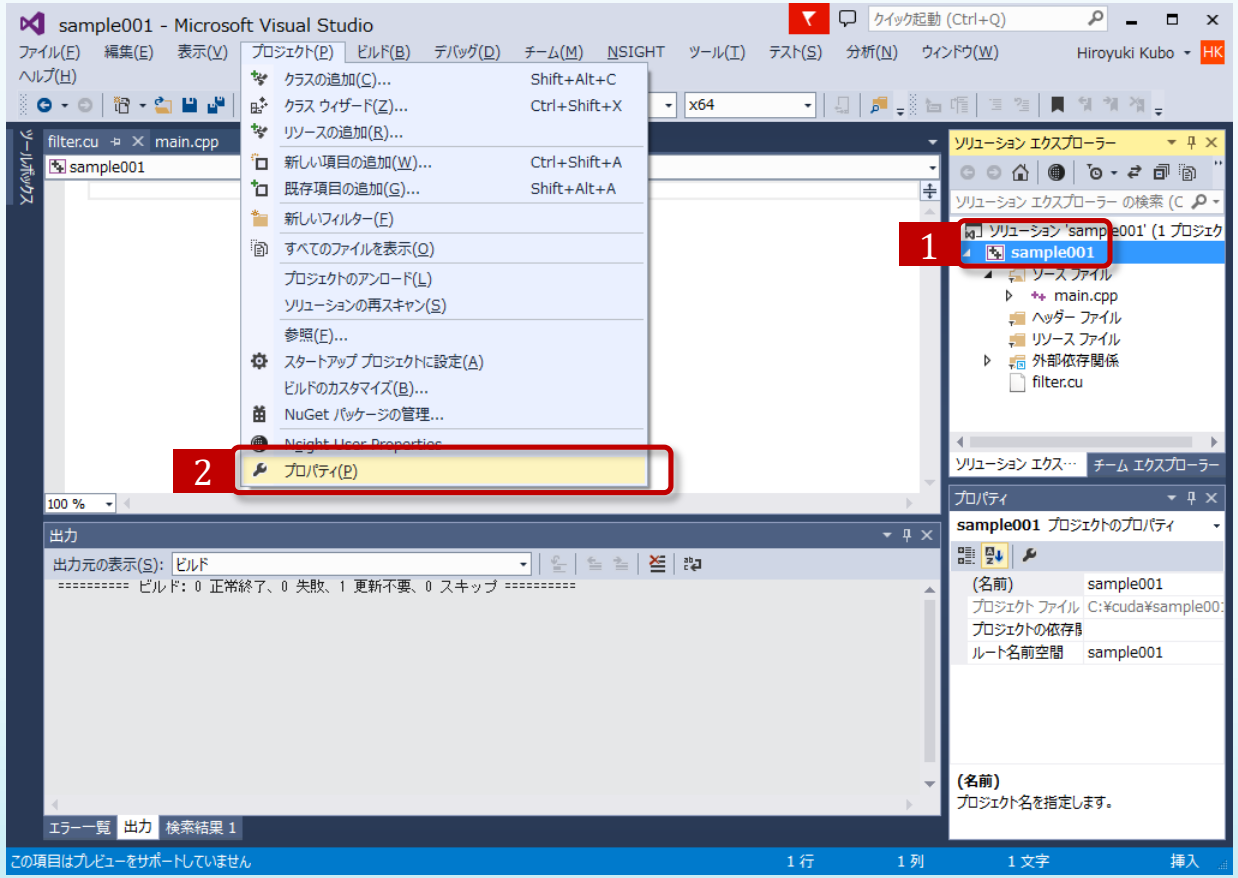


1. NVIDIA CUDA 7.5を選択し，さらにCodeを選択する
2. CUDA C/C++ Fileを選択する
3. ファイル名を指定する
   今回は”filter.cu”とする
4. 追加 をクリック

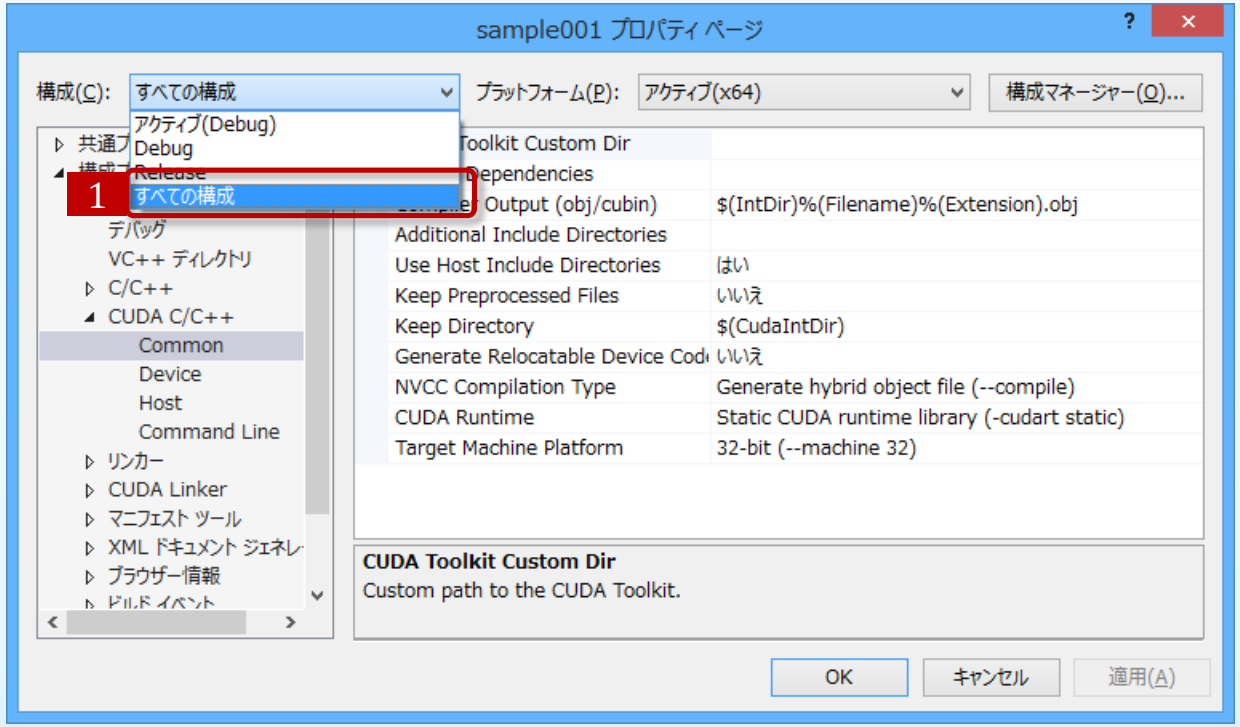


Compute Capabilityの設定

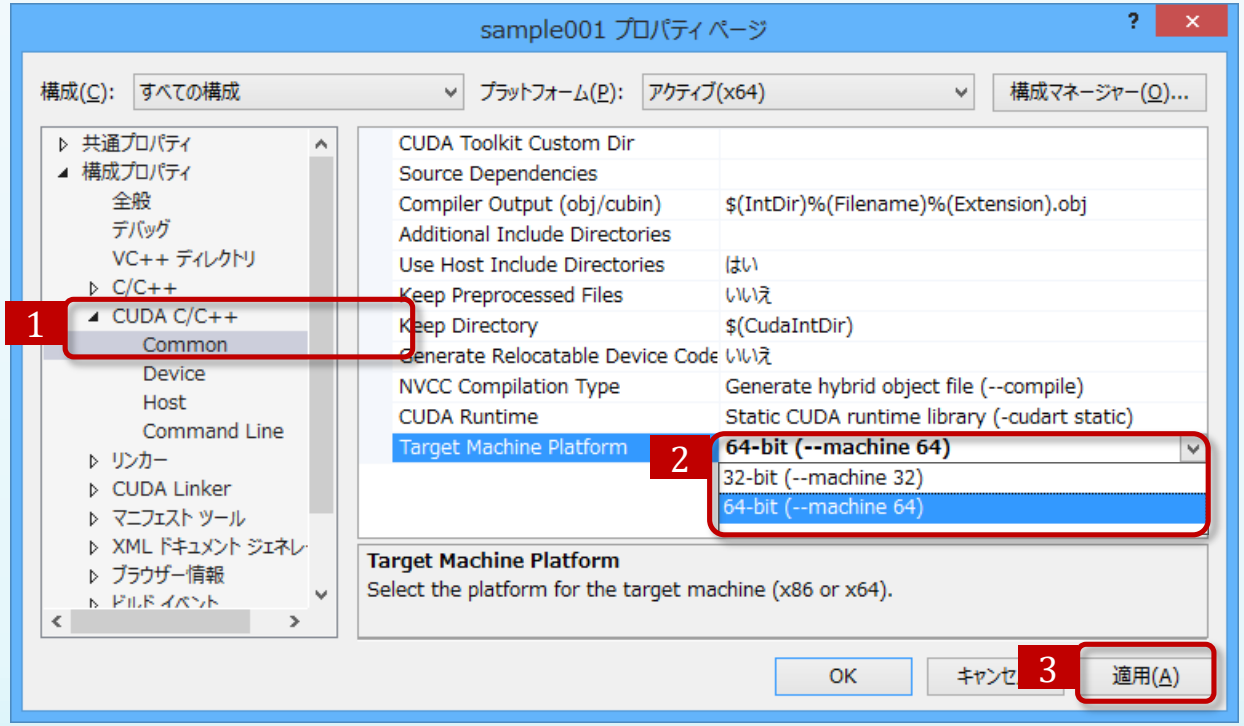
1. プロジェクトを選択した状態で
2. プロパティをクリック



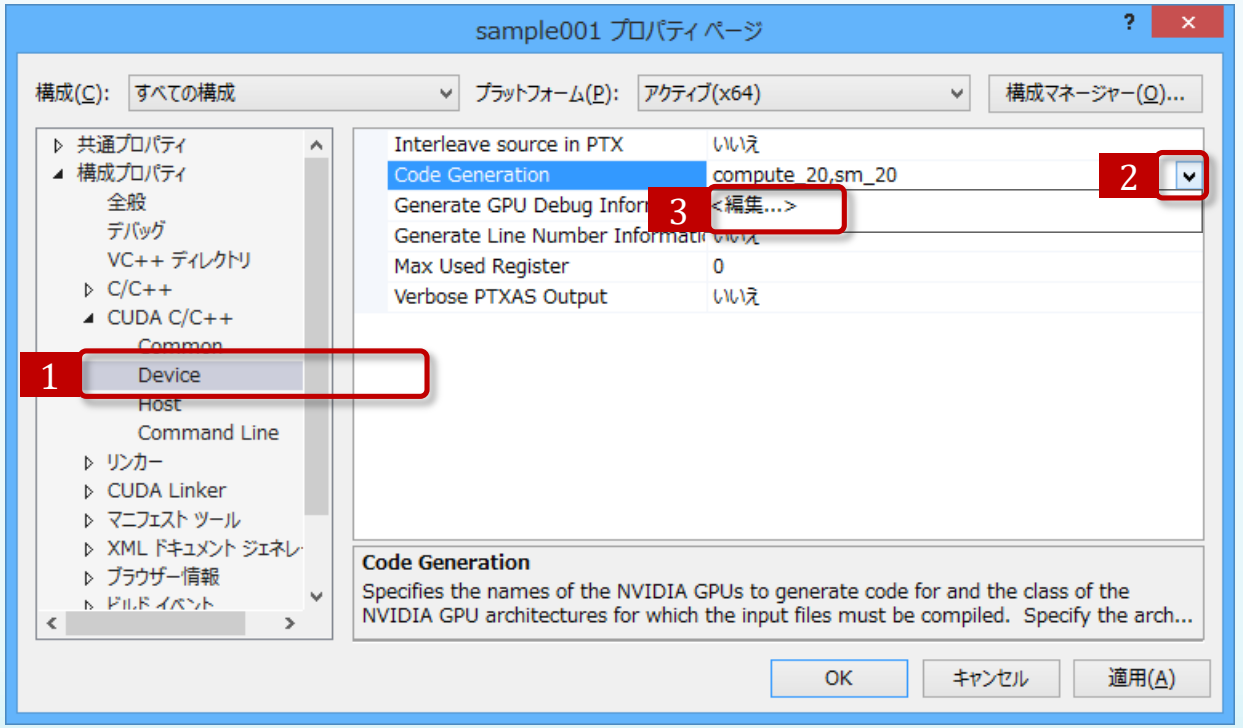
1. 構成から『すべての構成』を選択



1. CUDA C/C++のCommonを選択
2. TargetMachine Platformから
   64-bit(--machine 64)を選択する
3. 『適用』をクリック

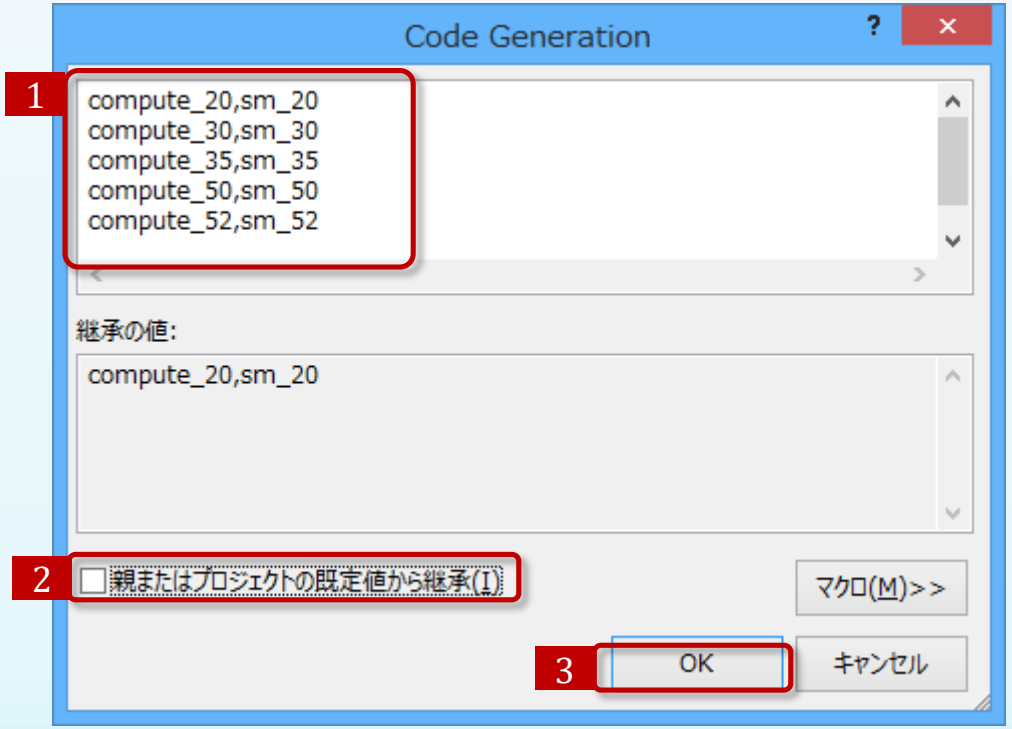


1. CUDA C/C++のDeviceを選択
2. Code Generationのドロップダウンをクリックし・・・
3. 『編集』をクリック

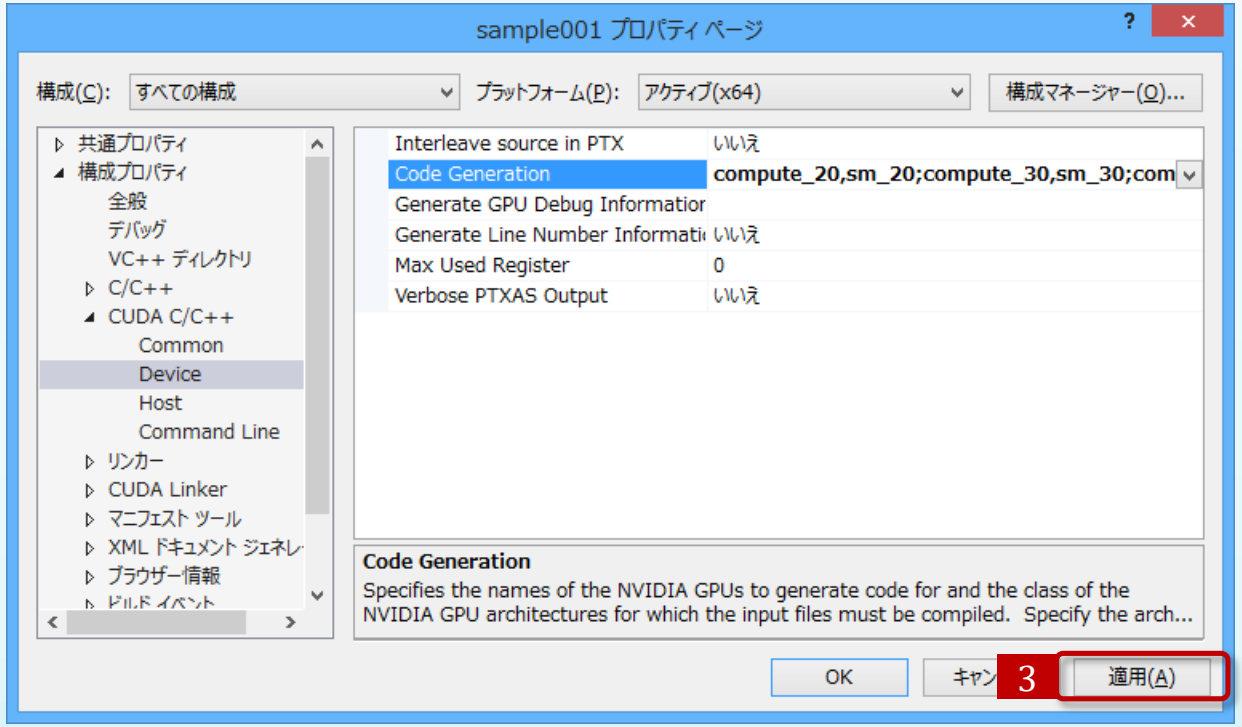


1. CUDA7.0の場合は，図のように設定
2. チェックを外す
3. OKをクリック

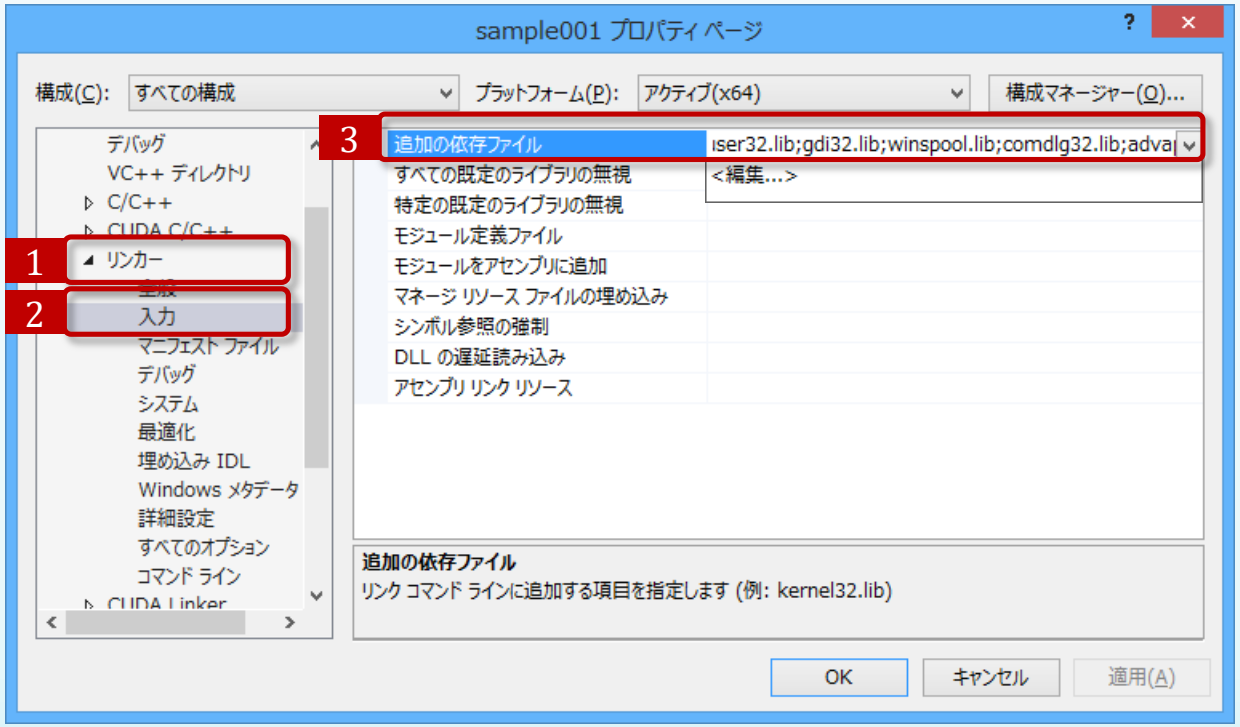
→ CUDA7.5環境でも同じだと思うけど
未検証.



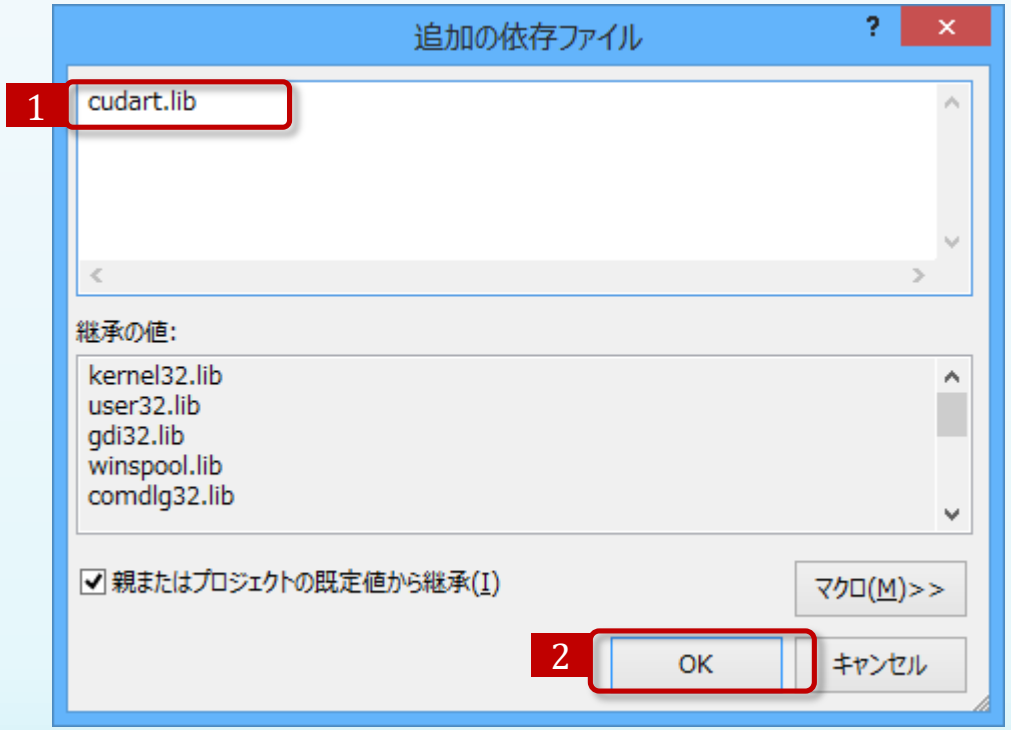
1. 『適用』をクリック


sample001 プロパティ ページ
構成(C): すべての構成
プラットフォーム(P): アクティブ(x64)
構成マネージャー(O)...
共通プロパティ
構成プロパティ
全般
デバッグ
VC++ ディレクトリ
C/C++
CUDA C/C++
Common
Device
Host
Command Line
リンカー
CUDA Linker
マニフェスト ツール
XML ドキュメント ジェネレ
ブラウザー情報
Interleave source in PTX
いいえ
Code Generation
compute_20,sm_20;compute_30,sm_30;com
Generate GPU Debug Information
Generate Line Number Informati
いいえ
Max Used Register
0
Verbose PTXAS Output
いいえ
Code Generation
Specifies the names of the NVIDIA GPUs to generate code for and the class of the NVIDIA GPU architectures for which the input files must be compiled. Specify the arch...
OK
キャン
3
適用(A)

1. リンカーを選択し，
2. さらに入力をクリックする
3. 追加の依存ファイルのドロップダウンをクリックし，『編集』をクリックする



1. “cudart.lib”入力
2. OKをクリック

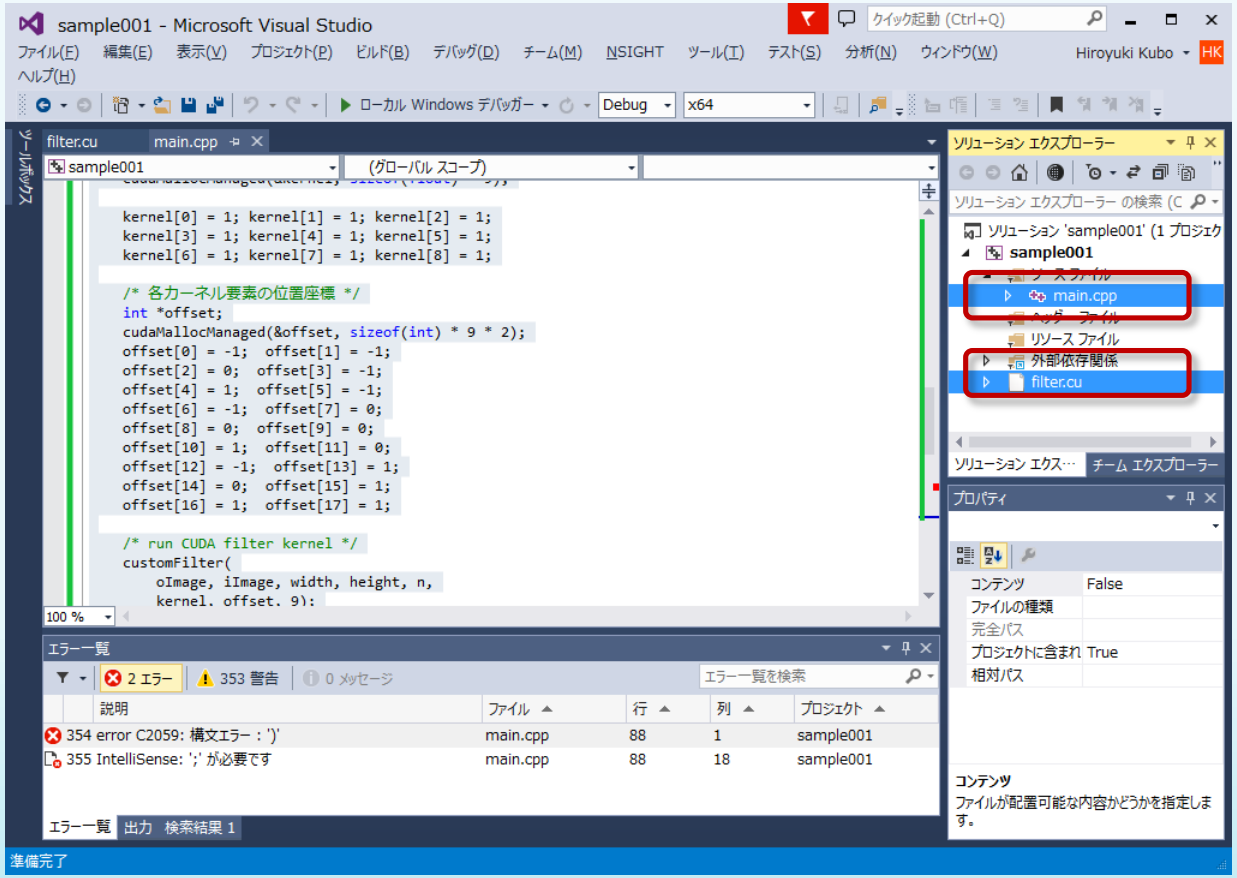


実際にプログラミングしてみる

- ３x３の平均値フィルタを実装
- main.cppとfilter.cuを編集する．

【プログラムの方針】

- 汎用性を考えて，フィルタカーネルを可変にしておく．
- フィルタカーネルのサイズ・重みはCPU側（main.cpp）で決めて，GPU側（filter.cu）に転送する．main.cppを編集してフィルタを変えれば，filter.cuは変えなくてもいろんなフィルタが試せる．
- 処理用のバッファはfloatで用意しておく．遅いのかもしれないけど気にしない．
- 入力，出力画像データ，フィルタカーネルはすべてfloatで保持する．遅いのかもしれないけど気にしない．

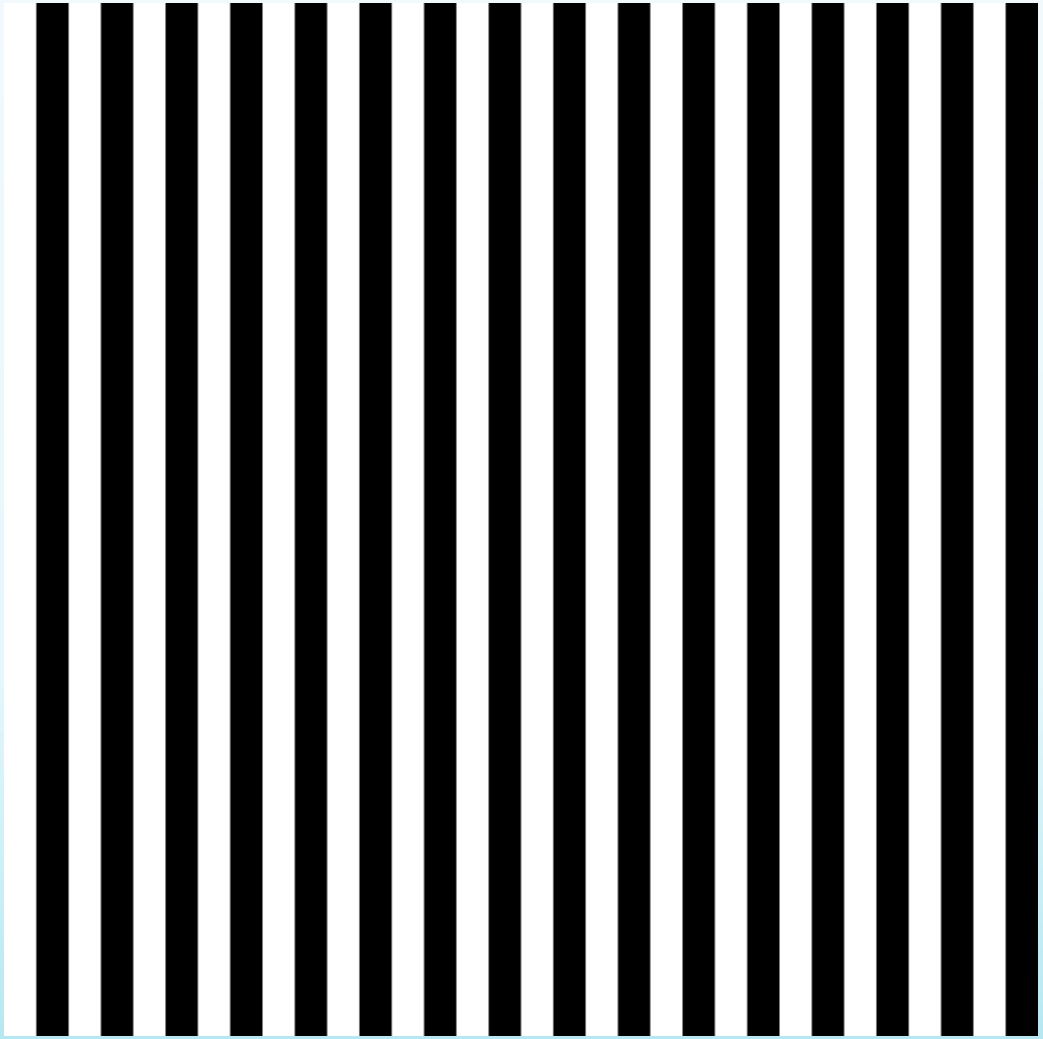


```cpp
#include <stdio.h>
#include <fstream>
#include <string>
#include <cuda_runtime.h>

void customFilter(
	float *output, const float *input,
	const int width, const int height, const int n,
	const float *kernel, const int* offset,
	const int kernelLength);

/* PPM形式で画像を保存 */
void SavePPM(const char* filename, const int width,
		const int height, const float* image)
{
	std::ofstream fout(filename);
	fout << "P3" << std::endl << width
		<< " " << height << std::endl << "255" << std::endl;
	for (int i = 0; i < width*height; i++)
		fout << (int)(image[3 * i + 0] * 255) << " "
			<< (int)(image[3 * i + 1] * 255) << " "
			<< (int)(image[3 * i + 2] * 255) << std::endl;
	fout.close();
}

int main(int argc, char *argv[])
{
	/* 128x128画像・3チャンネル */
	int width = 128, height = 128, n = 3;

	/* 入出力画像用メモリの確保 */
	float* oImage, *iImage;
	cudaMallocManaged(&iImage, sizeof(float) * width*height*n);
	cudaMallocManaged(&oImage, sizeof(float) * width*height*n);

	/* 入力画像を作成（縦縞模様） */
	for (int h = 0; h < height; h++)
	{
		for (int w = 0; w < width; w++)
		{
			float col
				= ((w - w % 4) / 4) % 2 == 0 ? 1.0 : 0.0;
			iImage[n*width*h + n*w + 0] = col;// red
			iImage[n*width*h + n*w + 1] = col;// green
			iImage[n*width*h + n*w + 2] = col;// white
		}
	}

	/* 入力画像を保存（確認用） */
	SavePPM("input.ppm", width, height, iImage);

	/* カーネル作成（平均値フィルタ） */
	float *kernel;
	cudaMallocManaged(&kernel, sizeof(float) * 9);

	kernel[0] = 1; kernel[1] = 1; kernel[2] = 1;
	kernel[3] = 1; kernel[4] = 1; kernel[5] = 1;
	kernel[6] = 1; kernel[7] = 1; kernel[8] = 1;

	/* 各カーネル要素の位置座標 */
	int *offset;
	cudaMallocManaged(&offset, sizeof(int) * 9 * 2);
	offset[0] = -1;  offset[1] = -1;
	offset[2] = 0;   offset[3] = -1;
	offset[4] = 1;   offset[5] = -1;
	offset[6] = -1;  offset[7] = 0;
	offset[8] = 0;   offset[9] = 0;
	offset[10] = 1;  offset[11] = 0;
	offset[12] = -1;  offset[13] = 1;
	offset[14] = 0;  offset[15] = 1;
	offset[16] = 1;  offset[17] = 1;

	/* run CUDA filter kernel */
	customFilter(
		oImage, iImage, width, height, n,
		kernel, offset, 9);

	/* CUDAの終了を待つ */
	cudaDeviceSynchronize();

	/* 処理結果を出力 */
	SavePPM("output.ppm", width, height, oImage);


	/* メモリ解放 */
	cudaFree(kernel);
	cudaFree(iImage);
	cudaFree(offset);
	cudaFree(oImage);

	/* 後片付け */
	cudaDeviceReset();

	return 0;
}
```

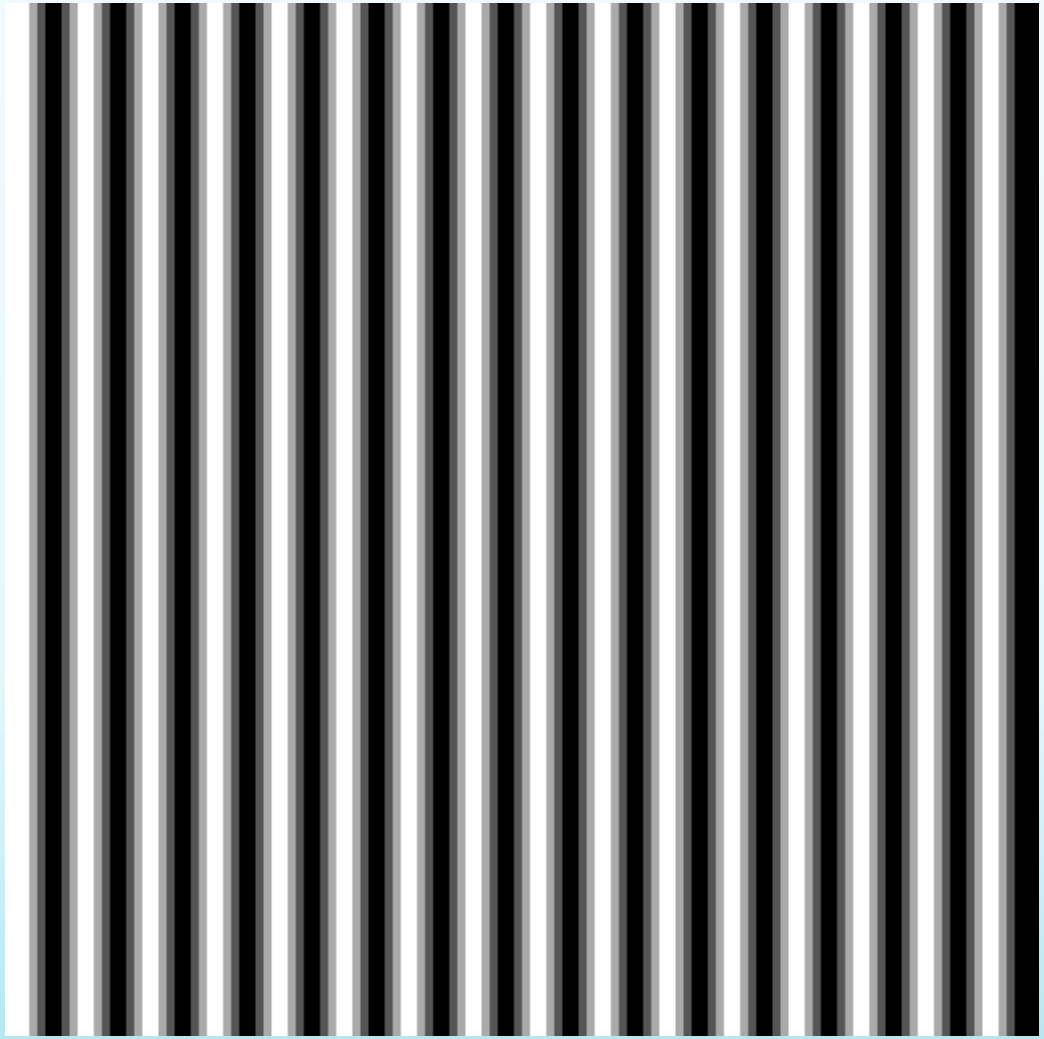
main.cpp

```cpp
#include "device_launch_parameters.h"

__global__
void customFilterKernel(
	float *output, const float *input,
	const int width, const int height, const int n,
	const float *kernel, const int* offset, const int kernelLength)
{
	/* 何番目の画素を見ているかを計算 */
	int gid = blockDim.x * blockIdx.x + threadIdx.x;
	if (gid < width*height*n)
	{
		float v = 0.0;
		float sum_weight = 0.f;

		/* 画像中のどの位置・色を見ているかを計算 */
		int c = gid % n;
		int w = ((gid - c) / n) % width;
		int h = ((gid - (n * w + c)) / (n*width));

		/* カーネルサイズ（3 x 3フィルタだから9）でループ */
		for (int i = 0; i < kernelLength; i++)
		{
			int ofx = offset[2 * i + 0];
			int ofy = offset[2 * i + 1];

			int x = w + ofx;
			int y = h + ofy;

			if (x < 0 || x >= width || y < 0 || y >= height)
				continue;

			float weight = kernel[i];
			sum_weight += weight;

			v += weight * input[n * width * y + n * x + c];
		}
		output[gid] = v / sum_weight; /* ウェイトの和を1に正規化 */
	}
}
void customFilter(
	float *output, const float *input,
	const int width, const int height, const int n,
	const float *kernel, const int* offset, const int kernelLength)
{
	dim3 blockDim(128);
	dim3 gridDim((width*height*n + blockDim.x - 1) / blockDim.x);
	customFilterKernel <<<gridDim, blockDim >>>
		(output, input, width, height, n,
			kernel, offset, kernelLength);
}
```

filer.cu



入力画像



出力画像

計算速度比較

- 入力画像：5120 x 2560 [pixel]
- 処理：21x21の平均値フィルタ
- 計算機：
  - CPU: Intel Core i7 3.00GHz, 8-core, 16-thread
  - GPU: Geforce GTX 980 (2048-cuda core)
- 処理時間
  - CPUシングルスレッド：　30.1[sec]
  - CPUマルチスレッド：　5.18[sec]
  - GPU (CUDA):　1.23[sec]

思ったほど速くない！！

最適化が重要な模様.

CUDAにはメモリの種類がたくさんある.
このサンプルでは，コードが単純にかけるユニファイドメモリを使った.
カーネル部分には容量は小さい（64kB）けどアクセスが高速な
コンスタントメモリを使うべきらしい.

おしまい