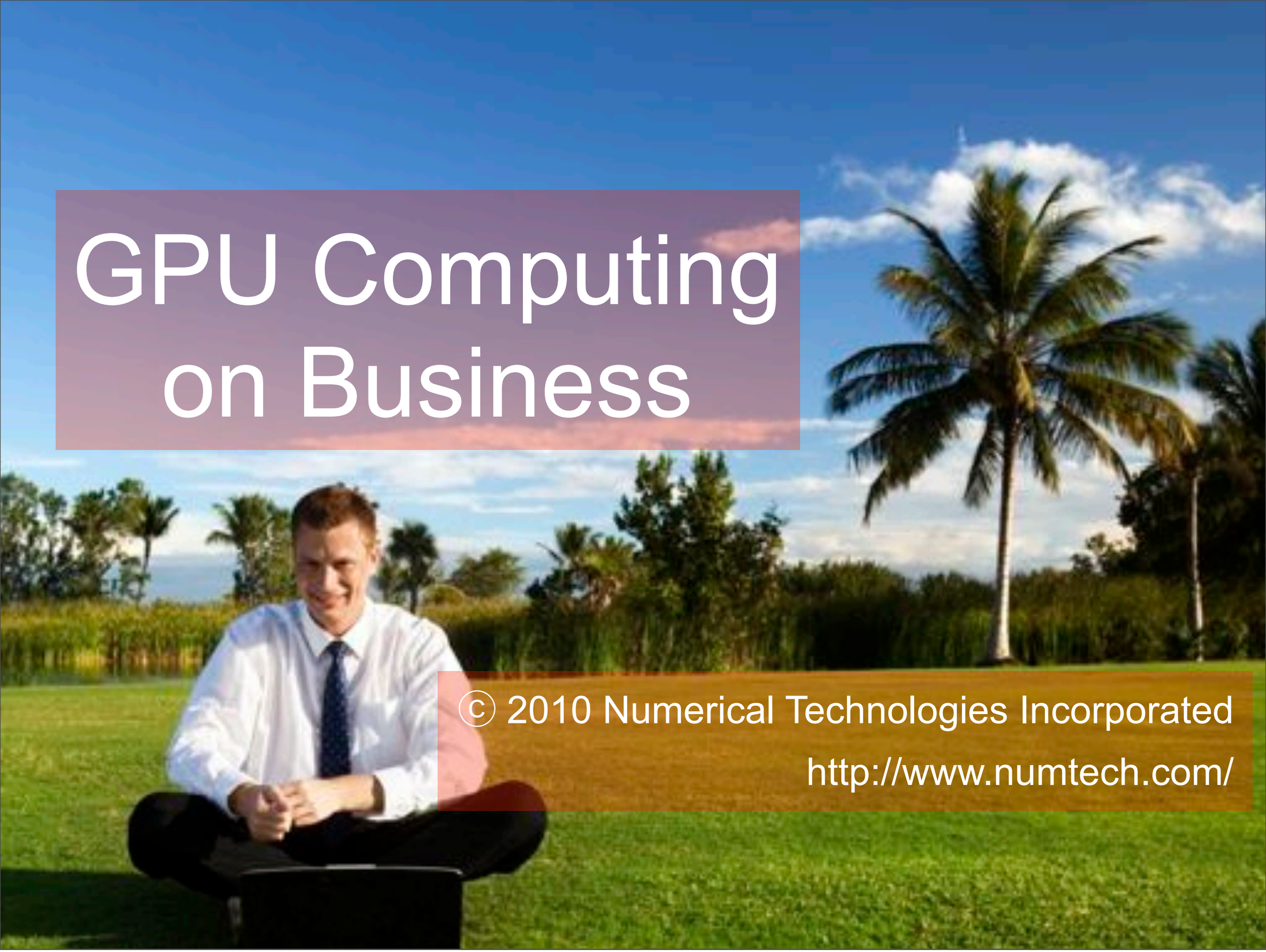
GPU Computing on Business



http://www.numtech.com/

HPC向けに普及が進む GPU Computing

しかし…

ビジネスへの普及は遅い


それはなぜか…


GPU コンピューティングは速いんだぜ！
話が噛み合ってない

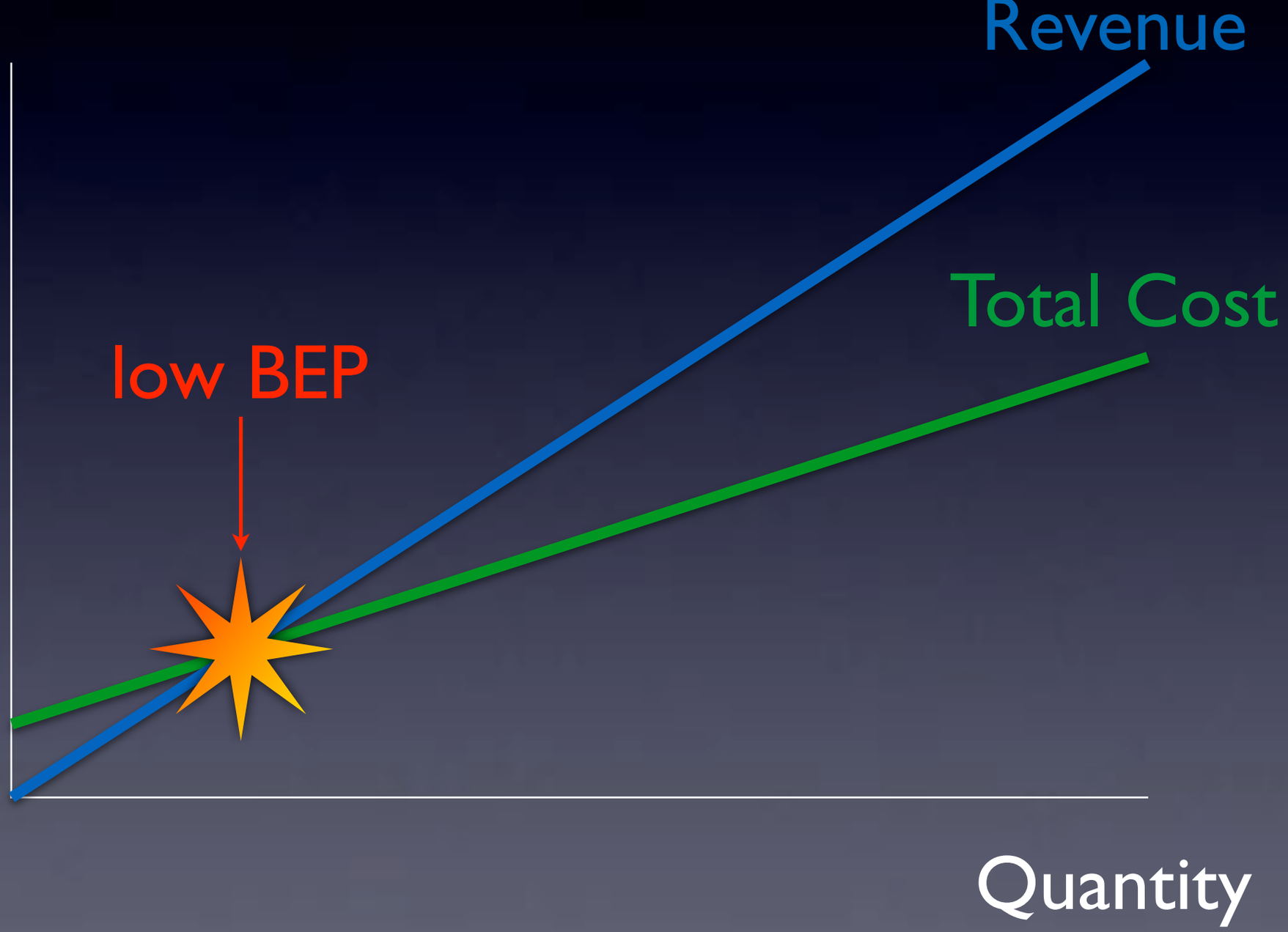
アカデミックのGPU Computingでは...

優秀な人が難しい学習の末に...

...辿り着く達人の技を駆使している

そんなのがCUDA Zoneにたくさんある。

学術系は損益分岐点が低い。投下コストを低く見積もれるからGPU Computingに投資できるのである



一方、ビジネスには...

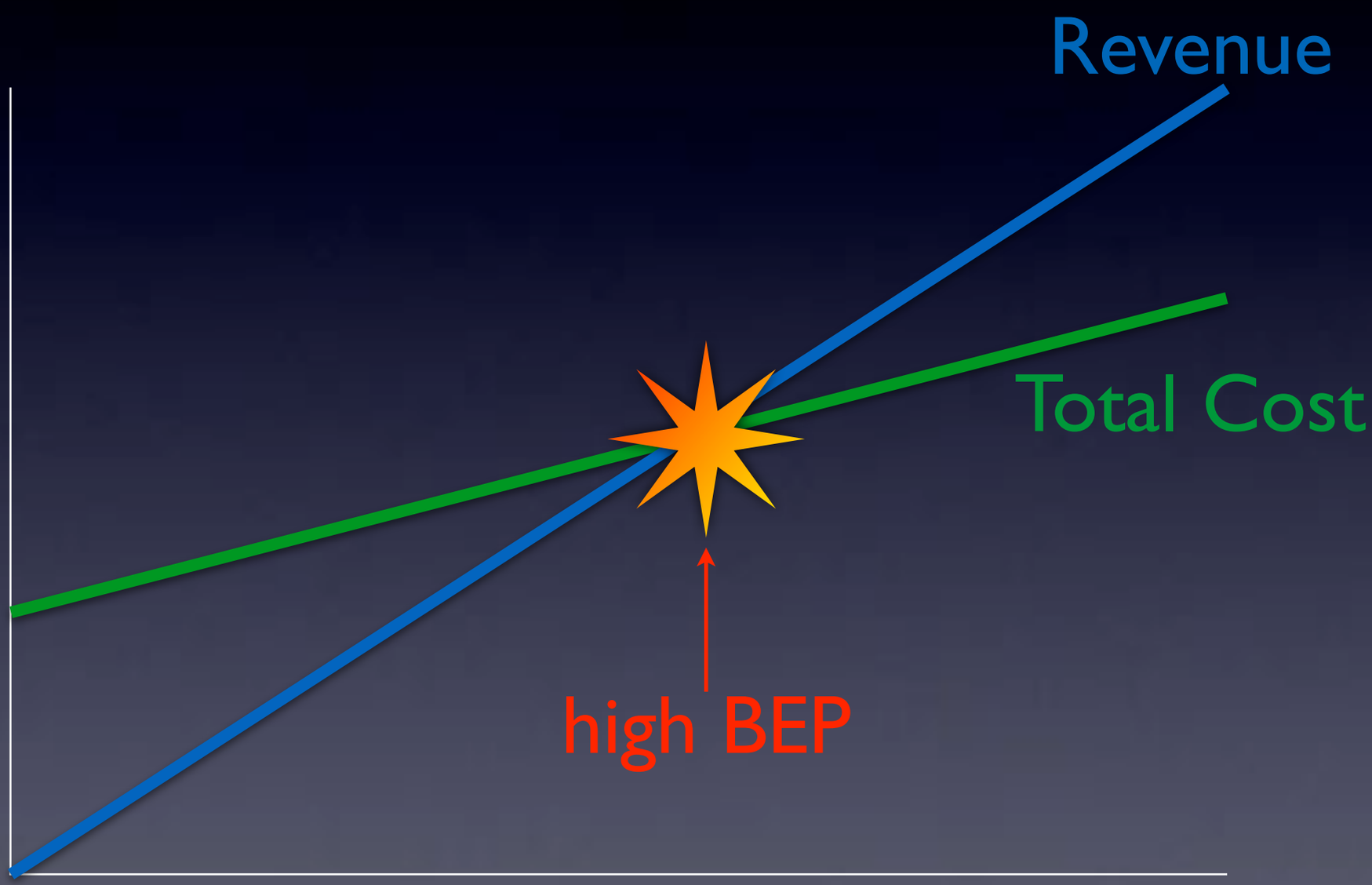
GPU Computingの普及を阻む壁がある


ちっとも速くならないぜ。俺のCUDAプログラムのどこが悪いってんだ。
難しい学習


エンジニアの採用費、教育費、外部専門家への支払い...
こりゃハードウェア増強した方がましだな。
技術以外のコスト負担

技術の継続性への不安

結局、多くのビジネスの視点から見ればGPU Computingは高コストすぎて敬遠してしまう



壁を破る方法

使い慣れた言語で開発

もちろんnvcc、
Toolkit、SDKは
使いたくないし、
覚えたくもない

GPUを外しても
動くようにしておけば、Teslaがなくなっても大丈夫

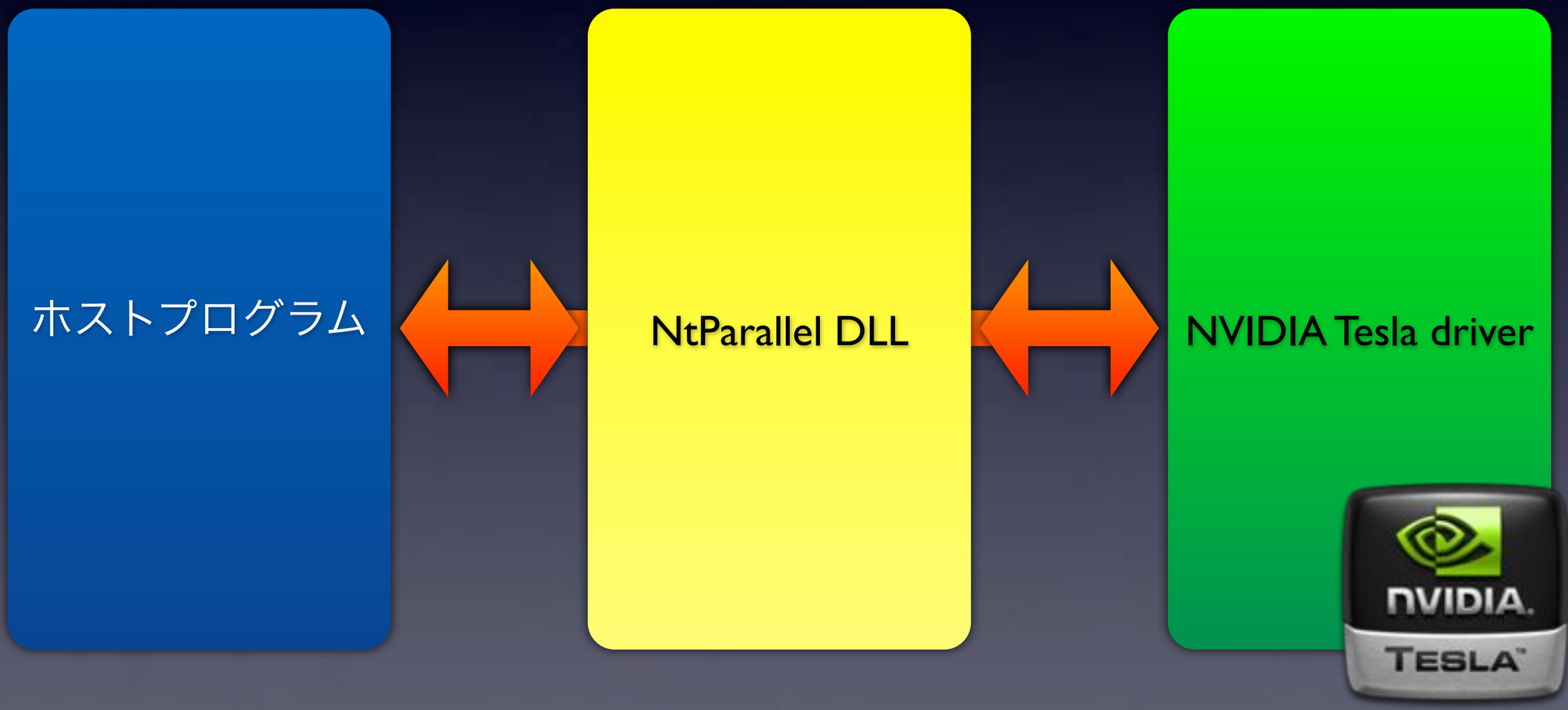
それにExcelから
でもGPUが使え
ればエンジニアも
不要じゃないか


そんなことが
できるか

Idea

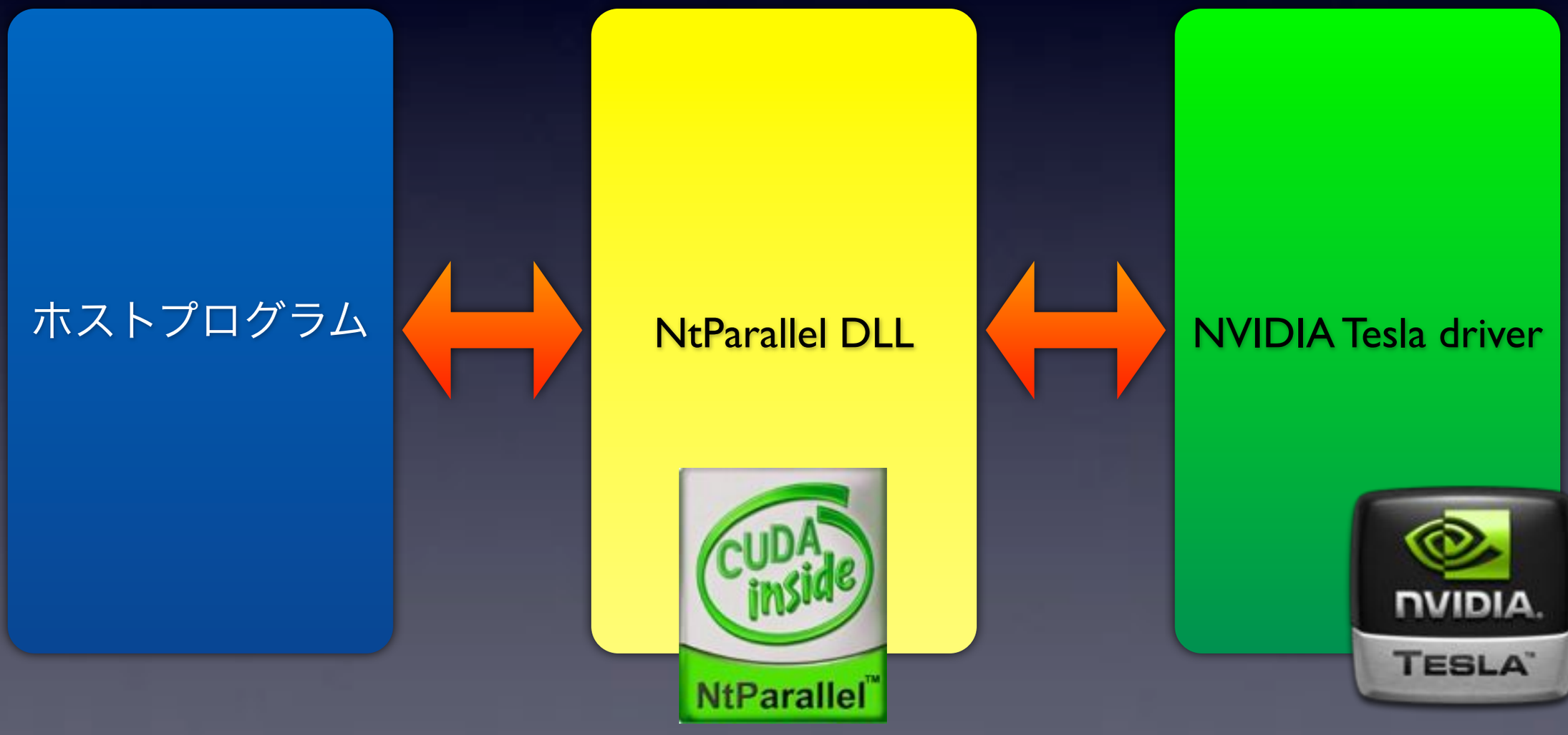
Lambda Expression

ホストプログラムから呼び出すドライバ(NtParallel DLL)を用意



CUDAへの呼び出しをドライバ(NtParallel DLL)の中に隠蔽

CUDAへの呼び出しをドライバ(NtParallel DLL)の中に隠蔽



GPU呼び出し関数

nt_parallel_for

```
bool nt_parallel_for(
  forループ内で実行するプログラム(string),
  forループの繰り返し回数(int),
  forループに与える引数配列1(void*),
  forループに与える引数配列2(void*),
  ...
)
```

元のプログラム


```
// Black Scholes Option Formula Batch Processing Demo.
void batch_black_scholes_pricer(
   int array_size,
   double* o_data,
   double* r_data,
   double* sigma_data,
   double* s_data,
   double* k_data,
   double* t_data
)
{
   // for-loop program.
   for (int i = 0; i < array_size; ++i) {
      double R = r_data[i];
      double Sigma = sigma_data[i];
      double S = s_data[i];
      double K = k_data[i];
      double T = t_data[i];
      double rt = R * T;
      double sigmaSqrtT = Sigma * sqrt(T);
      double d = (log(S / K) + rt) / sigmaSqrtT + sigmaSqrtT * 0.5;
      o_data[i] = S * NormDist(d) - K * NormDist(d - sigmaSqrtT) * exp(- rt);
   }
   // now you have output in o_data.
}
```

よくあるforループに注目


```
// Black Scholes Option Formula Batch Processing Demo.
void batch_black_scholes_pricer(
   int array_size,
   double* o_data,
   double* r_data,
   double* sigma_data,
   double* s_data,
   double* k_data,
   double* t_data
)
{
   // for-loop program.
   for (int i = 0; i < array_size; ++i) {
      double R = r_data[i];
      double Sigma = sigma_data[i];
      double S = s_data[i];
      double K = k_data[i];
      double T = t_data[i];
      double rt = R * T;
      double sigmaSqrtT = Sigma * sqrt(T);
      double d = (log(S / K) + rt) / sigmaSqrtT + sigmaSqrtT * 0.5;
      o_data[i] = S * NormDist(d) - K * NormDist(d - sigmaSqrtT) * exp(- rt);
   }
   // now you have output in o_data.
}
```

forループの中を文字列(ラムダ式)に変換

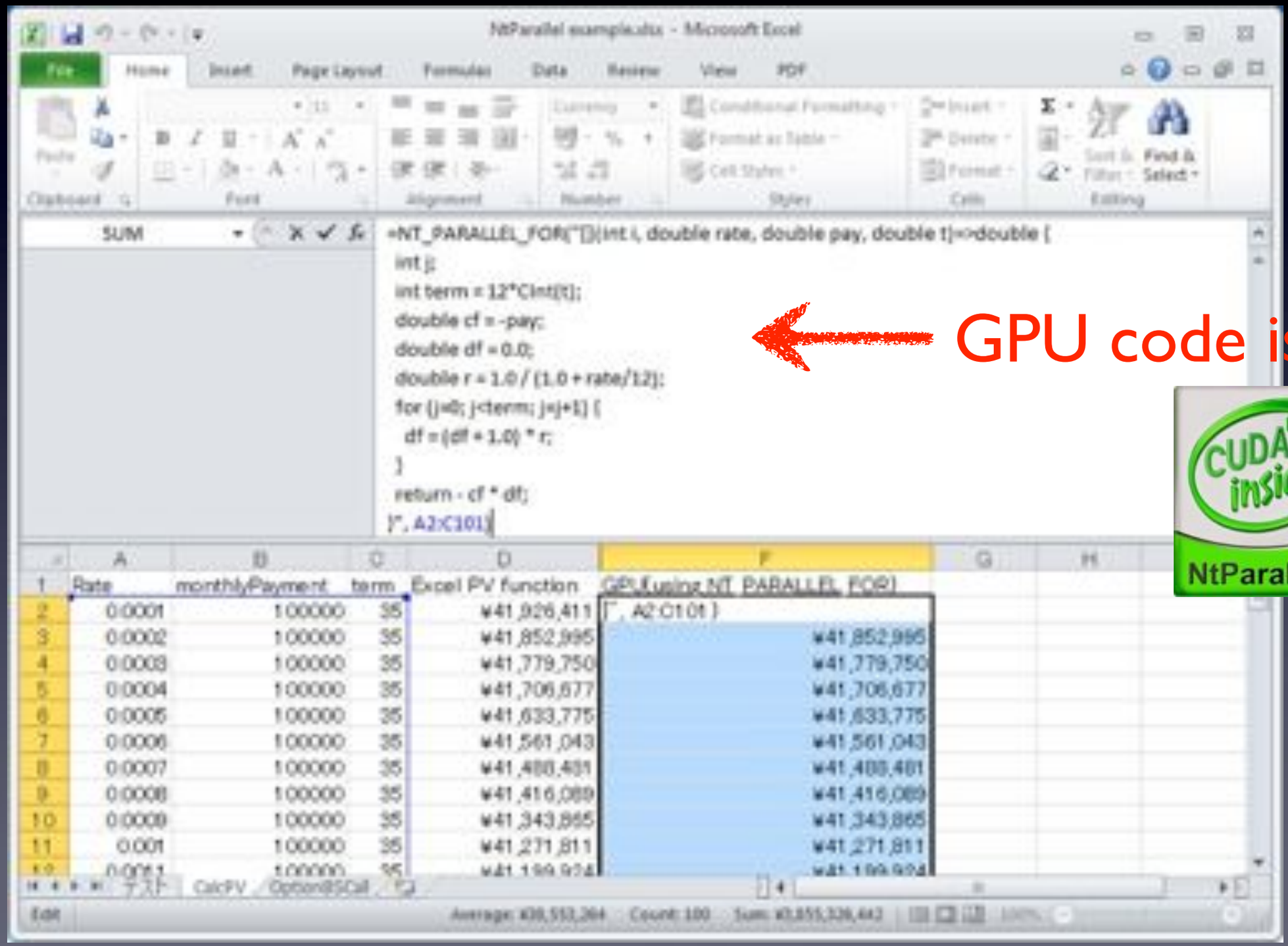

```
...
for (int i = 0; i < array_size; ++i) {
   double R = r_data[i];
   double Sigma = sigma_data[i];
   double S = s_data[i];
   double K = k_data[i];
   double T = t_data[i];
   double rt = R * T;
   double sigmaSqrtT = Sigma * sqrt(T);
   double d = (log(S / K) + rt) / sigmaSqrtT + sigmaSqrtT * 0.5;
   o_data[i] = S * NormDist(d) - K * NormDist(d - sigmaSqrtT) * exp(- rt);
}
...
```

```
...
string code = "
   [](int i, double R, double Sigma, double S, double K, double T) => double {
      double rt = R * T;
      double sigmaSqrtT = Sigma * sqrt(T);
      double d = (log(S / K) + rt) / sigmaSqrtT + sigmaSqrtT * 0.5;
      return S * NormDist(d) - K * NormDist(d - sigmaSqrtT) * exp(- rt);
   }
";
...
```

ホスト言語から呼び出す


```
// Black Scholes Option Formula Batch Processing Demo.
void batch_black_scholes_pricer(
   int array_size,
   double* o_data,
   double* r_data,
   double* sigma_data,
   double* s_data,
   double* k_data,
   double* t_data
)
{
   // for-loop program.
   string code = “
      [](int i, double R, double Sigma, double S, double K, double T) => double {
         double rt = R * T;
         double sigmaSqrtT = Sigma * sqrt(T);
         double d = (log(S / K) + rt) / sigmaSqrtT + sigmaSqrtT * 0.5;
         return S * NormDist(d) - K * NormDist(d - sigmaSqrtT) * exp(- rt);
      }
   “;
   // call the GPU.
   nt_parallel_for(code, array_size, o_data, r_data, sigma_data, s_data, k_data, t_data);
   // now you have output in o_data.
}
```

Excelからさえ呼び出し可能



これで難解な学習は不要、

生産性は10倍だ

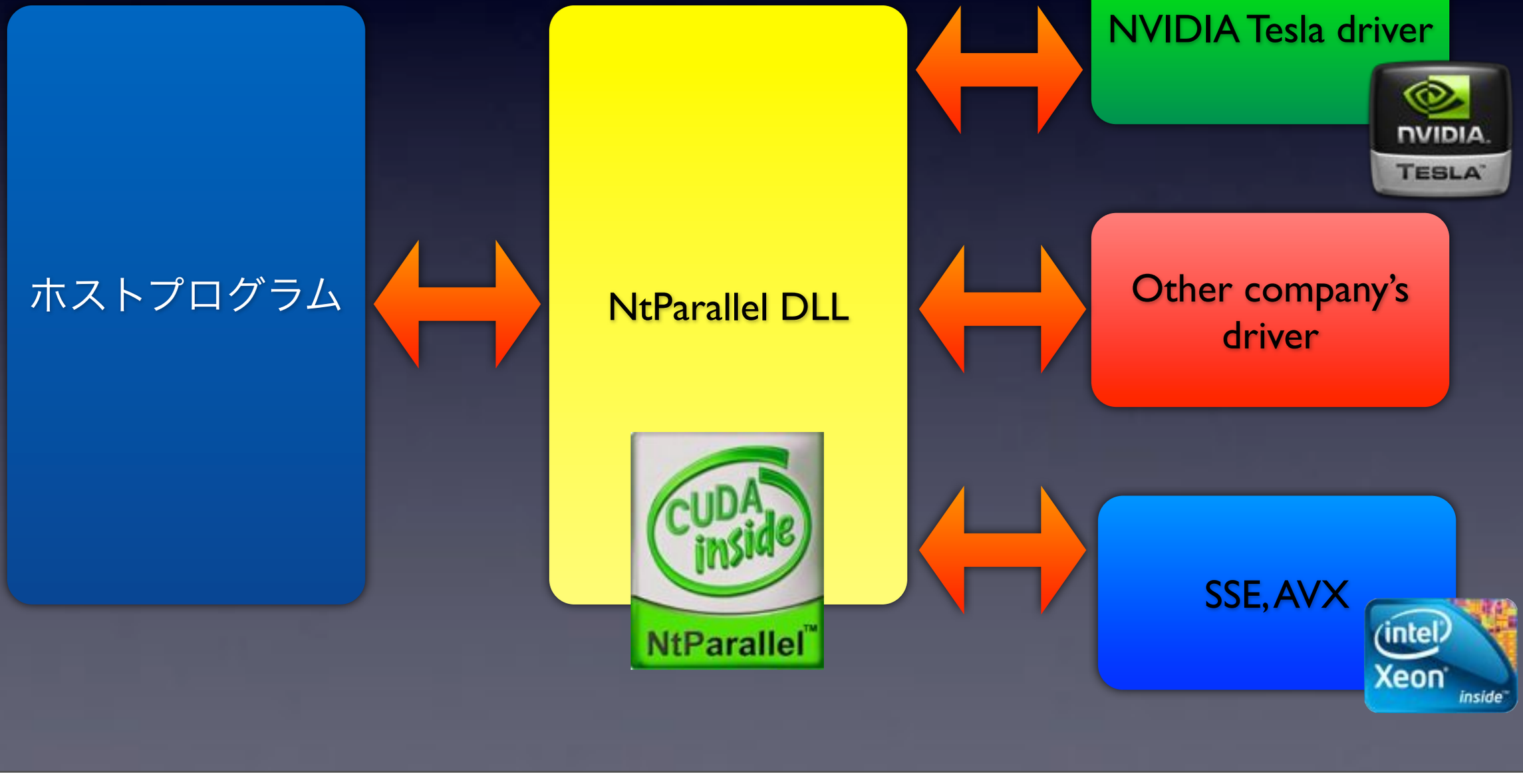
コピペ程度の作業でGPUコードを元のCPUコードに戻せる

GPUコードに高速化

簡単に元に戻せる

同じインターフェイスでNVIDIA以外のデバイス呼び出しが可能になる

技術的ヘッジ



Demonstration

Tesla C2050 vs. Xeon X5670

Scientific

1. Mandelbrot
2. Kirkwood Gaps
3. Wavelet Analysis

Wall St.

4. Binomial Tree Option Model
5. Black Scholes Option Model
6. Housing Loan Calculation

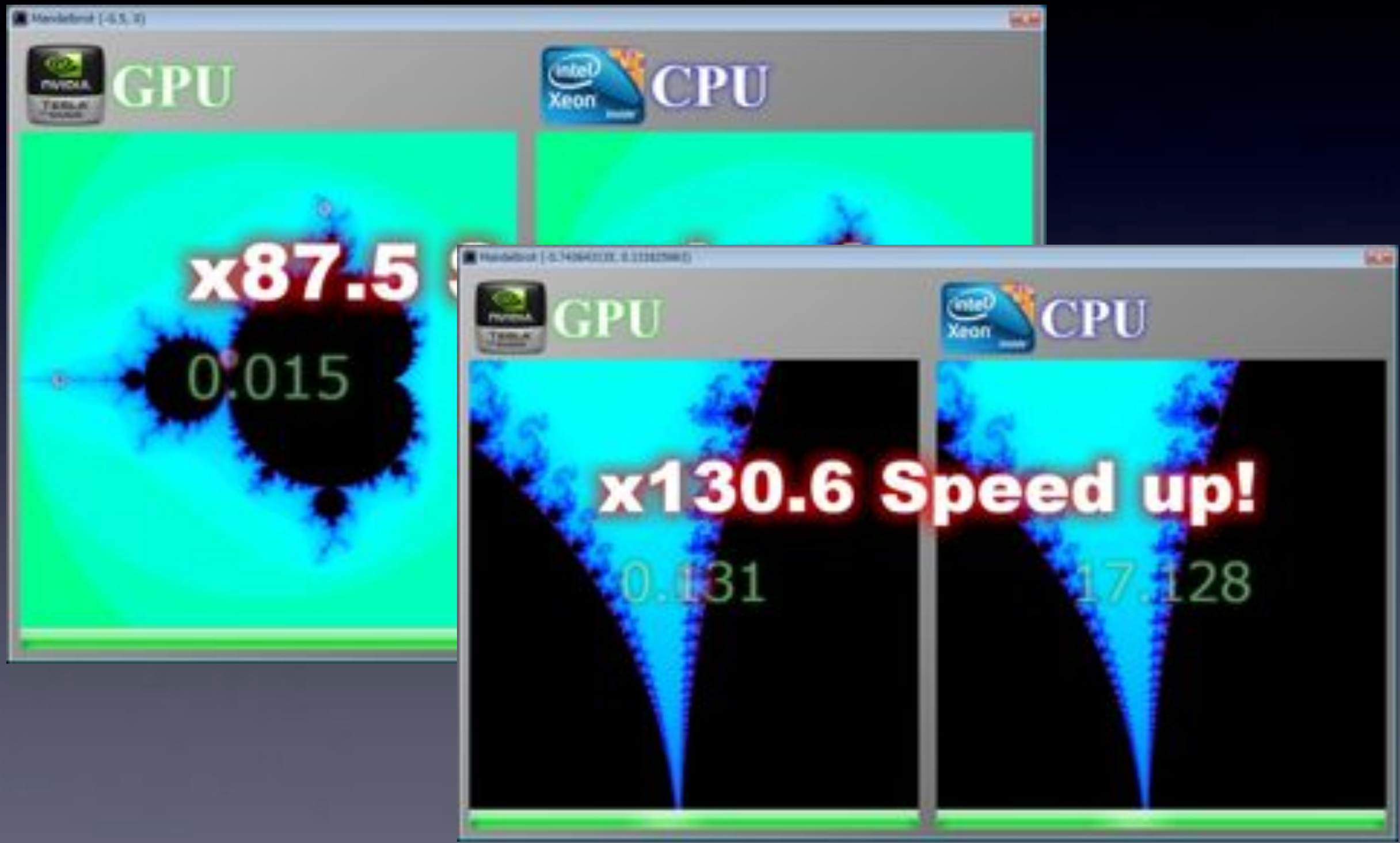
Boring

Accounting Stuff

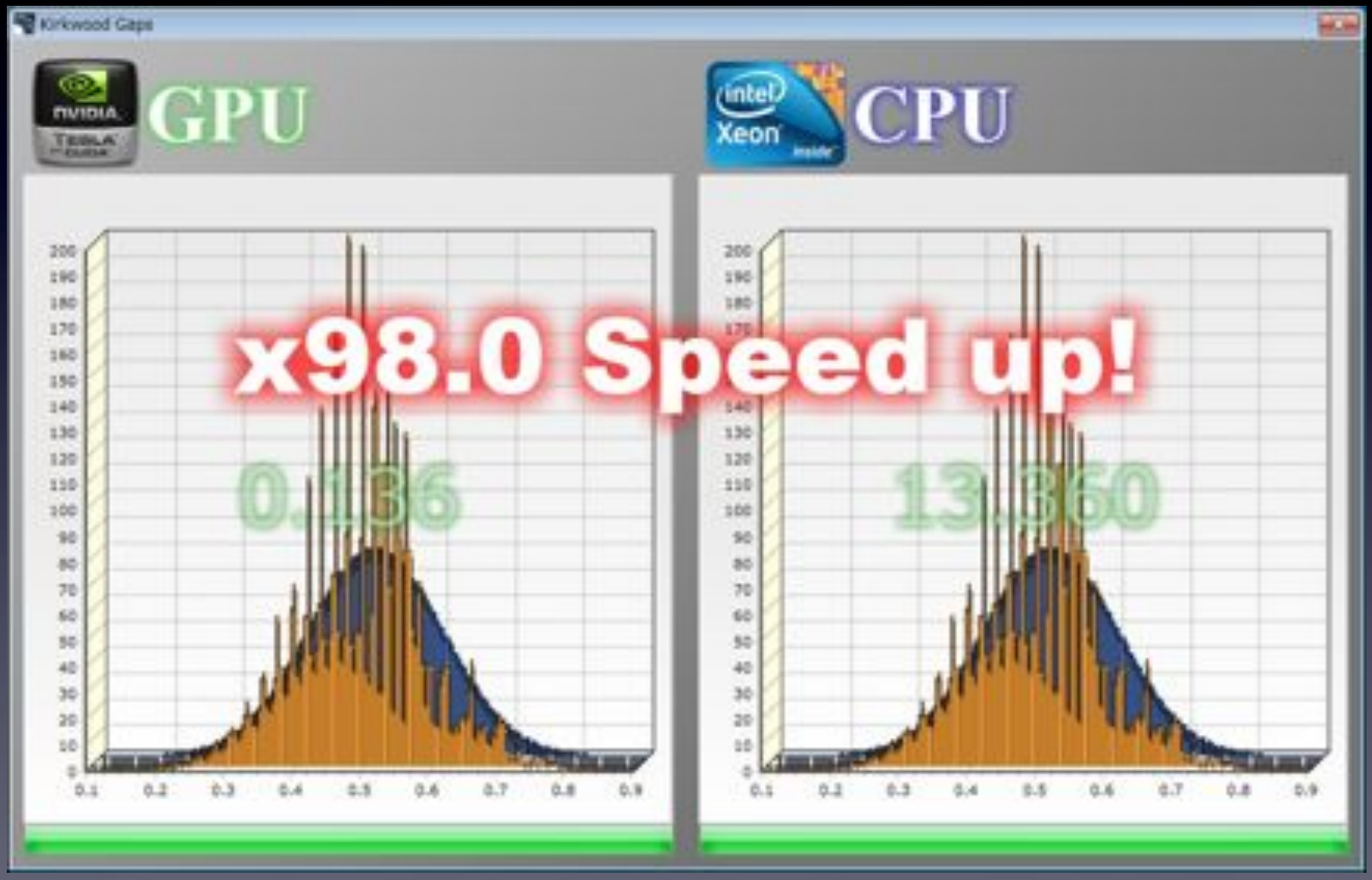
Test Machine: HP Z800/CT Workstation
- Windows 7 Professional 64bit
- Intel Xeon X5670/2.93GHz 6-core x2
- 24GB DDR-3 SDRAM/1333MHz, ECC
- 450GB 15Krpm SAS HDD

We compared single Tesla with single CPU thread code.

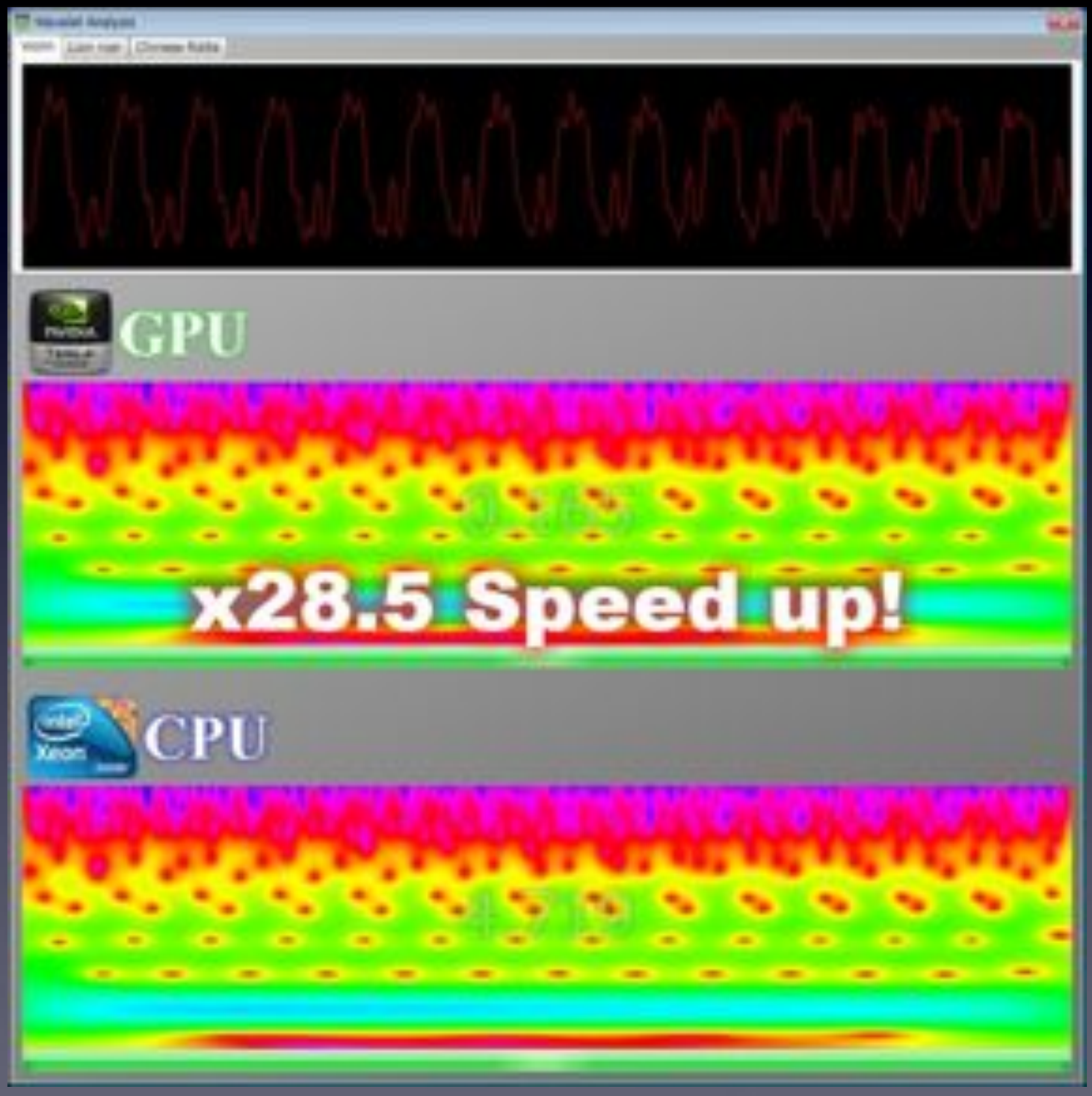
87-130倍速: Mandelbrot



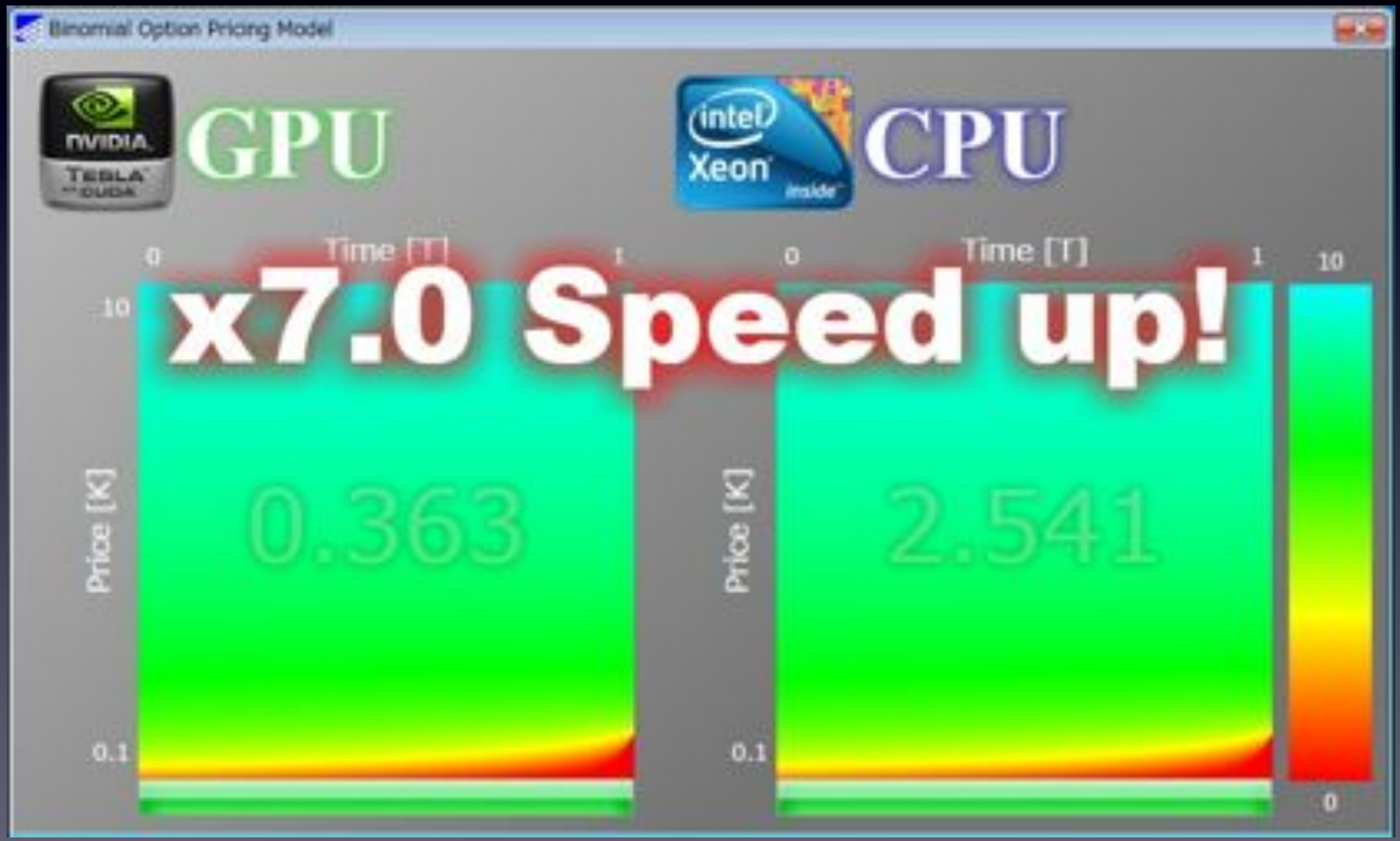
98倍速: Kirkwood Gaps



29倍速:Wavelet Analysis



7倍速: Binomial Tree Option Model



5倍速: Black Scholes Option Model

40倍速: 35年返済ローン

lambda expression方式

でもTeslaは十分に高速

速い理由

正しいスタートアップコード

GPU Computingでよくやりがちな間違いの防止

実行時コンパイルの採用

GPU側で動作させるプログラムをホスト言語からは文字列形式でAPIに渡し、nvccの代わりに実行時コンパイルしている。ここで最適化を行った。

デザインパターンの強制

nt_parallel_forの呼び出し方ではshared nothingな並列化しか書けない。逆に言えばそれ以外の用途に使いたければCUDAで書くしかない。

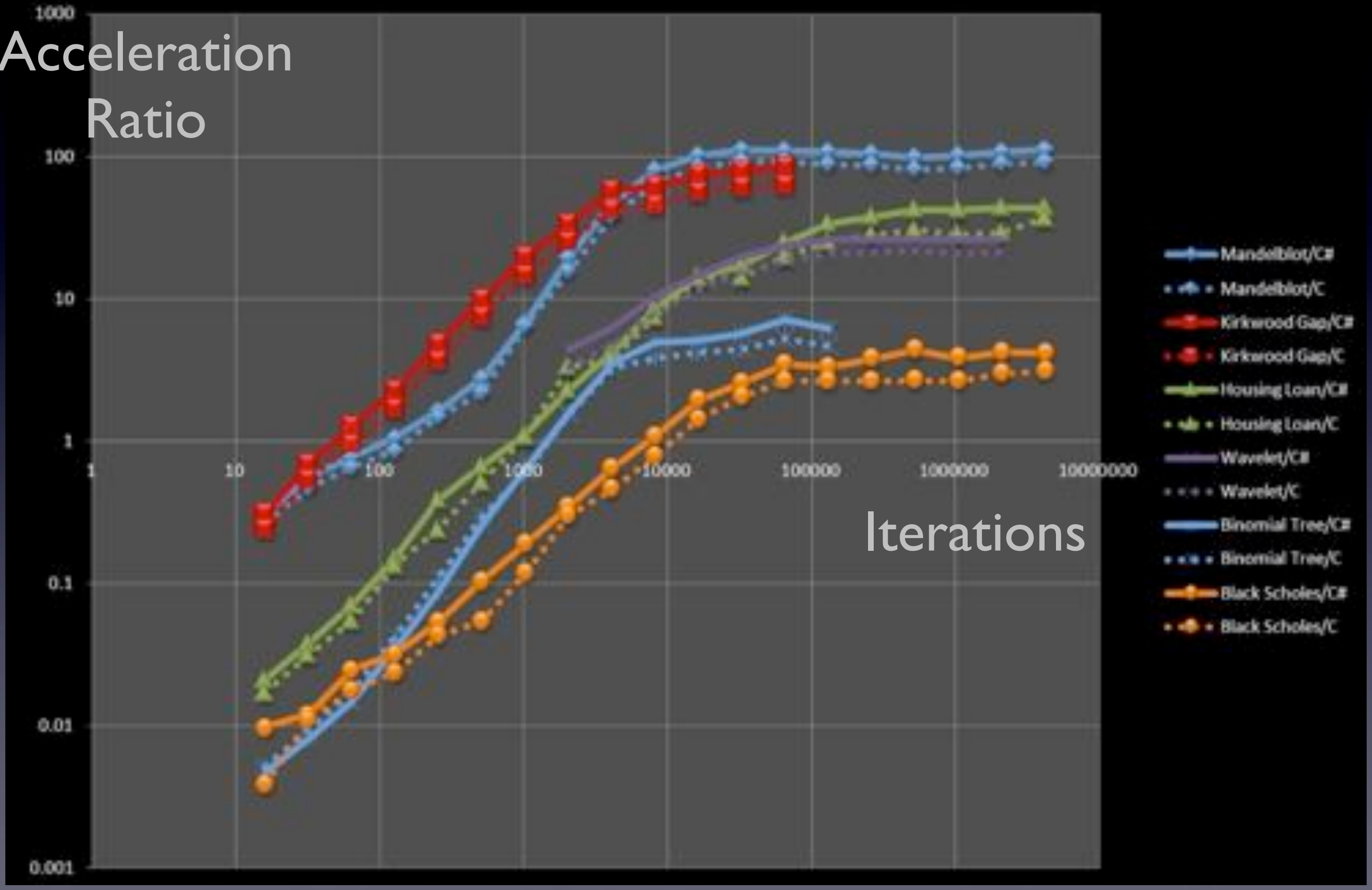
GPU Computingの普及を阻む壁はもう一つある


ビジネスは予算主義
事前に性能を見積もる方法論が必要だ
プログラムを作る前に性能を予想できないかな。
それが出来ないと予算をもらえないよ。

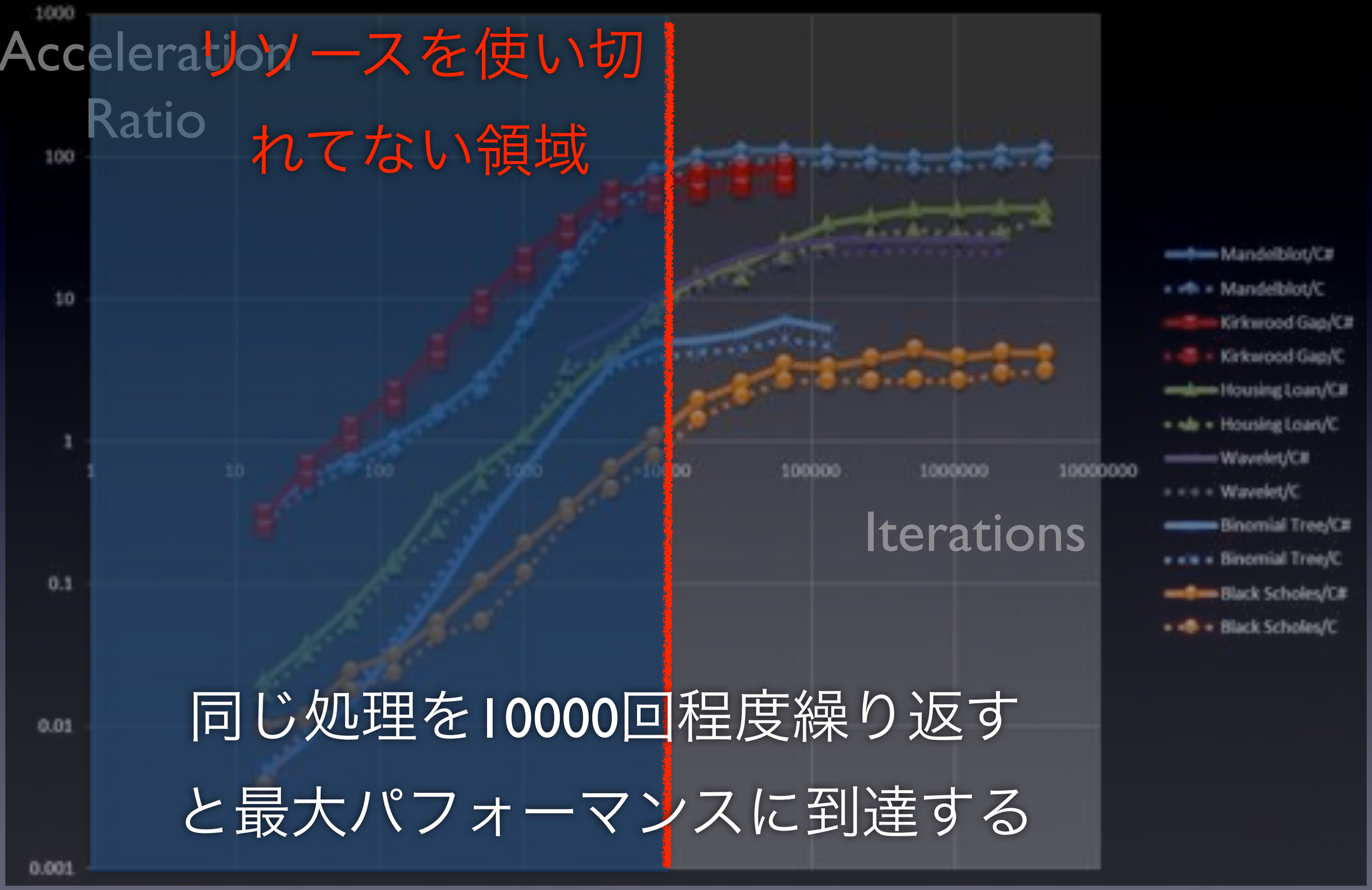
そこで、先程のベンチマーク

テストの結果を分析しよう

Relative Performance vs. for-loop Iterations

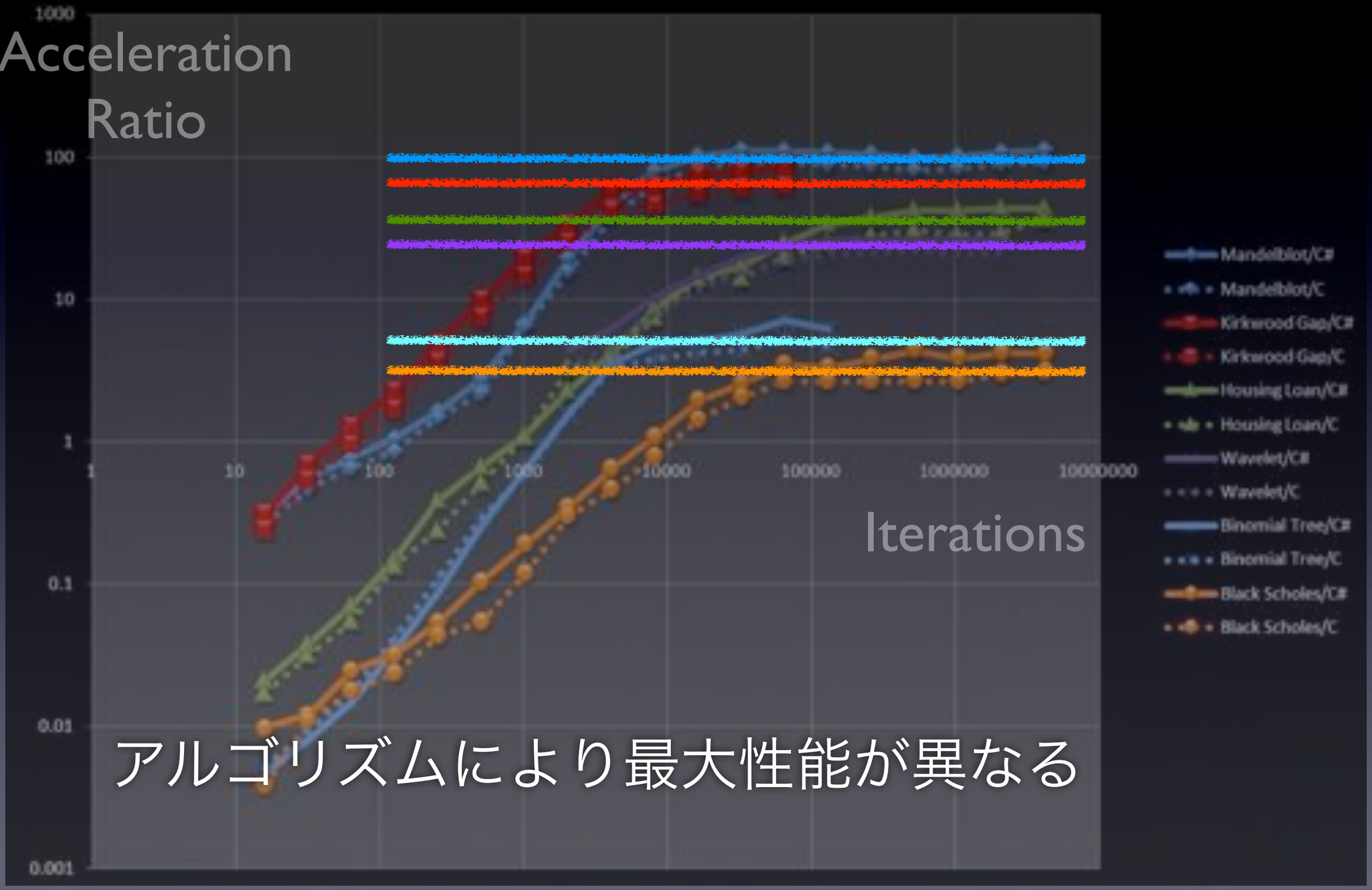


Relative Performance vs. for-loop Iterations



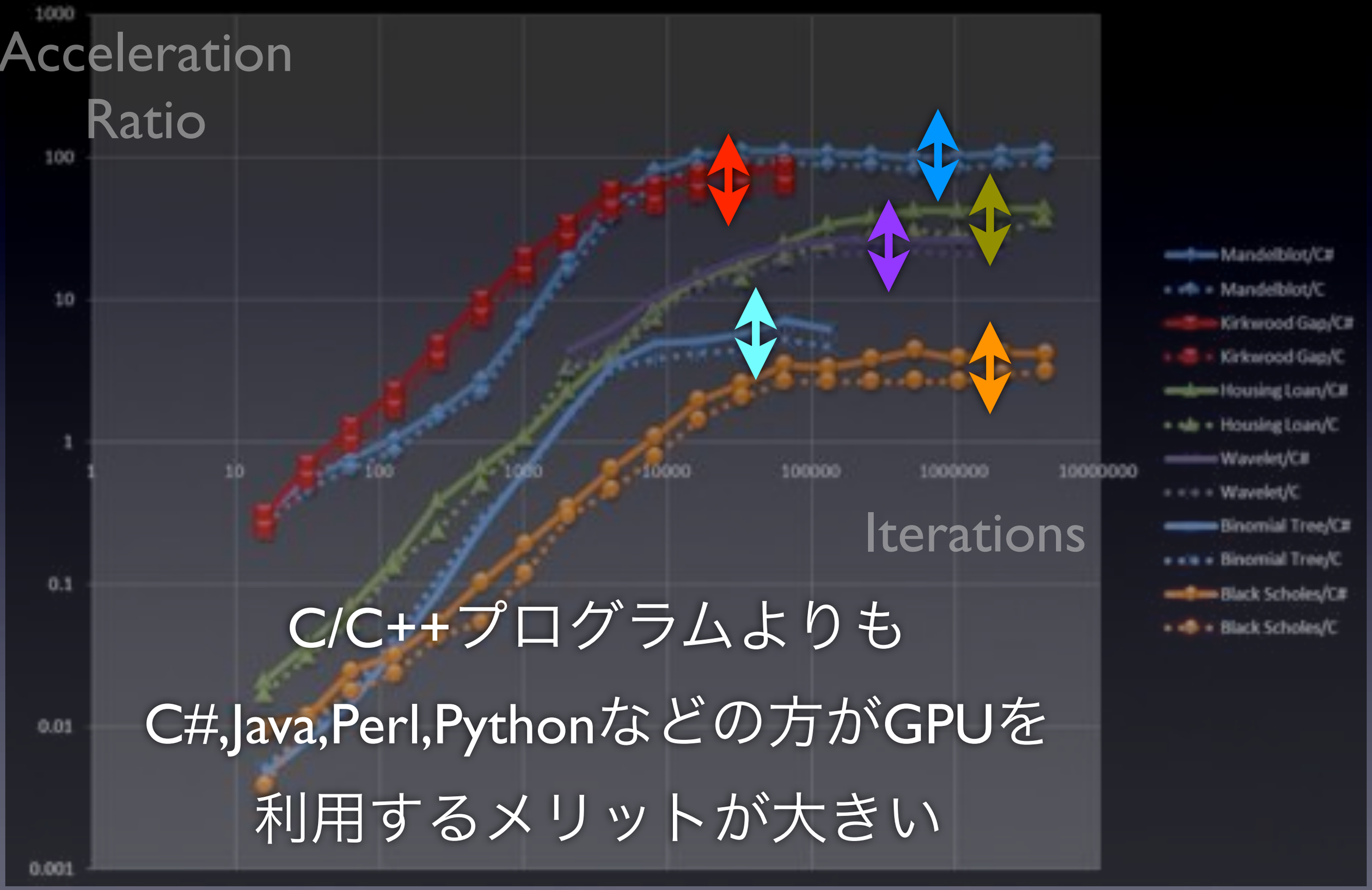
同じ処理を10000回程度繰り返すと最大パフォーマンスに到達する

Relative Performance vs. for-loop Iterations



アルゴリズムにより最大性能が異なる

Relative Performance vs. for-loop Iterations



C/C++プログラムよりも

C#,Java,Perl,Pythonなどの方がGPUを

利用するメリットが大きい


GPU Computing の
効率を握るカギは２つ

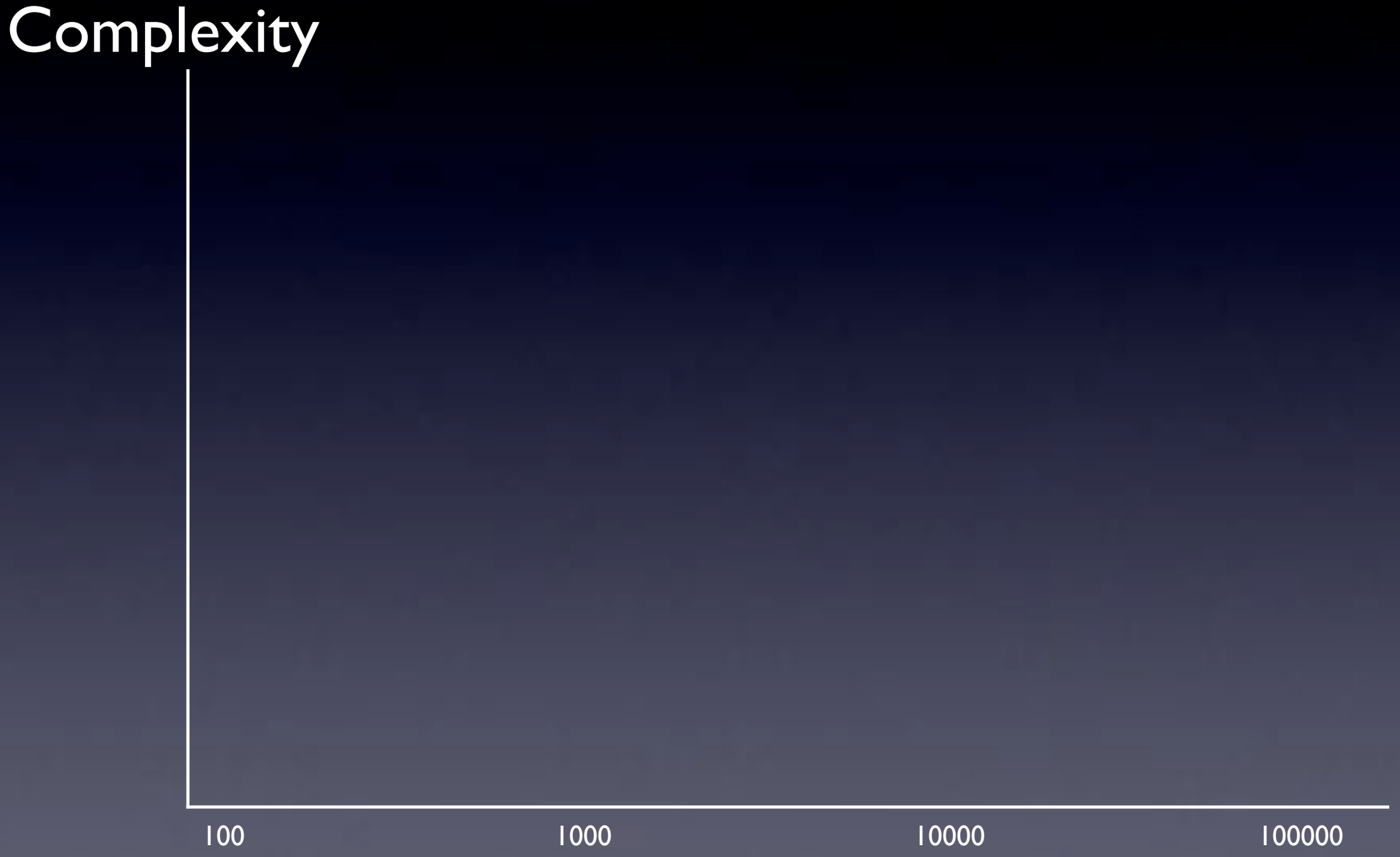
for-loop Iterations

同じ処理の繰り返しが多いほど性能が上がる。一般にTesla GPU 1枚あたり10000回程度の計算を行えば性能が最大化する。

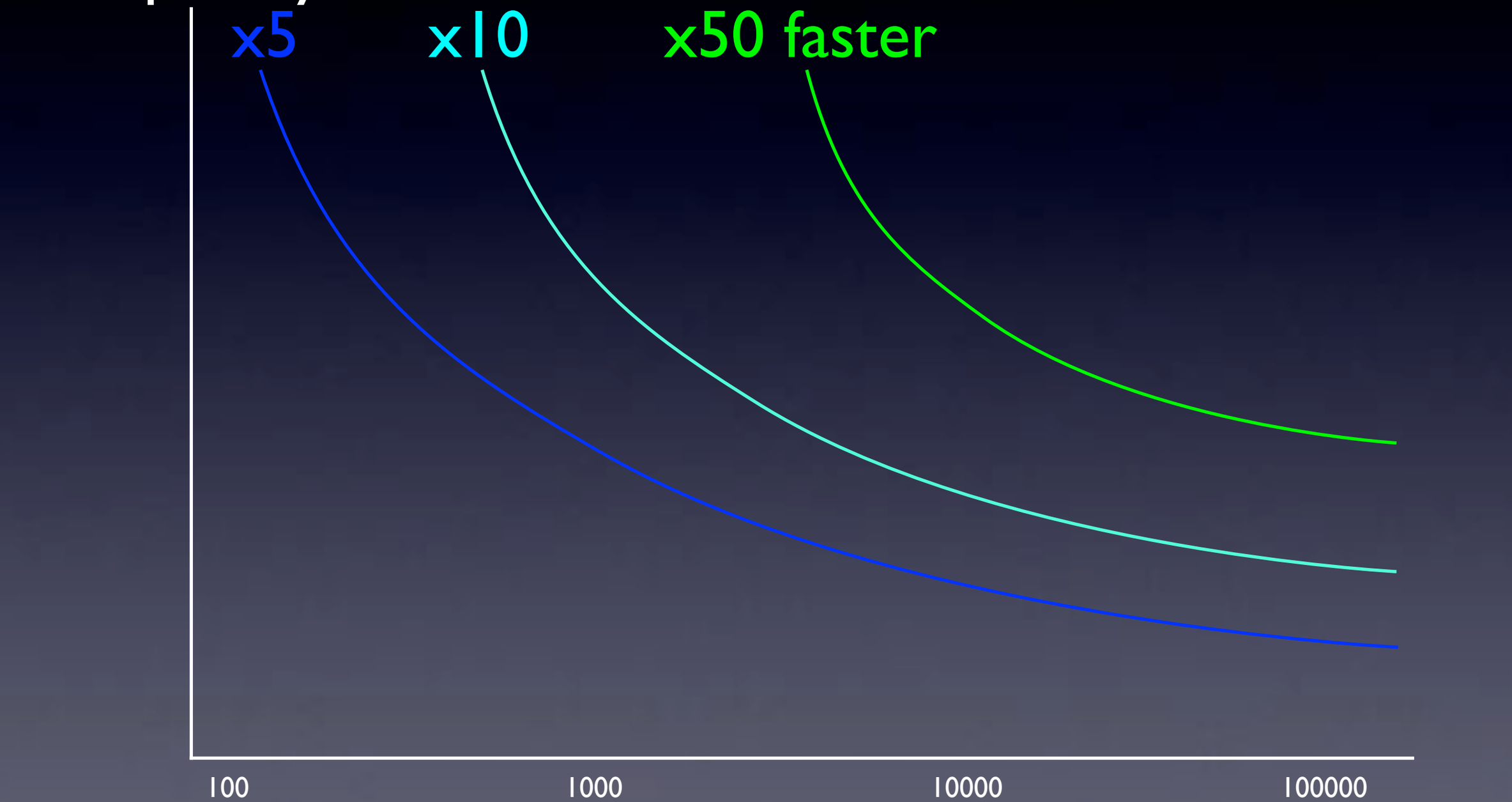
Complexity

Ops/Bytes比が高いほど性能が上がる。これは一般にプログラムが扱うデータ量に対してコードが長い状態を意味する。

この2つの指標を使って
チャートを作る

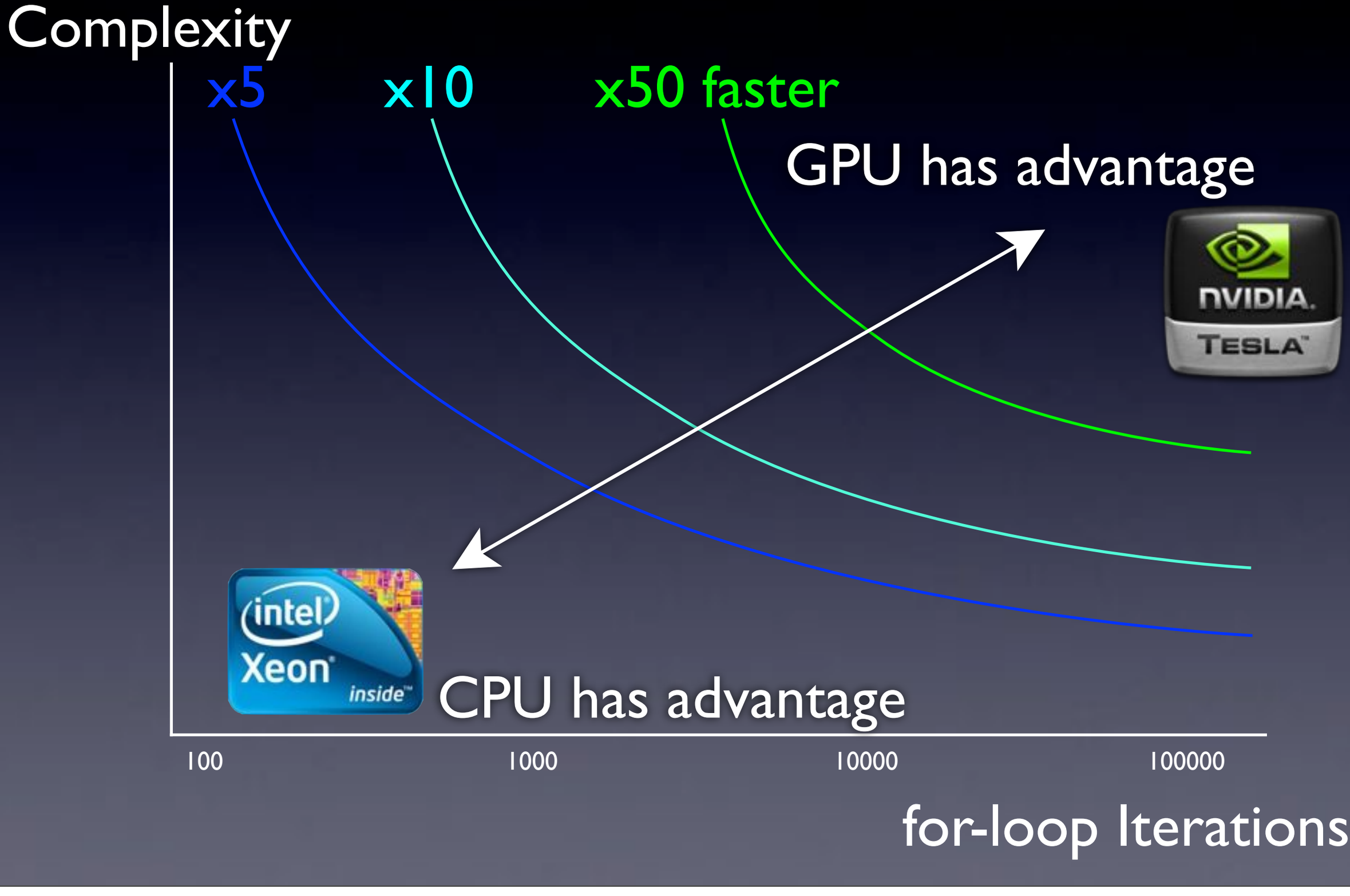


すでに見た通り、ComplexityとIterationsが大きくなるほどGPUのパフォーマンスが上がる



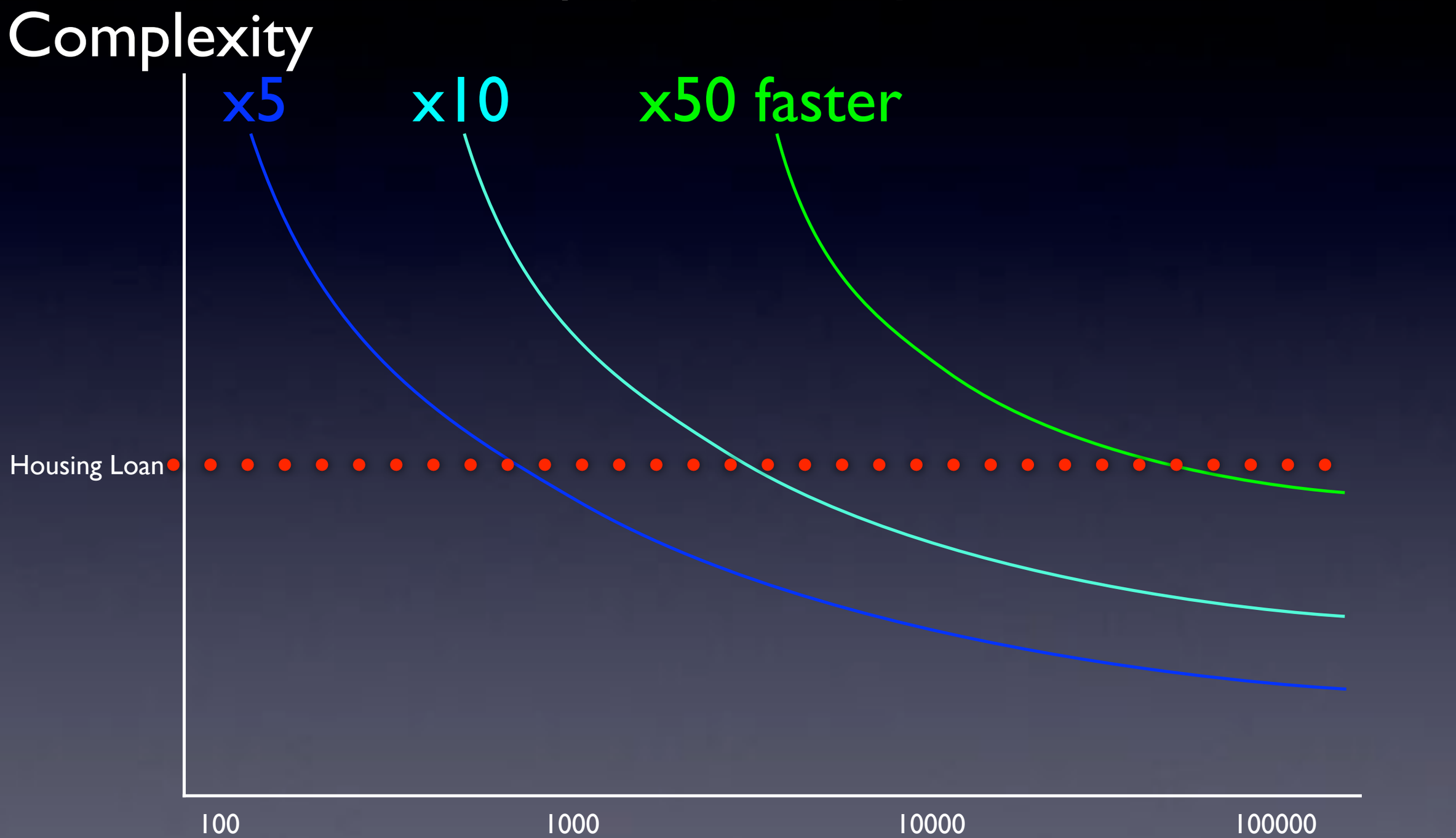
2ソケット8コアサーバーが安価に手に入る現在では、

10倍速を超えなければGPUを使う意味は乏しい

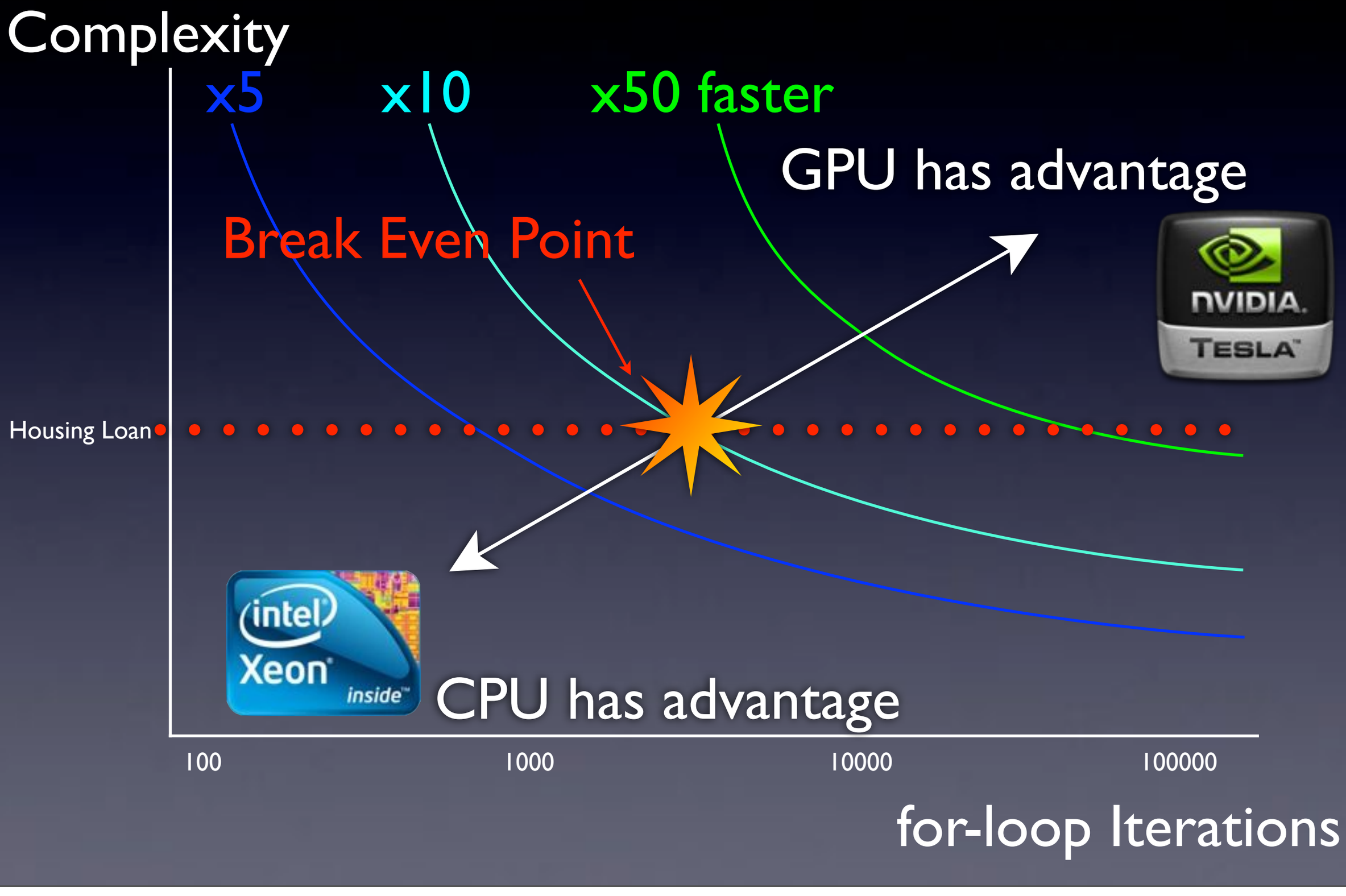


このチャートにアルゴリズム

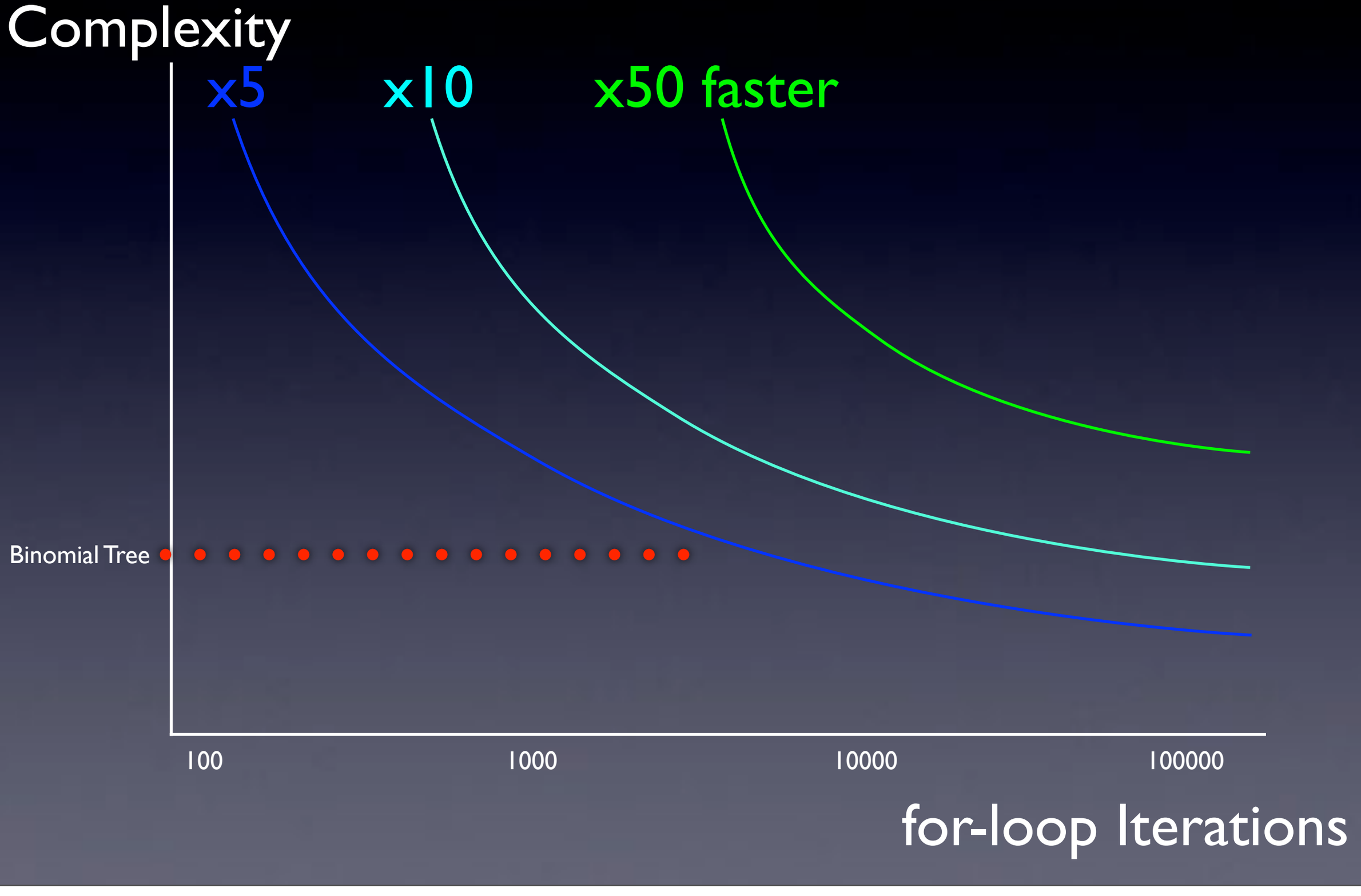
の性能線を引く



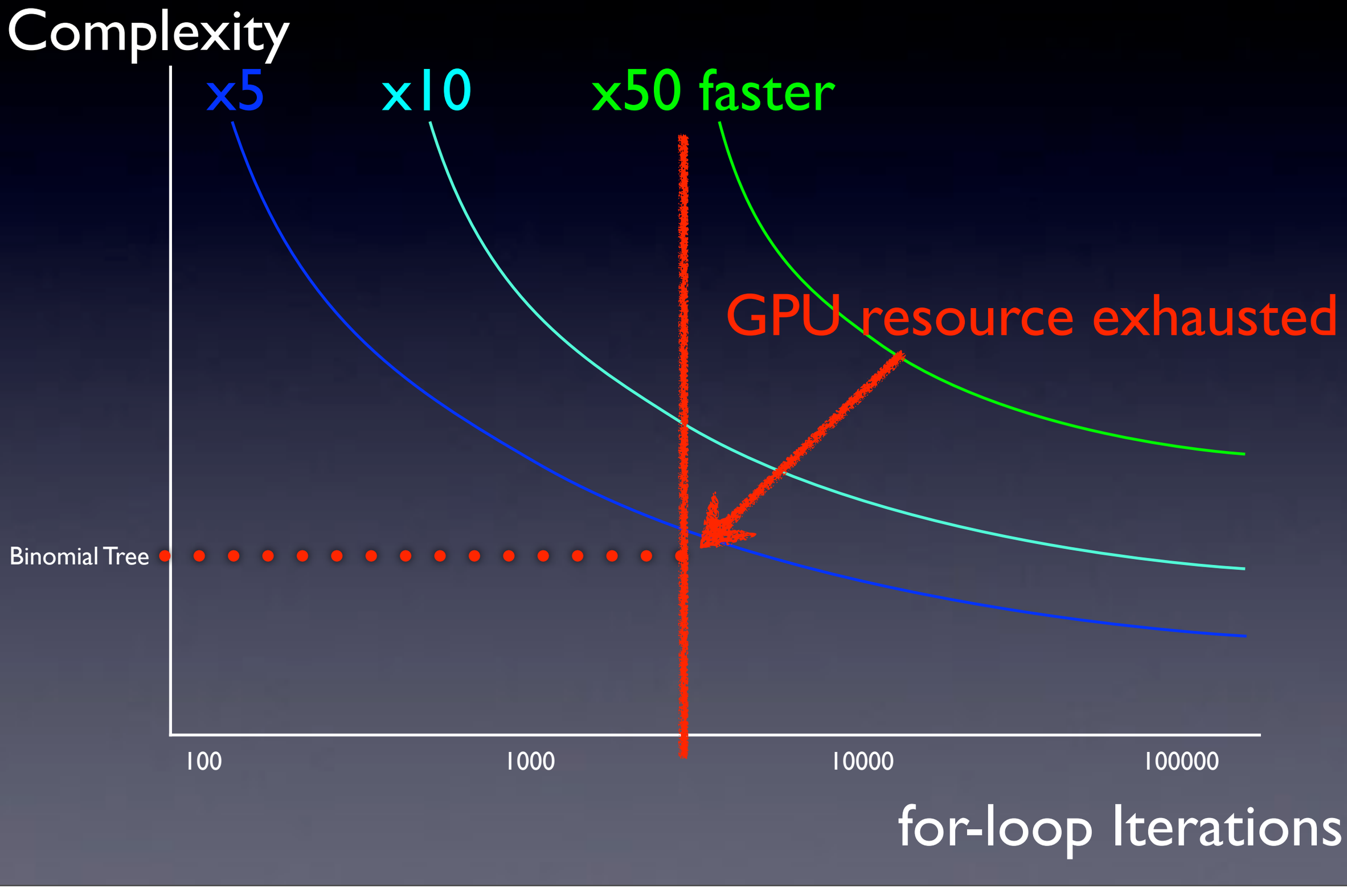
x10線との交点がこのアルゴリズムの採算点だ



性能が伸びないアルゴリズムもある

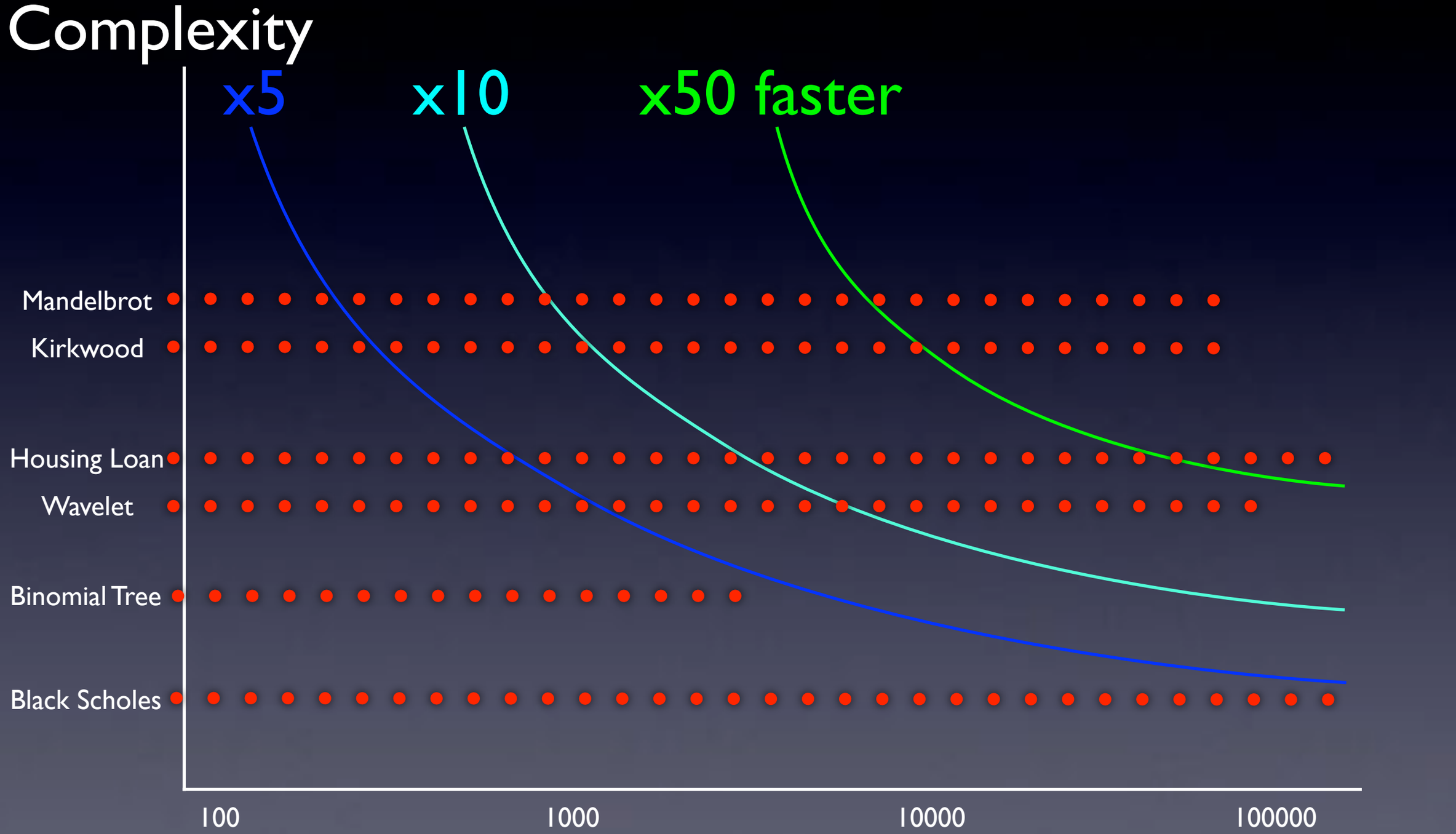


GPUのリソースを使い尽くしたのが原因だ



さらにいろいろな

アルゴリズムを並べてみよう



意外なことに、一見複雑に思われがちな金融デリバティブの計算よりも、単純なローン計算の方がGPUのパフォーマンスを出しやすい

x5: Black Scholes

x40: Housing Loan

単なる繰り返し計算の方がComplexityが高いからだ

```
...
string code = "
  [] (int i, double annualizedRate, int term, double monthlyPayment) => double {
      const int t = term;
      const double m = monthlyPayment;
      const double monthlyDf = 1.0 / (1.0 + annualizedRate / 12.0);
      int elapsedMonth;
      double val = 0.0;
      double df = 1.0;
      for (elapsedMonth = 0; elapsedMonth < t; elapsedMonth = elapsedMonth + 1) {
          df = df * monthlyDf;
          val = val + m * df;
      }
      return val;
  }
";
...
```

x40: Housing Loan

```
...
string code = "
  [](int i, double R, double Sigma, double S, double K, double T) => double {
      double rt = R * T;
      double sigmaSqrtT = Sigma * sqrt(T);
      double d = (log(S / K) + rt) / sigmaSqrtT + sigmaSqrtT * 0.5;
      return S * NormDist(d) - K * NormDist(d - sigmaSqrtT) * exp(- rt);
  }
";
...
```

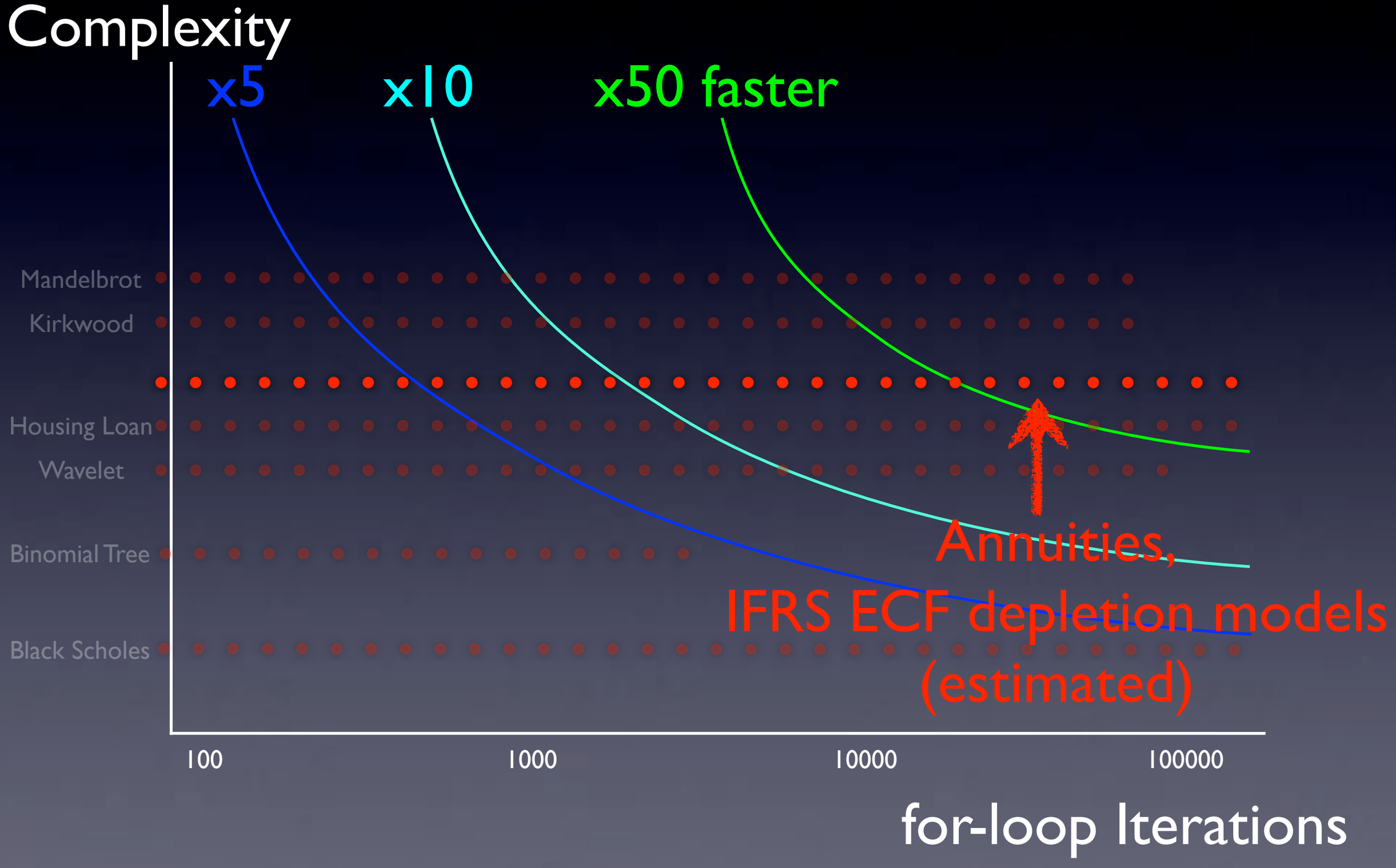
x5: Black Scholes

Q. あなたがGPUにやらせようとしてい
ることは、どれくらいのComplexity
とIterationsを持っているのか？

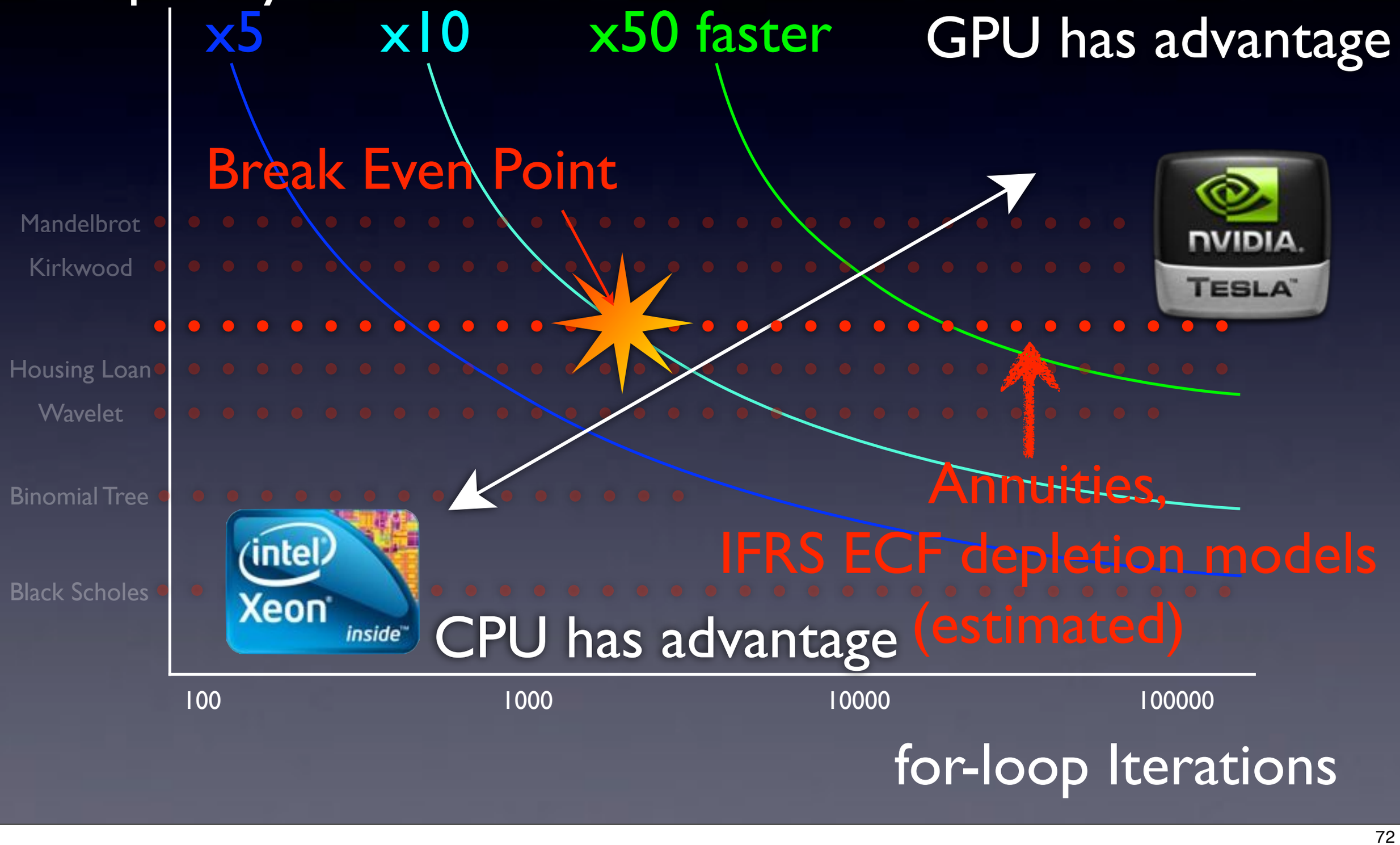
A. 類似しているアルゴリズムと比較しよう。例えばあなたが年金計算やIFRS ECF減損モデルに対応したローン評価をやりたいとする。ならば

Complexityは先ほどのHousing Loanよりも大幅に高い。

大体このあたりと予想がつく



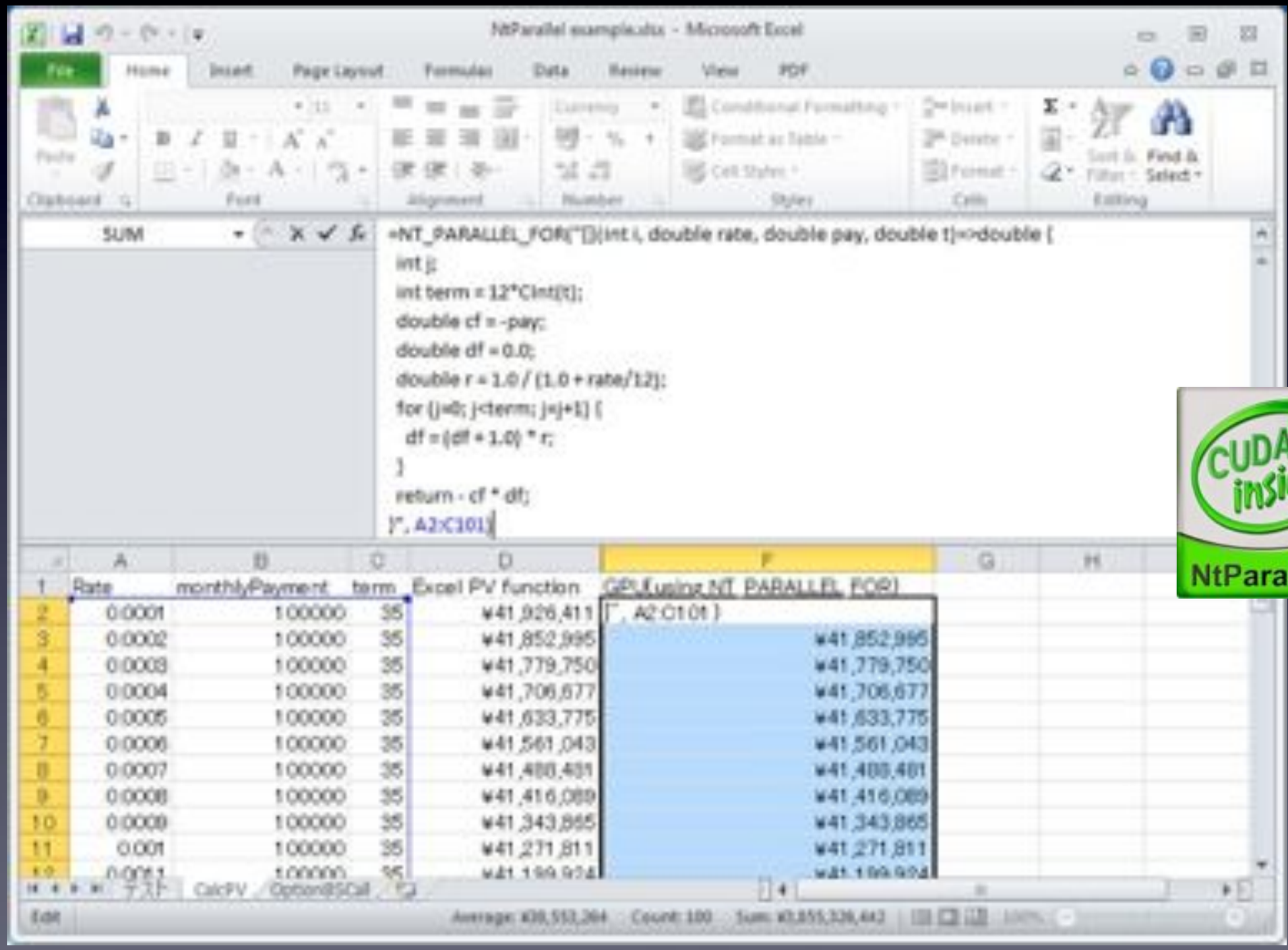
必要な処理件数がIterationsとx10線の交点を上回っていればGPUへの投資価値がある



このRule of Thumbを適用し
ようにも比較対象となるア

ルゴリズムがないならば...

nt_parallel_forを使って
実際に計測してみればよい



だから生産性が重要である

こうしてGPUの専門家がいらなくなった


GPU コンピューティング使ってみようか
話が噛み合った

この技術を我々の金融アプリケーションにも使っていきたい

nt_parallel_for手法も一般化していきたい。我々とともに働きませんか？

We want you!

http://www.numtech.com/