# 超大規模LSIを効率良く利用する超細粒度並列処理

(研究課題名：ハードウェア・プログラミングによる超細粒度並列処理)

「機能と構成」領域　井口　寧

## 要旨

VLSIチップ上のハードウェア回路は，各部分回路がそれぞれ独立に動作しており，本質的に並列処理を行なっています．そこで，数値計算やテキスト処理など，従来はC言語で記述されCPUによって実行されてきたソフトウェアを，内部回路が再構成可能なFPGA上に並列演算回路として実装できれば，演算子レベルの究極の超細粒度並列処理が可能となります．しかし，現在のハードウェア記述用C言語処理系をそのまま用いたのでは高い並列性を持つ回路を自動合成することはできません．そこで本研究では，プログラムの並列性を解析し，高並列演算回路を合成可能なCコードを出力し，VLSI上に高い並列性を持つ演算回路を合成することによって，超細粒度並列処理を行なうことを試みました．LSIチップのゲート量の制限下でプログラムの部分ごとの最適な並列度を求める手法を提案しました．本手法によって，電子透かし検出アルゴリズムを並列化回路として実装したところ，ソフトウェアに比べて大幅な処理速度を達成することができました．

## 1　研究のねらい

近年，VLSIの集積度は急速に向上しており，半導体の集積度に対する有名なムーアの法則では，およそ18ヶ月で2倍になると言われています．最新の半導体では1チップで約1億ゲート相当ですが，このまま集積度が向上すれば，5年後には10倍の10億ゲートのコンピュータチップが出現すると予想されます．これまでのところコンピュータのCPUチップの性能もゲート数の増加に伴って向上していますが，CPUチップ上の回路に占める演算器の回路量は，現在では既に全体の数%と非常に小さくなっていて，今後はCPUチップの回路量が増加しても，それに比例する処理性能の向上は困難だと考えられます．

一方で，多数のマイクロプロセッサを結合した超並列計算機は，自然科学分野におけるシミュレーションなどの多くの分野で，大規模な計算を高速に実行することができる計算機として盛んに研究され，研究並列処理に関する技術が蓄積されてきました．しかし従来のCPUを結合した超並列システムは，CPUチップ内の処理速度に比べてチップ間の通信速度が非常に低速なため，データや処理の分割の粒度が非常に粗くなり，効率良く並列実行可能なアプリケーションが

限られるという問題があります.

そこで本研究では，これまでの超並列処理技術を VLSI チップの回路設計に適用することによって，VLSI チップ内の大規模な論理回路上に高並列な演算回路を合成し，VLSI の論理回路量の増加を処理性能の向上に結び付けるための手法について研究しました．近年では，ハードウェア回路の設計手段として，C 言語に近い記述で論理回路を設計可能な C レベル設計ツール等が開発され，利用されるようになっています．しかし，これらの処理系は通常コンパクトな回路を合成するための最適化がなされているため，そのままでは高い並列性を持った回路を合成することは困難です．そこで本研究では，元のプログラムに含まれる並列性を解析・抽出し，C レベル設計ツールが並列演算回路を合成できるように，ディレクティブ (並列化指示) を挿入したりループを展開することによって，ソフトウェア・アルゴリズムを高い並列性を持つ演算回路として展開するアプローチを採りました．プログラムは，ディレクティブやループ展開などによって並列性を陽に含む記述に変換された後，C レベル設計ツールや配置配線ツールによって並列演算器を含むハードウェア回路として展開できるので，VLSI チップ上に並列演算回路を容易に構築できます．超並列計算機での並列性解析の知見を大規模 VLSI 上での回路設計に導入することによって，将来の大規模 VLSI チップのための超細粒度並列処理の実現を試みました.

## 2 研究方法と成果

### 2.1 処理の流れ

図 1 は，本研究における超細粒度並列処理の処理の仕組みです．実験用 VLSI として，FPGA 素子を利用しました．FPGA は，フラッシュメモリと同様な感覚で，ユーザーが使用時に内部回路をプログラムできる素子であり，ユーザーは任意の回路を繰り返し LSI チップ内に構成することができます．本研究では，C 言語で記述されたソフトウェアを入力とし，そのソフトウェアを並列化演算回路として展開し，FPGA 上に実装しました．プログラムごとに要求される内部回路が異なるので，プログラムの実行の都度回路合成を行い，FPGA の内部回路を指定するコンフィギュレーションファイルを得て FPGA にダウンロードし回路をスタートさせます.

例えば，加算を多く実行するプログラムが与えられた時，加算回路を高い並列度で合成すれば，このプログラムは高速に実行されることが期待できます．別のプログラムが与えた時，このプログラムが乗算を多く実行するのであれば，高並列の乗算回路をチップ内に合成することによって，このプログラムを高速に実行する回路が合成できます．このように，プログラムに内在する演算の種類や量，並列性を分析し，プログラムに最適な回路をその都度合成すれば，大容量の FPGA を用いてソフトウェア・プログラムをハードウェア回路として実行できます.

FPGA の規模は年々急速に増加しており，最近では 1 千万ゲート規模のものが発表されています．しかしながら，これらの大規模なチップを用いたとしても，複雑なプログラム全体を 1 つの FPGA チップ上に搭載することは困難です．一方でプログラムはコードの数%の部分が処理

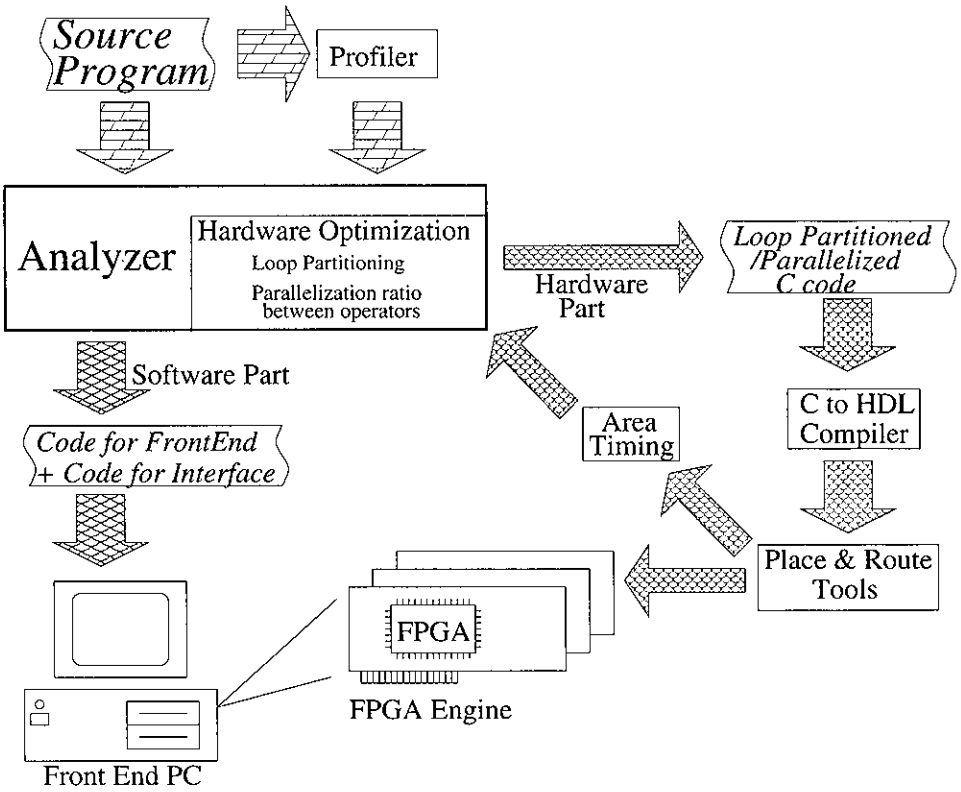


図 1: 処理の流れ

時間の大部分を占めるケースが多くあることが知られています．そこで，元となるプログラムを，コード量は少ないが多くの処理時間を消費する部分と，コード量は多いが実行時間が短かい部分に分割します．前者は，ハードウェア実装用の C レベル設計ツールで論理回路に展開された後，FPGA 上に実装され，並列化回路として高速実行されます．後者については，そのまま Front End PC 上の CPU によって，ソフトウェアとして実行されます．

## 2.2 並列実装法

次に，C 言語のソースコードと合成される演算回路の並列性について，概念図を用いて説明します．図 2 は逐次演算回路を合成する C レベルのソースコードと合成される回路の例です．図 2 のコードはループの繰り返しに対して独立しているので，各繰り返し内の演算は同時に独立して実行できます．しかし，既定の状態では C レベル設計ツールは図に示すような逐次実行回路を合成し，プログラムが本来持つ並列性を引き出すことはできません．

その一方でハードウェア記述用の C レベル設計ツールは，多くの場合並列な演算器を合成するための手段が用意されています．例えば，“par” ディレクティブを指定すると，par 文に続くブロック内の回路が並列化された回路として合成されます．そこで，この par ディレクティブを用いることによって，演算回路を並列化することを試みます．この例では，ループの繰り返しに

```
for( i = 0; i < N; i++ ){
    a[i] = b[i] + 1;
    x[i] = y[i] * c;
    z[i] = x[i]/a[i];
}
```

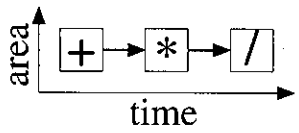


図 2: 演算の逐次実行

```
for( i = 0; i < N; i+ = M ){
  par{
    a[i] = b[i] + 1;
    a[i + 1] = b[i + 1] + 1;
    a[i + 2] = b[i + 2] + 1;
    ...
    a[i + M − 1] = b[i + M − 1] + 1;
    x[i] = y[i] * c;
    x[i + 1] = y[i + 1] * c;
    x[i + 2] = y[i + 2] * c;
    ...
    x[i + M − 1] = y[i + M − 1] * c;
  }
  par{
    z[i] = x[i]/a[i];
    z[i + 1] = x[i + 1]/a[i + 1];
    ...
    z[i + M − 1] = x[i + M − 1]/a[i + M − 1];
  }
}
```

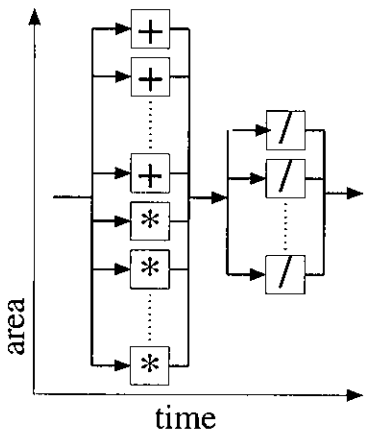


図 3: 演算の並列実行

対して独立なので，まずループ内を $M$ 個づつの行に展開します．すると，展開された行は同時に実行できることが分かります．次に展開された行の前後に，図 3 のような par ディレクティブを挿入することによって，展開された部分が並列演算回路として合成されます．この例では，$M$ 個づつの加算器と乗算器，および $M$ 個の除算器が合成されます．順序関係については，この例のように除算は前の加算と乗算の結果を用いますから，加算と乗算は 1 つの par ブロックで展開できますが，除算は加算と乗算の par ブロックの後で実行する必要があります．

### 2.3 実装方式

従来の超並列計算機で用いられてきた OpenMP や HPF などの超並列処理言語では，効率の良い並列化を支援するために，データや処理を分割するためのディレクティブが用意され，コン

パイラによる自動並列化の他に，ユーザーが並列化を指示することができます．本研究では，このようなソフトウェアにおける並列処理言語の様に，C レベル設計ツールに対するディレクティブを挿入するアプローチを取りました．現在，ハードウェア記述用の C レベル設計ツールの多くは，実装コスト低減のため，プログラムをコンパクトなハードウェア回路とすることに主眼が置かれていて，ある程度は並列化できるものの，能動的に並列化を行なうことはできません．図 1 の Analyzer は，与えられたプログラム中の並列性を抽出して並列性が陽に記述された C レベル記述に変換します．変換された並列化 C レベル記述は，既存の C レベル設計ツールによって並列演算回路に合成され，FPGA 上で実装実行されます．

## 2.4 並列化の戦略

各行が並列化可能だとしても，論理回路量には制限があるので，与えられた回路量の範囲内で最も効果的に並列化する必要があります．つまり，並列化しても回路量の増加が少なく処理速度向上の効果が大きい部分の並列度を高くする一方，並列化すると回路量が大きく増加する割には処理速度向上の効果が少ない部分は並列度を低くします．プログラムの各部分の並列度を決定するためには，以下の手順に従ってプログラムを分析します．

**Step1: ハード化部分の選択** 最初にソフトウェアとしてプログラムを実行し，関数ごとに実行時間の割合いを求めます．この分析で処理時間を多く消費すると判定された関数が，ハードウェア並列化の対象となります．

**Step2: 逐次版での実行速度/面積評価** 次に，行ごとに適切な並列度を決定します．コードプロファイラを用いて行ごとの実行時間を求めます．行 $i$ の実行時間を $T(i,1)$ と置きます．$T$ の最初の引数は行番号，2 つ目の引数は並列度で，逐次版なので並列度は 1 となります．また，対象となっている関数全体の実行時間を $T_{seq}$ と置きます．

C レベル設計ツールを用いて，逐次版の関数を論理回路に展開した時の使用論理回路量を求め，$A_{seq}$ と置きます．

**Step3: 試行的並列化によるデータ収集** 行 $i$ をある並列度 $P_t$ で並列化し，使用回路量を評価します．この時の論理回路量を $A(i,P_t)$ と置くと，行 $i$ の 1 並列当りに要する論理回路量が $\Delta A(i) = (A(i,P_t) - A_{seq})/(P_t - 1)$ と記述できます．

**Step4: 並列化後の実行時間/面積の推定** これらの情報を元に，各行の並列化後の実行時間と面積を推定します．行 $i$ を $P_i$ 並列で実装した時の推定実行時間 $T(i,P_i)$ および面積 $A(i,P_i)$ は次式で推定します．

$$
\begin{aligned}
T(i,P_i) &= T_{seq} + T(i,1)(1-P_i)/P_i \\
A(i,P_i) &= A_{seq} + \Delta A(i) \cdot P_i
\end{aligned}
$$

表 1: 実装環境

| | |
|---|---|
| FPGA ボード | celoxcia 製　RC2000 |
| 搭載 FPGA チップ | Xilinx 製　xc2v8000-4×2 |
| ホストと FPGA ボード間通信 | PCI バス |
| ホスト CPU | Pentium4 2.8GHz |
| ホストメモリ | 1GB |
| 回路開発環境 | DK-version3 |
| 回路開発言語 | Handel-C |

**Step5: 他の行の評価** Step3, Step4 の手順を他の行についても適用し，各行ごとの並列度の効果を求めます.

**Step6: 行ごとの最適並列度の決定** 計算時間を多く消費する各行に対する並列化後の実行時間と面積の推定結果より，どの行を何並列するのが最適なのか求めます．これは，

$$\sum_i \Delta A(i) \cdot P_i + A_{seq} \leq A_{chip}$$

の条件下で

$$\sum_i T(i, P_i)$$

を最小にする $P_i$ を求めることによって，各行ごとの最適な並列度を算出できます.

## 2.5 実験プラットホーム

本稿での実装環境を表 1 に示しました．RC2000 ボードには，800 万ゲート相当の Xilinx 製 xc2v8000-4 と，SRAM が 4MB×6 バンクが搭載されています．C レベル設計ツールとして DK3 を利用しました．DK3 は，Handel-C と呼ばれる C に似た言語であり，“par” などの並列化ディレクティブを挿入することにより，容易に並列化回路が合成可能です．また，配置配線ツールとしては， Xilinx 社の ISE を用いました.

ソフトウェアでは，・ツールとして，前節での行ごとの実行時間の分析に PGI 社の pgprof，並列展開を支援するツールとして HP 社の KAP を用いました．これらは，超並列計算機向けの並列化支援ツールであり，このままではハードウェア並列化に適用することはできません．本研究では，ハードウェアの並列化を目指していますが，従来の並列化処理技術の助けを得ながら，並列化演算回路の合成を行ないました.

```
1: int WM_detect(int y[], int w[][]){
2:     foreach d (all key){
3:         Δw[d] = Σ^N w[d][i];
4:         S_d = Σ^N y[i]w[d][i];
5:         T_d = Σ^N (α|y[i]|w[d][i])w[d][i];
6:         r_d = (S_d − (Δw[d]/N)|S_d|)/T_d;
7:         if( r_d > 0.5 ) return( d );
8:     }
9: }
```

図 4: 電子透かしの検出アルゴリズム

## 2.6 電子透かし検出への応用

### 2.6.1 音楽向け電子透かし

本並列化実装手法の実用的な応用として音楽向け電子透かし検出プログラムに適用し，性能向上を確かめました．電子透かしは，デジタル化著作物の著作権保護や流通の制御を可能とする技術として，近年注目されている技術です．しかし，従来のソフトウェアによる検出では検出速度が非常に低速で，数百 Mbps でインターネットを流通するデジタル化著作物の著作権制御には適用できない問題がありました．そこで，本並列化手法を適用して，電子透かし検出を並列化されたハードウェア回路として実装することにより，音楽ファイル中の電子透かし検出速度の大幅な向上を試みました．

電子透かし技術は，著作者の情報などのビット列を人間が知覚できないようにデジタル化著作物に埋め込む方法です．電子透かしの検出には様々なアルゴリズムが提案されていますが，本実験では，音楽向けアルゴリズムのうちハードウェア化に適した，乗算が少なく時間領域で検出処理を行なうアルゴリズムを採用しました．この方式では，透かしが埋め込まれた音楽データと，検出すべきビット列 (透かしデータ) を入力すると，その音楽データに透かしが埋め込まれているか否かを判定することができますが，この判定は透かしごとに行なう必要があるため，ソフトウェア処理では通常負荷が非常に大きくなります．しかし，この問題は，それぞれの透かしは独立しているので，並列化が容易であり，本手法を適用して合成される並列化検出回路によって処理速度の大幅な向上が期待できます．

### 2.6.2 検出アルゴリズム

採用した検出アルゴリズムの核となる部分を図 4 に示します．$y[i]$ は透かしが埋め込まれた音楽の時系列値，$w[d][i] = \{+1|-1\}$ は埋め込んだ透かし $d$ の時系列値，$\alpha$ は定数です．また，時

表 2: 透かしの検出速度と回路規模

| | ハードウェア検出 | | ソフトウェア検出 | |
|---|---|---|---|---|
| | 検出時間 | 検出速度 | 検出時間 | 検出速度 |
| 25 種類の透かし検出 | 500ms | 326.4Mbps | 5,807ms | 28.10Mbps |
| 50 種類の透かし検出 | 500ms | 326.4Mbps | 11,341ms | 14.39Mbps |

表 3: 回路量とクリティカルパス

| | LUTs | Number/total(%) | critical path |
|---|---|---|---|
| 25 並列の検出回路 | 29,485 | 31% | 24.696ns |
| 50 並列の検出回路 | 57,135 | 61% | 24.740ns |

系列は $N$ サンプルごとに区切られているものとします．図 4 の手順の結果，$r_d$ が 0.5 以上であれば音楽データ $y[i]$ は透かし $d$ を含むと判断されます．透かしは通常複数であり，複数の透かしを検出する場合は 3 ～ 6 行目の検出処理を複数の透かし $d$ について行ないます．一見して分かるように，本アルゴリズムは $d$ に対して独立であり，容易に並列性を抽出することができます．

### 2.6.3 実装結果

図 4 の音楽向け電子透かし検出アルゴリズムを本手法によって合成した並列化検出回路の実行速度を表 2 に示しました．用いた実験環境は，表 1 の環境と同一です．この例では，44.1H で PCM 録音された 20.4MByte の wave ファイルの左チャネルに透かしを埋め込み，この音楽ファイルを入力とした時の検出速度を評価しました．

ソフトウェアによる検出では，透かしの種類の数だけループを繰り返す必要があるため，透かしの種類が増加すると検出速度が低下します．これに対してハードウェア並列化された検出回路では，透かしの種類が増えても，それぞれの透かしのための検出回路は，チップ内に独立した並列化回路として合成され，これらが同時に動作しますので，透かしの種類が増えても検出速度は低下しません．表 4 より，50 種類の透かし検出時では，並列化されたハードウェア回路はソフトウェアに比べておよそ 23 倍の検出速度を達成できたことが分かります．

表 3 には，合成された回路の使用ゲート量，チップ上の全ゲートに対する使用ゲート量の割合，およびクリティカルパス長を示しました．合成された回路は C 言語で記述されていますが，十分実用的な回路量で実装できていることが分かります．クリティカルパス長は，回路の動作速度の指標です．合成された回路の性能は，従来の回路設計で行なわれている人間が最適化した回路に比べると決して高くはありませんが，ソフトウェアからの並列化回路合成の容易性という点で十分な成果だと言えます．

## 3 今後の展望

本研究では，C 言語で書かれたソースコードの並列性を抽出し，FPGA 上に並列な演算回路として展開するための機構について述べました．プログラム中の各行ごとの最適な並列度を求める手順を示し，その手法に従って音楽向け電子透かし検出アルゴリズムを並列化したところ，ソフトウェアによる処理に比べて約 23 倍程度の性能向上が達成できました．

これまでの研究では，試行的並列化での面積評価を配置配線まで行なって評価しているので，並列化後の論理回路量の推定に非常に長時間を要しています．今後は，配置配線を行なわずに論理回路量を推定する方法など，実用面での性能向上を図りたいと考えます．また，最近は FPGA に類似したデバイスとして，1 クロックで LSI 内部の回路の書き変えが可能な素子が利用可能になってきています．今回の研究では，FPGA を用いてプログラム単位の並列化を行ないましたが，今後はこのような高速書き変え可能素子を用て関数単位の動的並列化を行ない，更なる処理性能向上を目指す予定です．

## 4 成果リスト

### 論文


1. 榊原 憲宏, 井口 寧, “FPGA を用いたオーディオ電子透かしの超高速検出”, 信学論 (条件付採録)
2. Yuanyuan Zhang, Yasushi Inoguchi, “Influence of Inaccurate Performance Prediction on Task Scheduling in a Grid Environment”, IEICE Transaction Special Issue on Parallel/Distributed Computing and Networking, Vol.E89-D, No. 2, (Accepted)
3. M.M. Hafizur RAHMAN, Yasushi Inoguchi and Susumu Horiguchi, “Modified Hierarchical 3D-Torus Network”, IEICE Transaction on Infomation System, Vol.E88-D, No.2, pp. 177-186, Feb. 2005
4. 井口 寧, 松澤 照男, 堀口 進, “温度予測モデルを用いた重み付けシフトによるウェーハスタック実装の放熱”, 情処 数理モデル化と応用 論文誌, Vol. 44, No. SIG 14 (TOM9), pp. 81-90, Nov., 2003
5. 井口 寧, 堀口 進, “1 次元再帰シフトトーラス相互結合網の拡張”, 情処論, Vol. 44, No. 6, pp. 1521-1524, Jun, 2003
6. 井口 寧, 松澤 照男, 堀口 進, “ウェーハスタック実装の温度予測モデル”, 情処 数理モデル化と応用 論文誌, Vol. 44, No. SIG 7 (TOM8), pp. 1-11, May, 2003


### 口頭発表 (査読付)


1. Wei Sun and Yasushi Inoguchi, “A Scheduler for Local Grid Resource Domain in High Throughput Environment”, IASTED International Conference on Parallel and Distributed

Computing and Networks (PDCN 2006), (Accepted)

2. V.T. Le, X. Jiang, S. Horiguchi, Y. Inoguchi, "A New Fitness Function for GA-based Dynamic RWA Algorithms in Optical WDM Networks", Proc. In IEEE International Conference on Networks, Nov. 16-18, 2005, (Accepted)
3. Yuanyuan Zhang, Yasushi Inoguchi, "Influence of Performance Prediction Inaccuracy on Task Scheduling in Grid Environment", Proc. in 7th Asia Pacific Web Conference (APWEB05), pp. 838-844, Mar. 2005
4. 宮子 陽一, 井口 寧, 堀口 進, "時間評価型オプティカルフローの並列抽出法", 情処 HPCS 2005, P2-4, pp. 8-8, Jan. 2005
5. Yuanyuan Zhang, Yasushi Inoguchi and Hong Shen, "A Dynamic Task Scheduling Algorithm for Grid Computing System", Proc. in Second International Symposium on Parallel Distributed Processing and Application (ISPA04), IEEE, ACM, IPSJ, and Springer, pp. 578-583, Dec., 2004
6. Y. Inoguchi, "Outline of the Ultra Fine Grained Parallel Processing by FPGA", Proc. in IPSJ and IEEE, High Performance Computing in Asia Conference 2004, Workshop on Reconfigurable System for HPC, pp. 434-441, Jul., 2004

## 研究会等

1. 榊原 憲宏, 井口 寧, "FPGA を用いたオーディオ電子透かしの検出", 信学技報, CPSY2004-80, pp.25-30, Jan. 26, 2005
2. 井口 寧, "FPGA を用いた超細粒度並列処理の概要", 信学技報, 第一回 リコンフィギャラブル研究会 論文集, pp.1-6, Sep.18, 2003

## 講演

1. 井口 寧, 榊原 憲広, "FPGA を用いた音楽向け電子透かしのリアルタイム検出とインターネットにおける健全なファイル交換", 第二回 Cray HPC カンファレンス, 大手町サンケイプラザ, Oct. 4, 2005
2. 井口 寧, 榊原 憲広, "FPGA を用いた音楽向け電子透かしのリアルタイム検出とインターネットを介したファイル交換の健全化", 日本セロックシカ テクノロジーセミナー, ホテルパシフィック東京, Jul. 26, 2005

## 特許

1. 井口 寧, 榊原 憲広, "電子透かし検出装置, それを内蔵する中継装置, 及び, 電子透かし検出方法", 特願 2004-230360, 2004