PicturePerfect Workbook

# 機能要件

* OpenCVを利用した以下のフィルタを実行できること
  + 基本的な2D画像フィルタ
  + ファイルとの読み書き
* GPUを利用できること
* 画像，数値をCPU上およびGPU上で扱えること．
  + データは，（画像，数値）×（メモリ種類）で分類する．
  + 画像サイズは1回の実行では1種類のみ扱う．
  + レジスタとアーカイブという2種類のストレージを持つ．
* 同じ設定ならば，何度実行しても同じ結果が出ること（Reproducible）
* 画像サイズは好きなサイズを設定ファイルでしていできること．ただし，１つの実験を通じて１つのサイズのみとする．

## システム要件

* WindowsおよびLinux上で実行できること
* 32bit環境および64bit環境で実行できること
* GPUを活用できる仕組みを持つこと
* １つのスクリプトをマルチコアで並列処理できる仕組みを持つこと
* （非要件）マルチノードの場合，複数のスクリプトを実行する．

## 性能要件

* ディスク読み書きを最小化する工夫を持つ（アーカイブ）
* 前の世代までの結果を再利用して，演算を省略する工夫を持つ（キャッシュ）
* 目標値として，画像サイズ256×256，100個体10タスク10000世代の実行を8時間以内で終えること．
  + 8時間×60分×60秒＝28800秒
  + 世代は並列化できない：10000世代ならば2.88秒／世代
  + タスク，個体は並列化できる：10タスク，100個体ならば，2.88秒＠1000並列，2.88ミリ秒＠1並列，23ミリ秒＠8並列
  + タスクの並列化は並列度の最大値が低い
  + 個体の並列化は並列度の最大値が高い
  + 実行単位ごとに，（スクリプト生成＋演算）の時間が必要
  + 複数のタスクは，スクリプトは入力データ（と中間データ）が違うだけで，スクリプトは同じものを使い回せる 🡪 アーカイブの切り替えで対応できる
  + 複数の個体は，まとめれば実行単位を減らせる 🡪 メリット：オーバーヘッド減らせる，持ち時間を長くとれる，重複した演算を避けられる，デメリット：オーバーヘッド，演算自体が長くなる
  + 世代ごとに，チェックポイントの保存が必要 🡪 実行が速ければチェックポイントは減らしてもいい：デフォルトで200世代ごととする（約10分の処理に相当），設定できるようにする
  + 28800秒 – (Ng/Fc)\*Tc秒－Ng・(Np/Fs)・Ts＝Ng・Nt・(Np/Fs)・Ti  
    Tc: 世代チェックポイント保存時間 [秒]・・・1  
    Ts: 実行単位スクリプト生成時間  
    Ti: 実行単位演算時間  
    Ng: 世代数 [世代] ・・・ 10000  
    Nt: タスク数 [個] ・・・ 10  
    Np: 個体数 [個] ・・・ 100  
    Ns: 世代あたりの実行単位数 [個] ・・・ Np/Fs  
    Fc: チェックポイント保存間隔 [世代] ・・・ 200  
    Fs: 実行単位まとめ数 [個]
  + 28800 – (10000/200)\*1 – 10000 \* (100/Fs) \* Ts = 10000 \* 10 \* (100/Fs) \* Ti  
    28800 – 50 – 1000000Ts/Fs = 10000000Ti/Fs  
    28750Fs – 1000000Ts = 10000000Ti  
    Ti=0.02875Fs – 0.1Ts  
    Tiは，Fsが大きいほど大きい，Tsが小さいほど大きい．Tiを1増やすためにはTsを10減らすかFsを40増やす必要がある．  
    Ts =αFs とすると，  
    Ti=0.02875Fs – 0.1αFs=(0.02875 – 0.1α)Fs  
    よって，Fsは最大値Npを使い，あとは頑張ってスクリプト生成を高速化する，というのがベスト．  
    その場合，  
    Ti=0.02875 \* 100 – 0.1Ts = 2.875-0.1Ts  
    タスクあたり2.8秒くらいで終えないといけない
* 目標値として，画像サイズ256×256，100個体10タスク10000世代の実行を2GB（GPU用は別）のメモリで実行できる
  + 64KB/画像(Gray)，192KB/画像(RGB)
  + 2097152KB 🡪 32768枚分
  + アーカイブ： （タスク）・（3×タスクあたりのデータ）＋（結果））＝10 \* (3N + 1)  
    ※3はカラーなので．結果も複数の可能性がある
  + レジスタ： （最大並列実行数）・（実行単位あたりに必要なレジスタ数）= 8 \* 8 ～ 8 \* 800 (100アグリゲート)  
    ※個体のアグリゲーションをすると必要なレジスタ数は増える
  + キャッシュ（アーカイブの一部として保存する）：最大並列実行数）・（実行単位あたりに必要なキャッシュ数）=8 \* M
  + 10(3N+1) + 6400 + 8M < 32768 🡪 15N + 4M < 13179 N=1…878, M=0…3294, N=3のとき，M=3283．
  + 256x256なら2GBでも可．それ以上は64bit化が必要．

## 保守性要件

* 拡張しても，スクリプト仕様は原則として上位互換を保つ．
* 変更に対して，結果の一貫性を保つ．

## 信頼性要件

* チェックポイント機能－世代が終了するごとにデータを保存しておき，バグ等による中断があった場合にもその続きから再実行できること

## 拡張性要件

## インターフェース要件

* ライブラリとして実装し，スタティックリンクして利用できること．
  + APIを通じて操作する
  + 処理する内容はスクリプトとして記述し，ストリームとして与える．
* コマンドラインからも実行できること．
  + スクリプトファイルをオープンし，ストリームとしてリンクされたライブラリに渡すProxyとして作成する

# 詳細設計

## スクリプト

* スクリプトは以下のようなものとする
  + RISCライクな命令セットによる逐次実行型スクリプト

## スクリプト実行用仮想マシンアーキテクチャー

* スクリプト実行用仮想マシン（PicturePerfectEngine）は，以下のコンポーネントからなる
  + 実行ユニット： CPU Iフィルタユニット，CPU Vフィルタユニット， GPU Iフィルタユニット，GPU Vフィルタユニット， DeviceCopyユニット，DiskAccessユニット，Miscユニット
  + ストレージ： CPUレジスタ，CPUアーカイブ，ディスク，GPUレジスタ，GPUアーカイブ

## 実行ユニット

* 実行ユニットとして，下記の8種類がある
  + **CPU Iフィルタユニット（IF）**：CPUストレージ🡪CPUストレージ（Iデータ）  
    CPUで画像演算処理を実行する．CPUストレージからIデータ，Vデータ，Sデータまたはそのうち複数をロードし，CPUストレージへIデータをストアする．
  + **CPU Vフィルタユニット（VF）**：CPUストレージ🡪CPUストレージ（Vデータ）  
    CPUで数値演算処理を実行する．CPUストレージからIデータ，Vデータ，Sデータまたはそのうち複数をロードし，CPUストレージへVデータをストアする．
  + **GPU Iフィルタユニット（IFg）**：GPUストレージ🡪GPUストレージ（Iデータ）  
    GPUで画像演算処理を実行する．GPUストレージからIデータまたはVデータまたはその両方をロードし，GPUストレージへIデータをストアする．
  + **GPU Vフィルタユニット（VFg）**：GPUストレージ🡪GPUストレージ（Vデータ）  
    GPUで数値演算処理を実行する．GPUストレージからIデータまたはVデータまたはその両方をロードし，GPUストレージへVデータをストアする．
  + **DeviceCopyユニット（DC）**：GPUストレージ⬄GPUストレージ  
    CPU用ストレージと，GPU用ストレージとの間で，データの移動（コピー）を行う
  + **DiskAccessユニット（DA）**：CPUアーカイブ⬄ディスク  
    CPU用アーカイブと，ディスクの間で，データの移動（ロード，セーブ）を行う．スクリプトの命令では直接利用されない．
  + **Miscユニット（MU）**：その他処理

## ストレージ

### 概要

* 用途
  + レジスタ
  + ソースアーカイブ
  + リザルトアーカイブ
  + キャッシュアーカイブ
* メモリ種類
  + CPU
  + GPU
* データ型
* 使われ方
  + 用途ごとに１つのストレージインスタンスを生成する．１つのストレージは全てのデータ型をサポートする．複数のストレージを単一のアドレス空間にマッピングし，実行ユニットから利用される．
  + メモリ種類とデータ型はストレージの中で分かれている．

### 用途

用途には，レジスタ，ソース，リザルト，キャッシュの4種類がある．また，ソース，リザルト，キャッシュの3点を合わせてアーカイブと呼ぶ．

実装は分かれていないので，同じクラスからそれぞれインスタンスを生成する．

リザルトとして使ったストレージを，その次の実行でソースとして使う，などの状況を実現するため，別々のストレージインスタンスを作成する．

#### レジスタ（Register）

* レジスタは，実行中の一時データ保管用に使われるストレージ．原則として演算処理はレジスタ間でのみ行う．
* 実行単位（１つのスクリプト）をまたいだ場合，値の継続性は保証されない．実際に初期化するかどうかは実装依存または実行オプション．
* レジスタから，レジスタへの書き込みは，同じメモリタイプ，同じデータタイプに限り可能．それ以外は実行時エラーInvalidStorageAccess．

#### アーカイブ（Archive）

ストリームをまたいでアクセスできるデータのセット．スクリプトの外で，ディスクから読み込みまたはディスクへの書き込みを行うことが出来る．

#### ソース（Source）

* 参照は自由に行えるが，更新は行えない．（内容はAPIか定義ファイルによって，スクリプトの外で設定しておく必要がある）
* Sourceアーカイブから，レジスタへの書き込みはOK．
* レジスタから，Sourceアーカイブへの書き込みは不可（実行時エラー InvalidStorageAccess）．
* Sourceアーカイブから，ディスクへの書き込みは不可（実行時エラーInvalidStorageAccess）．
* Sourceアーカイブの更新は，事前定義されるアーカイブコンテンツが，スクリプト実行前に，ディスクから読み込まれる場合のみ．それ以降の更新は行えない（実行時エラーInvalidStorageAccess）．

#### リザルト（Result）

* 更新は自由に行えるが，参照は行えない．（内容はAPIか定義ファイルによって，スクリプトの外で保存する必要がある）．
* Resultアーカイブから，レジスタへの書き込みは不可（実行時エラーInvalidStorageAccess）
* レジスタから，Resultアーカイブへの書き込みはOK．
* Resultアーカイブから，Resultアーカイブへの書き込みはOK．
* Resultアーカイブから，Result以外のアーカイブへの書き込みは不可（実行時エラーInvalidStorageAccess）
* Resultアーカイブから，ディスクへの書き込みはOK．
* ディスクから，Resultアーカイブへの書き込みは不可（実行時エラーInvalidStorageAccess）

#### キャッシュ（Cache）

* 中間ノードの結果を保存しておき，スクリプトをまたいで利用するためのキャッシュ用．レジスタとのやりとりが可能．
* Cacheアーカイブから，レジスタへの書き込みはOK．
* レジスタから，Cacheアーカイブへの書き込みはOK．
* Cacheアーカイブから，Cacheアーカイブへの書き込みは不可（実行時エラーInvalidStorageAccess）
* Cacheアーカイブの内容は（原則として）ディスクには保存されない．

### メモリ種類

* メモリ種類には，CPU，GPUの2種類がある．

### データ型

以下のデータタイプをサポートする

* 0h V1D (value, double)
* 4h I1B (image, byte, 1 component)
* 5h I3B (image, byte, 3 components)
* 6h I1D (image, double, 1 component) <future>
* 7h I3D (image, double, 3 components) <future>
* 9h S (Unicode text)
* Ah A1D (array, double) 可変長配列型(サイズ0以上)．平均値，標準偏差，最大値，最小値もキャッシュしている．
* Fh L (List) データアドレスのリスト（サイズ0以上）。データアドレスを複数まとめて扱うために用いる。
* ストレージクラスと実装
* 10h V1D double
* 14h I1B cv::Mat(CV\_8UC1)
* 15h I3B cv::Mat(CV\_8UC3)
* 16h I1D cv::Mat(CV\_64FC1)
* 17h I3D cv::Mat(CV\_64FC3)
* 19h S std::wstring
* 1Ah A1D Vector (std::vector<double>の派生で，平均値などをキャッシュする)
* 20h V1Dg double
* 24h I1Bg cv::ocl::OclMat(CV\_8UC1)
* 25h I3Bg cv::ocl::OclMat(CV\_8UC3)
* 26h I1Dg cv::ocl::OclMat(CV\_64FC1)
* 27h I3Dg cv::ocl::OclMat(CV\_64FC3)
* ※GPUはText, Arrayをサポートしない．

### アドレスマッピング

* ストレージは，ストレージ種類，データタイプおよびメモリ種類によって，格納されるアドレス（番号）が決まっている．
* １つのストレージは，10000000h個のエントリー（アドレス0000000h～FFFFFFFh）を持ち，その中でデータ型に分かれている．
* アドレス
  + 01000000h – 010FFFFFh V1D CPUレジスタ
  + 01400000h – 014FFFFFh I1B CPUレジスタ
  + 01500000h – 015FFFFFh I3B CPUレジスタ
  + 01600000h – 016FFFFFh I1D CPUレジスタ <future>
  + 01700000h – 017FFFFFh I3D CPUレジスタ <future>
  + 01900000h – 019FFFFFh S (Unicode text string) CPUレジスタ
  + 01A00000h – 01AFFFFFh A1D CPUレジスタ
  + 02000000h – 020FFFFFh V1Dg GPUレジスタ
  + 02400000h – 024FFFFFh I1Bg GPUレジスタ
  + 02500000h – 025FFFFFh I3Bg GPUレジスタ
  + 02600000h – 026FFFFFh I1Dg GPUレジスタ <future>
  + 02700000h – 027FFFFFh I3Dg GPUレジスタ <future>
  + 11000000h – 110FFFFFh SV1D CPUソースアーカイブ
  + 11400000h – 114FFFFFh SI1B CPUソースアーカイブ
  + 11500000h – 115FFFFFh SI3B CPUソースアーカイブ
  + 11600000h – 116FFFFFh SI1D CPUソースアーカイブ <future>
  + 11700000h – 117FFFFFh SI3D CPUソースアーカイブ <future>
  + 11900000h – 119FFFFFh SS CPUソースアーカイブ (Unicode text string)
  + 11A00000h – 11AFFFFFh SA1D CPUソースアーカイブ
  + 12000000h – 120FFFFFh SV1Dg GPUソースアーカイブ
  + 12400000h – 124FFFFFh SI1Bg GPUソースアーカイブ
  + 12500000h – 125FFFFFh SI3Bg GPUソースアーカイブ
  + 12600000h – 126FFFFFh SI1Dg GPUソースアーカイブ <future>
  + 12700000h – 127FFFFFh SI3Dg GPUソースアーカイブ <future>
  + 21000000h – 210FFFFFh RV1D CPUリザルトアーカイブ
  + 21400000h – 214FFFFFh RI1B CPUリザルトアーカイブ
  + 21500000h – 215FFFFFh RI3B CPUリザルトアーカイブ
  + 21600000h – 216FFFFFh RI1D CPUリザルトアーカイブ <future>
  + 21700000h – 217FFFFFh RI3D CPUリザルトアーカイブ <future>
  + 21A00000h – 21AFFFFFh RA1D CPUリザルトアーカイブ
  + 22000000h – 220FFFFFh RV1Dg GPUリザルトアーカイブ
  + 22400000h – 224FFFFFh RI1Bg GPUリザルトアーカイブ
  + 22500000h – 225FFFFFh RI3Bg GPUリザルトアーカイブ
  + 22600000h – 226FFFFFh RI1Dg GPUリザルトアーカイブ <future>
  + 22700000h – 227FFFFFh RI3Dg GPUリザルトアーカイブ <future>
  + 51000000h – 510FFFFFh CV1D CPUキャッシュアーカイブ
  + 51400000h – 514FFFFFh CI1B CPUキャッシュアーカイブ
  + 51500000h – 515FFFFFh CI3B CPUキャッシュアーカイブ
  + 51600000h – 516FFFFFh CI1D CPUキャッシュアーカイブ <future>
  + 51700000h – 517FFFFFh CI3D CPUキャッシュアーカイブ <future>
  + 51A00000h – 51AFFFFFh CA1D CPUキャッシュアーカイブ
  + 52000000h – 520FFFFFh CV1Dg GPUキャッシュアーカイブ
  + 52400000h – 524FFFFFh CI1Bg GPUキャッシュアーカイブ
  + 52500000h – 525FFFFFh CI3Bg GPUキャッシュアーカイブ
  + 52600000h – 526FFFFFh CI1Dg GPUキャッシュアーカイブ <future>
  + 52700000h – 527FFFFFh CI3Dg GPUキャッシュアーカイブ <future>

### 実装ノート

* ストレージ自体は，0000000h～FFFFFFFhの範囲のアドレスしか認知しない．

## 実行手順

* PPEは以下の手順で実行する
  + PPEインスタンスの生成
  + 画像サイズ，およびその他の実行オプションの設定
  + アーカイブインスタンスの生成
  + アーカイブの初期化
  + 実行リストのコンパイル
  + 実行リストの実行（ソース，リザルトアーカイブを渡す．キャッシュはオプション）
  + アーカイブの保存
* コマンドラインツールの場合は，スクリプトファイルをロードし，ブロックに分けて，APIを呼び出す．
* ライブラリの場合は，利用者がAPIを呼び出す．

API

* すべてのAPIはリエントラント．
* PPE PicturePerfectEngine::createInstance() PPEインスタンスを作成する
* void PicturePerfectEngine::setImageSize(x, y) 画像サイズをセットする．
* Archive PicturePerfectEngine::createArchive()
* Archive::setValue(addr, value) valueに\_\_BLACK\_\_や\_\_WHITE\_\_を与えると，0や1をセットする
* Archive::setString(addr, value)
* Archive::loadImage(addr, filename) filenameに\_\_BLACK\_\_や\_\_WHITE\_\_を与えると，黒や白で初期化する
* Archive::saveImage(addr, filename)
* Archive::load(istream) ストリームに従って，アーカイブを初期化する．ストリームはBegin Archive～End Archiveの中身のみ
* Archive::save(istream) ストリームに従って，アーカイブを保存する．ストリームはBegin Archive～End Archiveの中身のみ
* InstructionList PicturePerfectEngine::compile(istream) ．ストリームに従って，実行リストをコンパイルする．ストリームはBegin Archive～End Archiveの中身のみ
* void PicturePerfectEngine::execute(instructionList, archive) 実行リストと，アーカイブは同じPPEインスタンスから生成されたモノでなければならない．

# スクリプト

* スクリプトを与えるが，実際の実行はスクリプトをコンパイルして，数値データ列に変換してから実行する．理由は同じスクリプトをアーカイブを取り替えて何度も実行することが多いため，時間のかかる文字列処理をまとめ，命令判定に伴う条件分岐もできる限り一度だけの実行にとどめるため．
* スクリプトには，アーカイブ定義と実行定義の2種類を書く

## スクリプト命令

* Size X, Y
  + 画像サイズを指定する．
  + Size 320, 240
* Begin [Archive|Execution]
  + ブロックを開始する
  + Begin Archive
* End [Archive|Execution]
  + ブロックを終了する
  + End Archive

## 演算

* 各ストレージは，宣言なしでいきなり使える．アドレスによって型制限がある．そのアドレスのストレージがValid=Falseの場合の参照は実行時エラー．

## 命令

* 全ての命令は個別のOpCodeを持っており，対応するMnemonicがある．
* スクリプトの可読性と書きやすさのため，Mnemonicのうち，型違いまたはCPU/GPUの違いがあるが機能的に同一のものは，単一のAliasで指定できる．
* 実行はすべてOpCodeで行うので，Mnemonic（またはAlias）で書かれたスクリプトは，実行に先立ちコンパイルされる．
* スクリプトは各行1命令で記述．

OpCode定義

* OpCodeが決定すると，実行ユニットと演算内容が決まる．
* 実行ユニットとは，処理ごとに決まったクラスのインスタンス．
* 実行エンジンは，演算処理を行う．複数の演算に対応することもある．
* 実行エンジンに処理依頼する場合は，どの演算を行うか指定する必要がある．（0からの序列で指定する）
* 実行エンジンの処理は，リエントラントにする．実装上リエントラントが不可能な場合は，内部でクリティカルセクションを作る．
* OpCodeの実際の値は，PPEインスタンスの作成時点で決定する．
* PPE実装クラスのimpl名前空間内に，実行エンジンクラスのインスタンスを静的に作る．それを登録する．初期化が必要なモノは，OpCode決定プロセス内で初期化する．

実行エンジン１（SakuraEngine）

* 最初のエンジン．

# コンパイル手順

* 命令デコード
  + 引数を展開し，数値アドレスに直す．
    - 末尾が）で終わる場合は，相対指定とみなし，V1D(xx)のようなストレージクラスごとにxxには0から始まる番号を入れたものと会社解釈する．ストレージクラスが不明なモノだった場合は，コンパイルエラーInvalidStorageClass．xxが可能な範囲（0h～FFFFFh）を超えている場合は，コンパイルエラーRegisterOutOfRange．
    - ）ではない場合は，絶対指定とみなし，全体を数値として解釈する．0xで始まる場合は16進として，それ以外は10進数として解釈する．値が未定義の領域だった場合は，コンパイルエラーRegisterOutOfRange．  
      具体的方法：上3桁が存在するかを調べる．その後，下5桁が0h～FFFFFhに入っているかを調べる．
  + MnemonicをOpCodeに変換する
    - 手順
      * （Mnemonic, OpCode）テーブルを引き，OpCodeに変換する
      * テーブルになかった場合は，Aliasであると仮定し，引数のアドレスから型を判断し，型から接頭辞および接尾辞を自動的に付加し，再度テーブルを引く
      * 存在しないMnemonicだった場合は，コンパイルエラーInvalidInstruction
      * OpCodeは，実行ユニットによって実装依存とする．そのため，必ず実際に実行する実行ユニットでコンパイルする必要がある．
      * ただし，End命令のみのは常に0番にアサインされる
  + TODO　OpCodeから，Operandの入出力を調べる
  + TODO　Operandの入出力ストレージの組み合わせから，実行ユニットを決定する．
    - 具体的方法：出力と，すべての入力について，実行ユニットを調べる．１つでも実行ユニットなしがあるか，すべての実行ユニットが一致していない場合は，コンパイルエラーInvalidStorageAccess
  + TODO　OpCodeから，この命令の実行ユニットを調べる．
  + TODO　OpCodeから得た実行ユニットと，ストレージ組から得た実行ユニットが一致しているか調べる．一致していない場合は，InvalidExecutionUnit
* ストレージショートカットテーブルの構築
  + Operandとして指定されたストレージアドレスは，登場順にストレージショートカットテーブルに登録する．すでにそのアドレスが登録されていたら，既存のエントリーを使う．  
    ショートカットテーブルは（アドレス，データへのポインタ）からなる．データへのポインタは実行時にアーカイブと組み合わされた時点でセットされる．
  + 登録された場所（16から始まる連番）を，新しいOperand（ShortCutIndex）とする．(0～15は，実行ユニットのために予約されている)
* アセンブル
  + OpCodeとOperand（ShortCutIndex）の列（Program）に変換する
  + Programと，ShortCutTableを，Executableとして，返す．

# 実行手順

* レジスタファイルの生成
  + レジスタファイルはPPEインスタンスが生成された時点で作る．同じインスタンスの再利用では，同じレジスタファイルを再利用する．
* 実行形式（Executable）とアーカイブ（Archive）を受け取る
  + 内部的にレジスタファイル（RegisterFile）も受け取る
* ショートカットテーブルの構築
  + Program中のShortCutTableを参照し，ポインタリストを更新する．
  + 具体的には，ShortCutTableの各エントリーのアドレスで指定されたストレージの，データポインタを取得し，エントリーのポインタにセットする．
  + 0～15は，Archiveの各型のFFFFFh番に必ずアサインされ，実行ユニットが作業領域として使用する．
* 実行
  + Programを前から順に実行する．
  + OpCodeを取得し，その番号の関数オブジェクトを実行する．
    - 関数オブジェクトには，ProgramCounter（ProgramのIterator）を渡す．
    - Operandを，ShortCutTableを参照して，データアドレスに変換するため，T\* PicturePerfectEngine::getData<T>()を使用する．

アーカイブ定義ファイル

* アーカイブの内容を定義する，テキストファイル．拡張子は通常.ppe.txt（実行定義と混在している場合）または，.ppearc.txt（アーカイブ定義のみの場合）
* 以下のような文法
  + Begin ArchiveとEnd Archiveの行の間に書く
  + アドレス 引数
  + アドレスの与え方は，番号絶対値または，識別文字（相対数値）のどちらかが使える．数値は十進数または十六進数(0xで始める)が使える．  
    例： ホストメモリのV1Dソース，番号123の場合 285212795，0x1100007B，SV1D(123)，SV1D(0x7B)のいずれか．
  + 引数は，即値，組み込み定数，またはファイルのいずれか．次の順で判定．
    - ファイル（Iのみ）：@ではじめる．@xxxx.yyyと書くと，ソースの場合はxxxx.yyyから読み込み，リザルトの場合はxxxx.yyyに書き込む．  
      型違いの場合はInvalidStorageAccess，ファイルアクセスの失敗はFileOpenFailure．  
      サイズは，設定されたサイズに合わせて拡大・縮小される．
    - 組み込み定数：\_\_BLACK\_\_（すべて0，S・A1D以外），\_\_WHITE\_\_（すべて1または255，S・A1D以外），\_\_EMPTY\_\_（空集合，A1Dのみ）  
      型違いの場合はInvalidStorageAccess
    - 即値（V1D，S，A1Dのみ）：実数値または文字列を書く．配列はカンマ区切りで複数の実数値を書く．  
      型違いの場合はInvalidStorageAccess．
  + ファイル名は，以下のルールで探す
    - １．‘/’で始まる場合は，絶対パスとして探す
    - ２．スクリプトファイルからの相対
    - ファイル名処理は，PicturePerfectEngine（ベースクラス）に，用意する

# 実行リスト

* uint16\_tのリスト

# コマンドラインツール

* コマンドライン引数
  + ppe ScriptFileName
  + ppe –i ScriptFileName
  + 引数がない場合
* 起動時にスクリプトの場所をベースアドレスとして保存する

# 標準スクリプト

* 目的
  + 演算機能チェック用
  + ストレージ機能チェック用
  + 性能チェック用
  + エラー処理チェック用
* 演算機能チェック用
  + すべての命令を，ArchiveまたはRegisterで実行する．
  + ゴールデン・データを記録しておき，以降機能拡張の場合に照合する．
* ストレージ機能チェック用
  + すべてのストレージにRead/Writeアクセスする
* 性能チェック用
  + これまでの経験から得られる命令ミックスで実行する．
  + 実行時間により，全体の実行時間，高速化ターゲット，等の材料とする．
* エラー処理チェック用
  + エラーが起こるすべての命令・ストレージ組み合わせで実行する

# トレーニング

* すべてのオペレータを一度ずつ実行するトレーニング関数を用意する
* OpenCLバージョンのデバイスコードのコンパイル，CPUコードのキャッシュへの配置を主な目的とする．
* アーカイブ